Master Thesis

Scalable Real-Time Product Recommendation based on Users Activity in a Social Network

Author(s):
Haspra, Michael

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006450779

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 7
Systems Group, Department of Computer Science, ETH Zurich

Scalable Real-Time Product Recommendation based on Users Activity in a Social Network

by
Michael Haspra

Supervised by
Dr. Maria Grineva, Prof. Dr. Donald Kossmann

October 2010–April 2011
Abstract

The web has become a real-time communication medium, used by a large amount of people, in ever-increasing parts of their daily life. This new usage pattern gives advertisers and marketeers great opportunities to learn about their customers' temporal interests, thoughts and current context.

Despite it’s a well known fact that this information is extremely valuable for advertising and product recommendation, online advertising is adapting only slowly. This is mainly due to the fact that it’s not clear what information is valuable to use, the huge amount of produced data and the lack of efficient models to process this data.

This thesis describes an approach to implement *Scalable Real-Time Product Recommendation based on Users Activity in a Social Network*. The products are taken from Amazon.com and the used social network is the microblogging platform Twitter. It presents an implementation of this approach on top of the key-value database Cassandra, using a system called Triggy. Triggy extends Cassandra with incremental Map-Reduce tasks for push-style data processing.

Use cases that require high-performance analysis of large amounts of data, are the showpiece of every stream processing engine. These engines are built to process massive amounts of data in very short time. Therefore, this thesis contains a comparison between four state-of-the-art distributed stream processing engines and the implementation with Triggy. It’s showed that the analyzed use case has various properties that make it’s implementation in a stream processing engine impossible.

Finally, a demo application is presented, to show the described approach.
Acknowledgement

I would like to express my gratitude to my supervisor Dr. Maria Grineva for her help and her guidance through this thesis. I’d also like to thank Maxim Grinev and Martin Hentschel for their help with Triggy and Cassandra. Furthermore, I thank Prof. Dr. Nesime Tatbul, Dr. Peter Fischer and Roozbeh Derakhshan for their inputs regarding stream processing engines. Finally, I thank Stefan Müller and Adrian Elkuch for all the reviews. All your help was very much appreciated.
## Contents

1 Introduction .................................................. 8  
   1.1 State of the Web ....................................... 8  
   1.2 Situation in Online Advertising ......................... 8  

2 Problem Statement ........................................... 10  
   2.1 Motivating Example ................................... 10  
   2.2 Product Recommendations Perspective .................. 10  
   2.3 Systems Perspective ................................... 11  

3 Our Approach .................................................. 13  
   3.1 Datasets and Recommendation Model ....................... 13  
      3.1.1 Twitter Stream Data .............................. 13  
      3.1.2 Product Database ................................. 13  
      3.1.3 The Recommendation Model ....................... 14  
   3.2 Selection of the System ................................ 15  
      3.2.1 Triggy: Push-style incremental data processing based on a distributed Key-Value Store ............ 15  
      3.2.2 Comparison with other Systems .................... 16  

4 Related Work .................................................. 18  
   4.1 Product Recommendations ................................ 18  
   4.2 Twitter Analysis .................................... 18  
   4.3 Time- and context-aware Recommendations ................ 18  

5 Contributions ................................................ 19  

6 Solution ..................................................... 20  
   6.1 Overview of the System Architecture ...................... 20  
   6.2 Implementation ........................................ 21  
      6.2.1 Retrieval Tier ................................... 21  
      6.2.2 Processing Tier ................................... 24  
      6.2.3 Presentation Tier .................................. 35  
   6.3 Demo Application ....................................... 36  

7 Experiments ................................................... 38  
   7.1 Keyword Extraction .................................... 38  
      7.1.1 Test Setup ....................................... 38  
      7.1.2 Results .......................................... 39  
      7.1.3 Conclusion ....................................... 40  
   7.2 Performance Experiments ................................ 40  
      7.2.1 Test Setup ....................................... 40  
      7.2.2 Querying vs. Pushing .............................. 40  
      7.2.3 Is it better to have one big Map-Reduce Task or several small ones? ........................... 41  
      7.2.4 Scalability ....................................... 42  
   7.3 Comparison with Stream Processing Engines ............... 43  
      7.3.1 Selection of the Stream Processing Engines .......... 43  
      7.3.2 Implementation in Esper .......................... 44  
      7.3.3 Observations ..................................... 50
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Motivating Example: We want to find overlappings between user interests and interests that are related to a product</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Conceptual implementation of the matching algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Differences between the two Map-Reduce programming models</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>System architecture overview</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Implementation of the matching algorithm with Map-Reduce tasks</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Description of the score-calculation for a product-user matching</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Column families before any tweet was processed</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>Column families after the first tweet has been processed</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>Column families after the second tweet has been processed</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>Column families after the third tweet has been processed</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>Column families after the recommendation has been made and the model has been resetted</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>Screenshot of the input form for the simulation</td>
<td>36</td>
</tr>
<tr>
<td>13</td>
<td>Screenshot of the running demo application</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>Screenshot of a recommendation together with its justification</td>
<td>37</td>
</tr>
<tr>
<td>15</td>
<td>Querying vs. Pushing test results</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>Comparison of different numbers of Map-Reduce tasks test results</td>
<td>42</td>
</tr>
<tr>
<td>17</td>
<td>Scalability test results</td>
<td>43</td>
</tr>
<tr>
<td>18</td>
<td>Matching-Algorithm implementation in the Esper Stream Processing Engine</td>
<td>47</td>
</tr>
<tr>
<td>19</td>
<td>State of the Esper streams and windows after the insertion of the first tweet</td>
<td>48</td>
</tr>
<tr>
<td>20</td>
<td>State of the Esper streams and windows after the insertion of the second tweet</td>
<td>49</td>
</tr>
<tr>
<td>21</td>
<td>State of the Esper streams and windows after the insertion of the third tweet</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Tables

1. Results of the analysis of four state-of-the-art stream processing engines. 51
Listings

1 Description of a Cassandra column family .................................. 25
2 Description of a cassandra column family with two rows and some columns ....................................................... 25
3 Specification of a Map-Reduce task ........................................... 26
4 Skeleton of an incremental Map-Reduce task ............................... 26
5 If-condition to simulate a tweet match ................................. 40
6 Form of a tweet-event in the Esper implementation ................... 44
7 A query to find a matching and insert it into the next window .................. 45
8 Form of a matching-event in the Esper implementation ............... 45
9 A query to sum up and score matching counts per user, product and keyword .................................................. 45
10 Form of a score-event in the Esper implementation .................... 45
11 A query to sum up the scores from different keywords for a user-product recommendation .................................. 46
12 Form of a recommendation-event in the Esper implementation .... 46
13 Query to delete events from a window on recommendation ........... 46
1 Introduction

1.1 State of the Web

In recent years, the web changed from a relatively static collection of web pages to a dynamic source of real-time and user-generated information. This can be observed for the way how people interact with the web as well as for the web itself. In short, the web becomes more and more real-time.

For the users, one observes that they share their thoughts and information about their activities in real-time. Prominent examples – of this real-time interaction – are millions of tweets during popular events (Super Bowl 2011 [33], Inauguration of Barack Obama [32], . . .) or the fast emerging usage of check-in services such as Facebook Places [36] or Foursquare [19].

On the other hand, the web itself provides data (Google Instant Search [41]) and information from other users (Facebook [18], Twitter [45], . . .) in real-time too.

This shift to real-time is not the only novelty in user-web interaction. Thanks to mobile devices – together with social media applications – people use the web not only in real-time, but for an ever-increasing part of their daily life.

This tight integration of the web in everyday situations, allows new applications and services to respond with context-dependent and situation-relevant information. In particular, they can reply with real-time advertisement and product recommendations, based on recent user activity as well as the immediate context.

However, these new opportunities also bring complex challenges. A particularly crucial one is how to deal with the huge amount of data that has to be analyzed. On Twitter, people publish more than 1’000 tweets per second as an average workload. For popular events, this number can even rise a lot higher. More than 4’000 tweets per second were recorded during the Super Bowl 2011 and almost 7’000 tweets per second on New Year’s Eve 2011, in Japan [1].

In addition, the response time is increasingly important for a service – to provide valuable and up-to-date information. Therefore, this huge amount of data has to be processed in real-time and, as it will be stated later, in an incremental and scalable way.

1.2 Situation in Online Advertising

The usage of user-generated content and social media data for product recommendations and advertisement has got significant attention in research. A prominent example is collaborative filtering [29], which has led to a number of applications, such as GroupLens [39], the Google News Reader [13] or Amazon Recommendations [29]. Online shopping vendors slowly begin to make more use of these new information and techniques. As an example, Amazon [4] provides a connection to Facebook, for recommendations from and for friends.

With these techniques, one generally tries to recommend products based on long term analytics of user behavior and the user’s social context. However, it’s a known fact in consumer research that the immediate context of a person plays a significant role in judgment about products and advertisement [35].

This – importance of the immediate context – also holds for techniques to improve online advertisement campaigns, such as behavioral targeting [44].
Jun Yan et al. [49] showed that one can “significantly improve ads CTR\(^1\)”, by taking “the users’ recent and focused interests” into account. Different projects used this knowledge (about the importance of the immediate context) to improve product recommendations in retail-stores [16, 28]. However, they used special hardware in the store itself – in contrast to a web-based solution - and do not use social media data.

We are not aware of an online shopping website, or a web-based recommendation service, that makes use of social media data in real-time.

\(^1\)Click-Through-Rate
2 Problem Statement

2.1 Motivating Example

People use twitter to share information about their thoughts and activities during every-day life. From these tweets, one can learn about a person’s general and – even more important – temporal interests. To do so, one has to monitor this person’s tweets and try to identify these interests.

Figure 1 shows a motivating example. The user Michael tweeted about taking pictures, going to a photography exhibition and bike riding. From these tweets, we can infer that he is interested in bikes, photography and pictures. On the other hand, we have a camera in our product database for which we identified pictures, photography, canon and snapshots as potential user interests this camera is related to.

So we monitor Michael’s tweets and record his interests as well as overlappings between his interests and the potential user interests, which we have identified for our products. As soon as we think that the overlapping between his interests and the camera’s potential user interests is strong enough, we immediately recommend this product to him.

![Figure 1: Motivating Example: We want to find overlappings between user interests and interests that are related to a product](image)

One can easily imagine that the amount of data, one has to retrieve and process for such a task, is huge. Also extracting a user’s interests from tweets, generating potential user interests for products and find overlappings between them is non-trivial.

We want to address these problems from two perspectives, (1) from a product recommendation perspective and (2) from a systems perspective.

2.2 Product Recommendations Perspective

We will analyze how we can define a simple and flexible model for real-time product recommendations, based on social media data. Therefore, we define the following general use case, for real-time product recommendations based on users activity in social media:

- **Monitor User Interests**
  
  One needs to monitor a user’s general and temporal interests. This is done
by capturing his social network activity, more precisely the generated text-content.

- **Extract Product Keywords**
  For products, potential user interests for which a product is relevant for need to be identified. This is done by automatically extracting keywords (per product) that represent these interests.

- **Match User Interests to Product Keywords**
  The user interests will be matched to product keywords. This matching has to be:
  
  - **Incremental**
    It has already been stated that the amount of data will be huge. If one wants the system to react in real-time, traditional batch processing of this data is not an option. The matching algorithm must be incremental.
  
  - **Over a (potentially) very long time**
    The time-window size for this accumulation – of interests and matchings – depends on the user’s activity and the product. It will be different for every combination of user and product and can be very long. To give an example, we can collect more data for users who use Twitter more often, so we can learn about their interests faster than about the interests of a seldom user.

- **Do real-time Recommendations**
  For some recommendations, the time elapsed between the last user activity and the recommendation can be very important. For this reason, a recommendation has to be done as fast as possible.

- **Redo Recommendations**
  A recommendation can be made several times for the same product. Therefore, it must be possible to reset the model that will implement this use case.

### 2.3 Systems Perspective

We will analyze which system is suitable to implement the recommendation-model that we are going to propose for the previously defined use case. For such a system, the following requirements can be identified:

1. **Incremental Data Processing**
   The system has to allow an implementation which processes the incoming data in an incremental way.

2. **Scalability**
   The system has to be horizontally scalable. This has to be possible without complex tuning by the programmer. In particular, the implementation of the recommendation-model must not depend on whether it runs on a single machine or on a cluster.
3. **Fault-Tolerance**
   User interests will be monitored for a (potentially) very long time. To assure that this data is not lost, the system has to provide fault-tolerance.

4. **Persistent Storage**
   Since the time for which the user interests are monitored can be very long, a big amount of data will be accumulated. This data can easily exceed the main memory capacity of a processing machine. Therefore, the overflowing data has to be saved on persistent storage.

5. **Flexible and easy Programming Model**
   The programming model has to be flexible enough to implement the stated use case. On the other hand, it has to be easy enough to express it in a clear and simple way.

6. **Retrieval of Reference-Data/Provenance**
   Retrieval of reference-data is a key requirement for the justification of recommendations, as well as statistical analytics. That means, whenever an application produces intermediate values, it must be as easy as possible to retrieve them at a later time.

7. **Resettable**
   After a recommendation has been done, it must be possible to reset all the intermediate values (such as the accumulation of user interests), without violating the previous requirements.
3 Our Approach

3.1 Datasets and Recommendation Model

3.1.1 Twitter Stream Data

We take Twitter [45] as a source of social media data. Twitter is easy accessible over it’s Streaming API [6] and provides enough data to test the identified system requirements. Moreover, it’s real-time nature and wide usage in mobile devices suit our general use case very well. For all users, we are interested in, we monitor their tweets.

Doing so, we process a tweet – to identify potential user interests – as soon as it has been published.

3.1.2 Product Database

We create a product database based on Amazon [4], the largest online retailer on the web. Amazon sells millions of products in a high variety of categories. Also, most of the products come with a rich source of additional information such as product descriptions, reviews or user comments. In addition, Amazon provides efficient access to product- and category-information over it’s Product Advertising API [5].

For every product in our database, we create a set of keywords which identify the potential user interests this product could be recommended for. We examined different keyword extraction strategies (AlchemyAPI [3], Yahoo! Term Extraction [17]...) and different sources to extract keywords from (Product Descriptions, Reviews, User Comments...). The best strategy is to take user generated tags\(^2\). We use these tags directly as keywords for our products. Conveniently, it is also indicated how many customers voted for the same tag. These counts provide a very natural and straight-forward weighting scheme.

This leads to the following algorithm for creation of the product database with keywords:

1. Download the category structure for a department\(^3\) on Amazon (for example *Sports*), using the Amazon Product Advertising API.

2. Download a portion of the most popular products for every category in the department, together with product details such as name, id and the URL of the product’s webpage.

3. Download the product-webpage for every product and extract the user-generated tags together with the count for each tag.

4. Extract keywords for each product by taking the tags and set the keyword’s weight according to the count.

---

\(^2\)Amazon provides a function which allows customers to add a tag to a product. These tags should help to categorize products.

\(^3\)Amazon is divided into departments. These departments represent a very high level category of products. Examples of departments are *Books*, *Electronics* or *Sports & Outdoor*. 
3.1.3 The Recommendation Model

To make a recommendation – for a certain product – to a given user, the overlappings between the user’s interest and the potential user-interests this product is relevant for need to be identified. We propose a threshold-based, keyword-matching model for this task.

As already stated, the potential user-interests for a product are represented by keywords and the relevance is given by the weights of the keywords. For users, we just match these keywords to their tweets. If a keyword occurs in a tweet, we say the user is interested in this topic and thereby also in the products assigned to this keyword. The weight of the keyword indicates the strength of the interest.

We monitor these interests – for products – over time. The intensity of the user interest in a product is defined to be the sum of the weights from all matching keywords. To diminish the relevance of one (potentially bad) keyword, a decreasing weighting scheme is used for repeatedly matching keywords. We do this because we noticed that recommendations based on matchings from several keywords are more reliable than recommendations based on the repeated occurrence of only one keyword. Note that with this approach a user can have interest in a product based on different (potentially unrelated) keywords. These weights are accumulated until the sum has reached a predefined threshold. Then, we do the recommendation and reset the model. By resetting the model, we are able to recommend the same product again at a later time.

Figure 2 shows a conceptual implementation of the matching algorithm. The data comes in from three sources. (1) User Data contains information about the users we are monitoring. (2) Product Data is all information about products. That are product-details on one hand and weighted keywords on the other hand. (3) Tweet Data are the tweets, these come in as stream on runtime, whereas the other data is loaded on system startup. The matching algorithm then works in four steps.

1. Split the tweet text and match every word against the set of keywords. For every matching, we produce an output containing information about the user, the tweet, the product, the keyword and the weight.

2. Sum up the matchings for the same product and user per keyword.

3. Calculate the score for the user-product matching. To diminish the effect of repeatedly occurring keywords, we decrease their weights if they play part in a user-product matching several times.

4. Compare the score with the threshold for the product. If the score is above the threshold, do a recommendation and reset the model.

---

4The threshold can be set individually for every product.
The proposed recommendation model is simple. On the other hand it is very flexible and extensible (for example with a more complex aggregation formula or additional sources of information). Also, it is easy to implement it in an incremental way and the recommendations we get with this model are already very promising.

3.2 Selection of the System

3.2.1 Triggy: Push-style incremental data processing based on a distributed Key-Value Store

The application is implemented with Triggy [21]. Triggy provides an implementation for push-style incremental data-processing on top of Cassandra [27], a distributed key-value store. Therefore, it uses a modified implementation of Map-Reduce [14], a well-established programming model for batch-style distributed data processing. The modification – which allows this push-style incremental data processing – applies to the reduce function. In the original Map-Reduce
Traditional Map – Reduce:
map: \((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)
reduce: \((k_2, \text{list}(v_2)) \rightarrow (k_2, \text{agg})\)

Modified Map – Reduce:
map: \((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)
reduce: \((k_2, v_2, \text{agg}_{old}) \rightarrow (k_2, \text{agg}_{new})\)

Figure 3: Differences between the two Map-Reduce programming models

model, reduce is applied to a list of values and returns one aggregated value\(^5\). The modified reduce function is incremental. It applies the input values incrementally to the aggregated value as soon they are available. This makes it suitable for push-style data processing.

Figure 3 shows the differences between the traditional Map-Reduce programming model and the incremental modification. (It’s a slightly modified version of the according figure in [21])

To explain the differences a bit more, in traditional Map-Reduce the map method gets an input tuple \((k_1, v_1)\) and produces a list of intermediate values \(\text{list}(k_2, v_2)\). Since there are potentially several, parallel executions of map, there will be a number of these intermediate lists \(\text{list}(k_2, v_2)\), for the same key \(k_2\). The Map-Reduce framework takes all these lists and groups the intermediate values by \(k_2\). These groups are then send to the reduce functions, which aggregates the values \(v_2\) (batch-style).

The modified Map-Reduce implementation, does not wait until all intermediate tuples have been produced. It applies the reduce function to the intermediate tuples as soon as they are produced, together with an aggregation-value \((k_2, v_2, \text{agg}_{old})\) (incremental).

To use these incremental Map-Reduce tasks, the programmer first has to implement the two methods map and reduce. Then he has to specify a column family\(^6\) and (optionally) a column for which the task has to be executed. Whenever the value in the specified input column changes, the Map-Reduce task is called automatically by Triggy.

The execution of the Map-Reduce tasks is tightly coupled to the data distribution. This makes the system very interesting for our use case – as it scales automatically with Cassandra, the underlying key-value store.

### 3.2.2 Comparison with other Systems

For this kind of use case – and requirements in terms of data size and real-time processing - it’s natural to study how it can be implemented in a stream

\(^5\)This is a simplification, the reduce function can also return null or a list of values. However, this is not important for the distinction to the modified reduce function, as it can do this too.

\(^6\)A column family is the main data collection in Cassandra. Column families are described in detail later on, for now it’s sufficient to look at it as the equivalent to a table in a relational database.
processing engine. On account of this, we look into four state-of-the-art stream processing engines. These are: Esper/EsperHA [43], IBM InfoSphere Streams [42], Borealis [2, 37] and Yahoo! S4 [31].

Our analysis will answer two questions. (1) Can we express the proposed matching model? (2) Does the system meet the requirements stated in section 2?

Therefore, we will define a set of requirements, which are necessary to implement our use case in a stream processing engine. We will then provide an implementation in Esper and refine our requirements based on this implementation. What follows is a detailed analysis for each of the stream processing engines, which will show that none of these engines can fulfill all requirements. Hence we conclude that our use case cannot be implemented with a state-of-the-art stream processing engine.
4 Related Work

4.1 Product Recommendations

Recommendations based on social networks and user behavior got great attention in research over the last years. One of the topics that have been heavily covered is behavioral targeting. This is a technique to increase the relevance of ads by monitoring a user’s browsing and search history. Yan et al. [49] provide an empirical study about the effectiveness of behavioral targeting on the click-through rate of advertisements. Chen et al. [12] focused on the large scale aspect for behavioral targeting with big data. They provide an implementation based on the Apache Hadoop Map-Reduce framework [22].

Another popular topic is collaborative filtering. Collaborative filtering systems look at users which share the same preferences as the target user. The system recommends new products to them, based on similarities in preferences and interests. The probably most popular application of collaborative filtering is Amazon’s recommendation mechanism [29]. Also, other popular systems such as GroupLens [39] or the Google News Reader [13] use collaborative filtering. Herlocker et al. [23] give a good explanation about how and why one should implement collaborative filtering systems. Breese et al. [9] provide an empirical analysis for different algorithms used in collaborative filtering.

4.2 Twitter Analysis

Twitter is probably the most analyzed social-network. The number and variety of publications using tweets to detect all kinds of patterns and information is enormous. Jansen et al. [25] investigate Twitter as a form of word-of-mouth. Thereby, they analyze the effects of tweets containing comments and opinions about brands and products. Ramage et al. [38] provide an approach to characterize tweets by topics, based on Latent Dirichlet Allocation [8]. Ritter et al. [40] provide a method to model dialogues in twitter based on a Hidden-Markov-Model [30].

4.3 Time- and context-aware Recommendations

From the aspect of recommendations that take the users temporal interests and current context into account, there is some work for collaborative filtering. Gong et al. [20] improve collaborative filtering with time depended weighting and Blanco-Fernández et al. [7] use item-dependent time functions.

Densinger et al. [16] propose a user-context aware recommendation system, operating directly in retail stores. They explore how retailers can develop programs which are used by the customers during their shopping. Lawrence et al. [28] describe a system that recommends new products to supermarket customers, based on their previous purchasing behavior. Both systems are supposed to be used directly in a store, with special hardware and systems. This is different to our work, but we share the idea of providing recommendations based on the immediate-context of a customer.
5 Contributions

The main contributions of this thesis are as follows:

- Definition of a model for real-time product recommendations based on social media data.

- Analysis and implementation of different automated keyword extraction strategies for Amazon products, based on the text content of the products’ web pages. Thereby, suitable text-resources from the products’ web pages as well as the best algorithm to extract keywords are identified.

- Analysis and definition of the system level requirements to implement an application for scalable real-time processing of social media data.

- Implementation of the stated recommendation model in Triggy and different performance tests for this implementation.

- A comparison of the Triggy implementation with four state-of-the-art stream processing engines. That includes an implementation in the stream processing engine Esper [43].

Based on the provided implementation, different requirements, which are necessary for a stream processing engine to implement the defined model, are identified. These requirements are analyzed and it’s showed that none of the examined stream processing engines can fulfill all requirements.

- Implementation of a demo application to show the proposed approach.
6 Solution

This section describes in detail all solutions and implementations that have been made during this thesis. First, let us recall the problems that have to be solved: (1) We need to automatically produce weighted keywords for products from Amazon [4]. (2) We need to retrieve tweets from Twitter [45]. (3) We need to match tweets against the keywords using Triggy. (4) We need a demo-application to show our approach.

6.1 Overview of the System Architecture

Figure 4 shows the system architecture, it consists of three tiers:

- **Retrieval Tier**
  
  In the Retrieval Tier, all the data retrieval and data processing is done. It consists of two modules. The Tweets Retrieval Module implements retrieval and preprocessing of the tweets. The Keyword Extraction Module is responsible for product data retrieval from Amazon and keyword extraction. As one can see from figure 4, the data exchange between the Retrieval Tier and the Processing Tier is not in real-time (It’s done over files). The reasons for this are as follows:

  - **Tweets Retrieval Module** This module is used for testing and simulation. Therefore, it’s no option to monitor tweets in real-time because the results are (1) unpredictable and (2) not repeatable.
  
  - **Keyword Extraction Module** We extract the keywords for our products, before we process the tweets. Updating the keywords or adding new products in a running system, is out of scope for this thesis.

  The demo application however, will add a nearly real-time component. This component will download a number of tweets, for a desired user, and then simulate the publication of them.

- **Processing Tier**

  This is the implementation of the matching algorithm, using Triggy’s incremental Map-Reduce tasks on top of Cassandra.

- **Presentation Tier**

  The Presentation Tier displays the tweets and recommendations. It is built with the Jetty webserver [47].
6.2 Implementation

6.2.1 Retrieval Tier

**Tweets Retrieval Module** The Tweets Retrieval Module can (1) collect a set of valuable users and (2) generate a snapshot of their tweets. To retrieve data from Twitter, it uses the twitter4j API [46].
Twitter requires all APIs to use the OAuth\(^7\) authentication. The Tweets Retrieval Module provides a function to retrieve the required credentials for this authentication.

To collect a set of valuable users, one has to provide a user to start from and the desired number of users one wants to collect. We say a twitter user is valuable if he has between 200 and 10'000 followers and follows between 200 and 10'000 other people himself. This requirements ensure that the selected users are most likely “normal” people, with a fair amount of twitter usage. Moreover, we can explicitly exclude celebrities and bots. The modul takes all friends of the provided starting-user – which meet the valuable user requirements – into the result set. This procedure is applied recursively to the new found users, until the result set has the desired size. This set is then written into an XML file, for further processing.

To generate a snapshot of tweets, the module takes the valuable users from the XML file and downloads tweets for every user. The tweets are downloaded in inverse chronological order – most recent tweet first – up to maximal 3’200 tweets per user, as this is the Twitter API’s maximum tweet limit for a user. These tweets are then saved into one separate XML file per user. After all tweets have been downloaded, the tweet XML files are transformed into files which contain tweets ordered per day. These files can then be used for a simulation.

The Twitter APIs have a very restrictive rate limit. Applications are not allowed to do more than 350 requests per hour and roughly not more than 2 requests per second. For this reason, the Twitter web service sends timeout responses, whenever it feels that the request rate is too high or the hourly rate limit is achieved. The presented implementation accounts for both cases and either lowers the request rate or waits until the next hour begins.

However, that is not satisfying when one wants to download millions of tweets. The recommended approach for such a big data set is: first download the valuable users, then split them into several parts and finally download the tweets from different machines and with different twitter accounts. At the end, apply the tweet files inverter to all files, to generate tweet files per day.

\(\text{Keyword Extraction Module}\) The Keyword Extraction Module has two parts. The first part is the \textit{Amazon Crawler} – to download products and product information, the second part is the \textit{Keyword Extractor} – to extract keywords from the downloaded information, with different keyword-extraction strategies.

The \textit{Amazon Crawler} can download products for a certain department from Amazon. Therefore one has to provide three things. (1) The department node id. Every department on the Amazon website has an id, which is a part of the URL. This id is needed for the request to the Amazon Product Advertising web service. (2) The desired maximal category level. Amazon categories are highly nested. In account of this nesting, the \textit{Amazon Crawler} can parse this structure down to a desired level. (3) The number of products to retrieve, for every subcategory found at the previously stated level. This download is ordered by product popularity.

\(^7\text{OAUTH is an open protocol for secure API authorization. For more information see: http://oauth.net}\)
It’s useful to know that the nesting of Amazon categories is very wide. For this reason, the retrieval of subcategories for the first three levels, of the nesting structure, already leads to more than 1’500 results, for most departments. Hence, it’s better to keep the maximal category level small for most applications.

The Amazon Crawler first downloads the desired subcategories, using the Amazon Product Advertising API. Then it opens the web page for every downloaded subcategory, which displays the most popular products. Amazon only displays 24 products per page. The Crawler parses the page for product ids, using the open source HTML parser HtmlCleaner [24]. It then follows the Next-link on the page, until enough ids have been collected. With these ids, it can download the product title, the URL for a thumbnail and the URL of the product’s detail-page from the Product Advertising web service. At the end, this data is stored into an XML file.

From the products XML file, the crawler downloads the detail page of each product and parses the page for reviews, product detail descriptions, technical descriptions, user generated tags and comments. This data is then added to the XML file. Since the data size is large (several tens of thousands of products) for these tasks, they are all done with SAX style XML processing.

Unlike Twitter, Amazon has no hourly rate limit. But the web service also sends timeout responses when it feels that the request rate is too high (Amazon states that 1-2 requests per second should be ok). The crawler accounts for that and waits a few seconds when it detects such a response.

The Keyword Extractor extracts keywords, using the product files, produced by the Amazon crawler. One can choose between five keyword-extraction strategies.

- **AlchemyAPI [3]**
  This strategy takes all text content that is available for each product and extracts keywords using AlchemyAPI. AlchemyAPI is a collection of different commercial natural language processing services. One of them is a keyword extraction service that can extract topic keywords from a text document. It has a trial license, which allows 100’000 requests per day. This strategy uses the keyword extraction web service over the provided Java SDK.

  For every product, contained in the product XML file, the text data (reviews, product descriptions...) is sent to the AlchemyAPI web service. The web service responds with a list of keywords together with a relevance score. This relevance score is used as weight.

- **Yahoo Term Extraction Web Service [17]**
  This strategy does roughly the same as the previous one. It just uses the Yahoo! Term Extraction Web Service instead of AlchemyAPI. However, the Yahoo! Web service is more restricted, it has a rate limit of 5’000 requests per IP and day. That means, one has to do that from different computers or wait very long. It was not very limiting for our tests though, but for a productive system it would be. Besides, the license would not allow using it in a commercial system.

---

8With SAX style processing, one can parse an XML file sequentially. This is necessary for big files, since the whole file could exceed the system’s main memory.
The web service does not give back weights or relevance scores. However, the resulting keywords are ordered by relevance and so the strategy just applies a linearly decreasing weighting scheme.

- **Tf-idf Keywords**
  This strategy creates keywords using the well-established tf-idf weighting scheme. This weighting scheme defines the weight for a term $t_i$ in a document $d_j$ as follows:

$$ weight_{i,j} = \frac{n_{i,j}}{|d_j|} \times \log \left( \frac{|D|}{|\{d : t_i \in d\}|} \right) $$

(1)

Where $n_{i,j}$ is the number of occurrences of term $t_i$ in document $d_j$, $|d_j|$ the size of document $d_j$, $|D|$ the number of all documents in the collection and $|\{d : t_i \in d\}|$ the number of all documents in which term $i$ occurs. In our case, a document means all text data that is available for a product.

The strategy only takes nouns and names as candidates for keywords. To do this, it takes the tokenizer and part-of-speech tagger of the open source natural language processing library OpenNLP [34].

While all other strategies can be applied to single departments, or even to a single category, it is crucial for this strategy to have products from a variety of categories and departments. Otherwise the document frequency can be high for terms which are rather descriptive.

- **Tags**
  This strategy only considers the tags, which users can add to Amazon products. Amazon not only has this tags, it is also indicated how many people added the same tag. So this strategy simply takes the 10 most popular tags (of each product) and weights them according to the number of occurrences. For a tag $t_i$, that is:

$$ weight_i = \frac{|t_i|}{\max_i(|t_i|)} $$

(2)

Where $|t_i|$ is the number of occurrences of tag $t_i$ and $\max_i(|t|)$ the number of occurrences of the most occurring tag.

Because not all products have enough tags, for the strategy to deliver a fair amount of keywords, all products with less than 10 tags are neglected. Also if the most added tag occurs less than 10 times, the product is classified as not well enough tagged and neglected too.

- **Tags with Synonyms**
  This strategy first produces keywords from tags in the same way as the previous one. It then adds additional keywords by crawling http://thesaurus.com for each tag and adding the found synonyms. The weight of a synonym is set to a third of the weight of its originating tag.

### 6.2.2 Processing Tier

The **Processing Tier** implements the matching algorithm. It uses Triggy, an incremental Map-Reduce programming model on top of Cassandra. To describe
the matching model implementation, we first need to say a few words about Triggy and Cassandra. After this introduction, we will describe all components of the matching-algorithm, followed by an example.

**Cassandra**  Apache Cassandra is a highly scalable, distributed open-source NoSQL database. One can look at Cassandra as a mixture between Google’s BigTable [11] and Amazon’s Dynamo [15].

From Dynamo, Cassandra borrows the symmetric P2P\(^9\) architecture and distributed hash table for data storage, which provides a really good linear scalability. Also inspired by Dynamo, Cassandra picks availability and partition tolerance as primary guarantees to fulfill from the CAP-Theorem [10]. However - again like Dynamo - it has user tunable eventual consistency. That means that the user can choose to require more consistency in exchange to availability or vice versa, whenever it makes sense.

From BigTable, Cassandra has its data model. The data model has five building blocks: (1) **Keyspaces**; (2) **Column Families**; (3) **Rows**; (4) **Columns** and (5) **Super Columns**.

- **Keyspaces**  A **keyspace** is the outer most element of the data model – one can think of a **keyspace** as the equivalent to a tablespace of a relational database. A **keyspace** contains different **column families**.

- **Column Families**  A **column family** contains a potentially infinite number of **rows**. To make the relation to relational databases again, this would be a table. Listing 1 gives an example of a **column family** using JSON-notation (that is a common way to describe Cassandra data).

```json
ColumnFamilyName : {
    rowId : { // Row data } 
}
```

Listing 1: Description of a Cassandra column family

- **Rows**  **Rows** represent a record of data (for example a user). A **row** consists of an identifier and a number of **columns**. It’s important to see that there is no schema here, **rows** - even in the same **column family** - can have any number and kind of **columns**.

- **Columns**  **Columns** are the basic piece of data. They consist of a **name**, **value** and **timestamp**. Listing 2 shows an example of a **column family** with two **rows** and some **columns**.

```json
Users : {
    rowId1 : {
        username : {name: username, value: michael, 
                    timestamp:120223},
        adress : {name: adress, value: hometown, timestamp 
                  :178423}
    },
    rowId2 : {
```

\(^9\)Peer-to-peer
username: {name:username, value:mhaspra, timestamp:12043}
}

Listing 2: Description of a cassandra column family with two rows and some columns

The standard datatype of a column is string.

- **Super Columns** A super column is also a column, but instead of a string value it contains a map of columns.

That’s it. The Cassandra data model is very lenient, which makes it extremely flexible.

**Triggy** As already explained, Triggy extends Cassandra with incremental Map-Reduce tasks for push-style data processing. These Map-Reduce tasks are implemented in terms of triggers on rows. Therefore, the programmer has to specify on what column, of which column family, the task has to be executed (if the column is omitted, the task is executed for the whole row), and has to provide an implementation.

Listing 3 shows an example of the definition of a Map-Reduce task that is executed for the column `text`, of the column family `Tweets`. The results are written into the column `count`, of the column family `WordCounts`.

```java
name: TweetWordCounter
  keyspace: TweetProcessing
  input_cf: Tweets
  input_column_name: text
  output_cf: WordCounts
  output_column_name: count
  implementation: ch.ethz.systems.example.TweetWordCounter
```

Listing 3: Specification of a Map-Reduce task

This task is executed whenever something is inserted in the specified column, or a value changes.

Listing 4 shows the skeleton of such a Map-Reduce task, the programmer has to implement the two methods `map` and `reduce`.

```java
public class TweetWordCounter implements IMapReducer {
  @Override
  public void map(String key, Object value, OutputWriter writer) {
  }

  @Override
  public void reduce(String key, String value, Object aggregatedValue, boolean added, OutputWriter writer) {
  }
```

This class implements the IMapReducer interface provided by Triggy.
Listing 4: Skeleton of an incremental Map-Reduce task

Data Model Figure 5 gives a detailed description of the implementation of the matching algorithm. The algorithm is an interplay of column families and incremental Map-Reduce tasks. The column families are:

- **Tweets**
  This column family contains all tweets. The retrieval tier writes tweet data into it. The rows have the tweet id as row id and hold the tweet text. The author id is added as first word of the tweet text. This is a performance trick to prevent the following Map-Reduce task from querying Cassandra, which is - as we will see in the experiments section - an expensive operation.

- **UserTweetProductWordCounts**
  Whenever a tweet contains a keyword, this information will first be put into the `UserTweetProductWordCounts` column family. The key is composed out of the tweet id, the author id, the product id and the keyword of the found matching. The rows have only one column `count`, which contains the number of occurrences.

- **UserProductWordCounts**
  This column family contains the number of occurrences of a keyword (for a certain product) over all tweets for a user. So the row key is now only the author id, the product id and the keyword.

- **UserProductScores**
  `UserProductScores` contains the actual user-product matchings, together with a score. As we will see later, this score is a string containing all keywords that have led to this matching and their portion of the overall score.

- **Traces**
  In `Traces`, we have data for justification of recommendations. That means whenever a tweet contains a keyword – and therefore causes one or several matchings – this is saved there. A row has the user and the product of the matching as id. The data in the row contains all ids of the tweets that play part in this matching, together with the matched keyword.

- "**Users**"
  There is also a `Users` column family, which contains data about the twitter users. This column family is not important for the matching algorithm and therefore not displayed in the picture. It can be used by the presentation tier to display information about the user and is filled in by the retrieval tier.

- "**Products**"
  Additionally, we have a column family `Products` (it is also not important for the matching algorithm). It contains detail data about the products that can be used by the presentation tier when it shows a recommendation.
Map, which is also showed in the picture and looks like a column family is a just hash-map that contains the keywords and all product ids, this keyword is relevant for. This map is loaded on system startup.

Tweets:

```
Tweets: {
  tweetid: { "authorId": text }
}
```

Map:

```
Map: {
  keyword: { productId(weight), productId(weight), ... }
}
```

TweetWordCounter
Find keywords in text

UserTweetProductWordCounts:

```
UserTweetProductWordCounts: {
  authorId: tweetId: productId: keyword: { "count" }
}
```

UserWordCounter
Aggregate over tweetId

UserProductWordCounts:

```
UserProductWordCounts: {
  authorId: productId: keyword: { "count" }
}
```

UserProductScorer
Calculate the keyword score

UserProductScores:

```
UserProductScores: {
  authorId: productId: { keyword: score: keyword: score: ... }
}
```

Recommender
Check for threshold

Tracer
Write matching traces for provenance

Traces:

```
Traces: {
  authorId: productId: { "tweetId": word: ... }
}
```

Figure 5: Implementation of the matching algorithm with Map-Reduce tasks

**Map-Reduce Tasks** The following list describes the Map-Reduce tasks of the matching algorithm, displayed in figure 5.
• **TweetWordCounter**
This Map-Reduce task is called whenever a new tweet is inserted into the column family Tweets. The map method splits the tweet into words and matches every word against the keywords. For every found matching the user id, tweet id, product id, keyword and the number of occurrences are written onto a single row to the next column family.

• **UserWordCounter**
This task aggregates matchings with the same product id, user id and keyword. Thereby, the tweet id is dropped and the counts are summed up.

• **UserProductScorer**
The UserProductScorer turns the counts into a score (based on the weight of the keywords) and aggregates matchings with the same user id and product id, dropping the keyword. To understand this computation, note figure 6. For every keyword that caused a matching, between a user and a product, we have an entry in UserProductWordCounts. This entry contains the user, product and keyword as key and a count as value, to indicate how often we have seen this keyword for this user.

Figure 6: Description of the score-calculation for a product-user matching

To now compute a score for the user-product matching, the UserProductScorer first computes an intermediate score for every participating keyword (these intermediate scores are not saved in a column family) and then sums up these intermediate scores to an overall score. It’s important to see that this whole computation must be incremental! That means if the count of a keyword increases or a new keyword is found, we don’t want to recompute the whole score again. We only want to recompute or increase the corresponding intermediate score and then increase the sum.

The function for the intermediate scores is given in 3. It computes the intermediate score based on the weight of the keyword and the number of occurrences.

\[
\text{score} = \sum_{i=0}^{\text{count}} \frac{\text{weight}_i \times 2}{i \times 2}
\]  

(3)

However, there is an optimization of Triggy, which makes the implementation of this computation a bit harder than it may seems. The point is that Triggy can buffer intermediate values between several calls to the same reduce function\(^{10}\). For our computation, that means that the counts

\(^{10}\)This may not be the case for future versions of Triggy.
for a keyword can be increased in unpredictable steps. Together with the
scoring function in 3, this can lead to the following problem:

Assume, we have a keyword $a$ for a user-product matching, which has the
count 1 and a weight of 1. So the intermediate score for this keyword is 1
and therefore the score for the user-product matching is 1 too.

If now the count of keyword $a$ is increased from 1 to 2 and then to 3, the
score of the user-product matching has to be adapted. That means, the
UserProductScorer has to first add 0.5 and then 0.25 to it. If we could rely
on the fact that it sees both values (2 and 3), we had no problem – because
we can always tell what the last value was. But since the optimization
can make the count to rise from 1 to 3, we don’t know what the last value
was when we see a count of 3. If it was 1, we have to add $0.5 + 0.25 = 0.75$
and if it was 2 we have to add 0.25.

The solution to this, is to always re-compute the whole score for this par-
ticular keyword. But now, we cannot simply add this new intermediate
score to the matching score. We have to subtract the amount we added
earlier for this keyword. Since the score is only a number and keyword
$a$ might not be the only keyword we have found for this matching, we
cannot tell how much this is. So the final step is to encode the score in
a way so that one can identify the amount that was added for each key-
word. We do this by encoding the score for a user-product matching as
keyword1(score):keyword2(score). . . This leads to the following implemen-
tation of the UserProductScorer:

The map function calculates the intermediate score based on the new count
value with function 3 and writes a string of the form “keyword(score)”.
The reduce function replaces this string in the user-product score, of the
column family UserProductScores.

- **Tracer**

The Tracer is responsible to keep track of the reference data. Whenever
a keyword occurs in a tweet - and therefore a new (or several) entry in
UserTweetProductWordCounts appears, the Tracer starts, or extends, a
trace for the corresponding user-product matching. It’s important to see
that this isn’t necessary for later reference-data retrieval. All data is
already stored in the system. It makes it a lot easier though.

- **Recommender**

The recommender is called whenever the score for a user-product matching
changes. It first gets the threshold value for the product and compares
it with the new score. If the score is high enough, it sends out a recom-
mandation to the presentation tier (over a socket connection), deletes the
counter of this matching (from UserProductWordCount) and copies the
trace of this matching (for further justification of the recommendation).
Since a product can be recommended to the same user more than once,
the matching notification and the copied trace both contain a timestamp
of the time of the recommendation.

With this way of resetting the model it could possibly happen that too
many data is deleted. This is the case when a tweet results the score to rise
above the threshold and therefore starts the score resetting mechanism. If a following tweet - that affects the same score - is fast enough to be processed before the resetting happens, its effect is also deleted. This is not crucial for our application and one could solve that by just keeping track of the tweet id for every operation.

**Algorithm**  
The matching algorithm now works as follows:

- A tweet is stored into the *Tweets* column family.
- This tweet is processed by the *TweetWordCounter*. It saves every found keyword matching into *UserTweetProductWordCounts*.
- For every new matching keyword, the *UserWordCounter* aggregates the counts for the same user and product ids and the *Tracer* records the matching in a separate column family.
- Whenever the *UserWordCounter* changes a count for a user-product-keyword matching, the *UserProductScorer* updates the score for the according user-product matching.
- At the end, the *Recommender* compares the new score against the threshold for the product that plays part in the matching. If it is high enough, it sends out a recommendation and resets the counts.

**Example**  
To finish the description of the processing tier, we show the following example. Assume a user *Michael* publishes three tweets “Did some cool pictures today http://ti.com/ksdj7sdkj, I really like taking pictures…”, “Im at a photography exhibition.” and “Check out the pictures of my bike ride. http://ti.com/jkdj5sj”. Additionally, assume we have a product *Canon 3x* in our database for which we have the keywords *pictures*, with a weight of 1.0, and *photography*, with a weight of 0.8. The threshold for a *Canon 3x* is set to 2.5. Figure 7 shows the column families before any tweet has been processed.

<table>
<thead>
<tr>
<th>Column Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tweets: {}</td>
</tr>
<tr>
<td>UserTweetProductWordCounts: {}</td>
</tr>
<tr>
<td>UserProductWordCounts: {}</td>
</tr>
<tr>
<td>Traces: {}</td>
</tr>
<tr>
<td>UserProductScores: {}</td>
</tr>
</tbody>
</table>

Figure 7: Column families before any tweet was processed

Figure 8 shows the column families after the first tweet has been processed. The tweet contains the keyword *pictures* twice, this has led to an entry in *UserTweetProductWordCounts, UserProductWordCounts, UserProductWordScores* and *Traces*. Remember that the weight of the keyword *pictures* is 1 and that we
have decreasing score function for each keyword. This results in a score of $1.0 + 0.5 = 1.5$.

Tweets: 
  tweet1 : {Michael Did some cool pictures today http://ti.com/ksdj7sdkj, 
    I really like taking pictures...}

UserTweetProductWordCounts: 
  michael:tweet1:canon3x:pictures : [2]

UserProductWordCounts: 
  michael:canon3x:pictures : [2]

Traces: 
  michael:canon3x : {tweet1(pictures)}

UserProductScores: 
  michael:canon3x : {pictures(1.5)}

Figure 8: Column families after the first tweet has been processed

The second tweet contains the keyword *photography*. Figure 9 shows the state of the column families after it has been processed. Since this is a new keyword, for the matching between *Michael* and *Canon 3x*, there is a new entry in the column family *UserProductWordCounts*. The trace, as well as the score entry, are extended with an entry for the new keyword.
Tweets: {
  tweet1 : {Michael Did some cool pictures today http://ti.com/ksdj7sdkj, I really like taking pictures...}
  tweet2 : {Michael I’m at a photography exhibition.}
}

UserTweetProductWordCounts: {
  michael:tweet1:canon3x:pictures : {2}
  michael:tweet2:canon3x:photography : {1}
}

UserProductWordCounts: {
  michael:canon3x:pictures : {2}
  michael:canon3x:photography : {1}
}

Traces: {
  michael:canon3x : {tweet1(pictures):tweet2(photography)}
}

UserProductScores: {
  michael:canon3x : {pictures(1.5):photography(0.8)}
}

Figure 9: Column families after the second tweet has been processed

The third tweet now contains the keyword picture again. This causes the counter in UserProductWordCounts to rise from 2 to 3, what then again changes the score, for this keyword, from 1.5 to 1.75. This is showed in Figure 10.
With the third tweet, the score for the matching between Michael and Canon 3x is now 2.55, which is above the threshold of 2.5. Because of that, the model is resetted. That means, the trace is deleted and inserted again, with a timestamp added to the key. Also, the entries in UserProductWordCounts and UserProductScores are deleted. The resulting column families are showed in Figure 11.
Tweets: {
tweet1 : {Michael Did some cool pictures today http://ti.com/ksdj?sdkj, I really like taking pictures...}
tweet2 : {Michael I’m at a photography exhibition.}
tweet3 : {Michael Check out the pictures of my bike ride. http://ti.com/jkdj[5s]}
}

UserTweetProductWordCounts: {
michael:tweet1:canon3x:pictures : {2}
michael:tweet2:canon3x:photography : {1}
michael:tweet3:canon3x:pictures : {1}
}

UserProductWordCounts: {}

Traces: {
michael:canon3x:903222 : {tweet1(pictures):tweet2(photography):tweet3(pictures)}
}

UserProductScores: {
}

Figure 11: Column families after the recommendation has been made and the model has been reset.

The entries in UserTweetProductWordCounts don’t have to be deleted – to allow a second matching of the same product. This column family can easily be cleaned at a later time.

6.2.3 Presentation Tier

The Presentation Tier has four main components. (1) The Matching Listener, which can receive matchings from the processing tier. (2) The Presentation Servlet, which is responsible to handle requests from the users' web pages. (3) The Details Servlet, to handle requests for justification of recommendations. (4) The JavaScript Frontend, for the user GUI.

Matching Listener The Matching Listener waits for incoming recommendations on a socket connection. It is implemented as an observable object. If such a recommendation comes in, it sends the recommendation data (user, product and timestamp) to its observers.

Presentation Servlet The Presentation Servlet handles requests from the JavaScript Frontend. This frontend constantly polls for new recommendations. Therefore, the Presentation Servlet adds an observer to the Matching Listener, using jetty continuations. This causes the request to block until, either a timeout happens or the listener gets a recommendation for this observer.

Details Servlet The Details Servlet handles justification requests from the GUI. Therefore, it retrieves the trace from the database and analyzes which tweets were responsible for this recommendation. It then sends these tweets (together with the trace) back to the GUI.

35
**JavaScript Frontend**  The *JavaScript Frontend* implements the logic of the GUI. It does the following:

- Send a request for new recommendations to the *Presentation Servlet*. It does this using the AJAX methods of the JavaScript library jQuery [26].
- Receive recommendations and display them to the user.
- Send a justification request to the *Details Servlet* when a user wants to see why a product has been recommended.
- Receive the justification response and display the containing tweets.

### 6.3 Demo Application

To demonstrate the system, the presentation tier has been extended with a simulation module. One can enter a twitter name and a number of tweets to simulate. The simulation module then downloads these tweets and runs the simulation, which inserts a new tweet every second. The simulation speed as well as the threshold for recommendations can be changed.

Figure 12 shows the input form of the demo application.

![Screenshot of the input form for the simulation](image)

Figure 12: Screenshot of the input form for the simulation

Figure 13 shows a screenshot of the running system. On the left side, one sees the tweets of Barack Obama and on the right side the recommendations. As one can see there are a lot of books about health recommended. That is because he twittered heavily about the U.S health care system. Also some surveillance equipment is recommended because recent tweets contain the keyword *security*, due to the situation in Lybia.

![Screenshot of the running demo application](image)

Figure 13: Screenshot of the running demo application
The last screenshot shows the justification mechanism. One can click on the "why" button and the system then shows the tweets that have led to this recommendation.

Figure 14: Screenshot of a recommendation together with it's justification
7 Experiments

This section presents the experiments for the system and the recommendation-model. We are interested in answering three questions. How can we automatically extract keywords for a product, which represent possible user interests this product could be recommended for? How is the performance of our implementation? What are the differences between the implementation with Triggy and possible implementations in a state-of-the-art stream processing engine?

7.1 Keyword Extraction

For the keyword extraction, we analyze two aspects. First, we want to know from what source (Product Description, User Comments...) we can extract valuable keywords. Second, we are interested in the best keyword-extraction strategy (AlchemyApi, Yahoo! Term Extraction...).

For each combination of source and extraction strategy we analyze (1) the keywords themselves and (2) at the recommendations they lead to.

7.1.1 Test Setup

As product test set, we took a sample of the best selling products from the Amazon departments: Books, Sports & Outdoors and Electronics. This has led to a set of \(~2'000\) Books, \(~14'000\) Sports & Outdoor products and \(~16'000\) Electronics products. From each product webpage, we extracted the text content from the sections: Product Name, Product Description, Review, Technical Details, Product Features, User Generated Tags and Comments.

For the tweets test set, we first collected 5'000 valuable twitter users. We say a twitter user is valuable if he has between 200 and 10'000 followers and follows between 200 and 10'000 other people himself. We collected these 5'000 valuable users by taking a user we know and downloaded the rest with the Tweets Retrieval Module. For every valuable user, we then downloaded all recent tweets, up to a maximum of 3'200 tweets per user. This produced a test set of \(~6'000'000\) tweets, with tweets from the last 4 years.

To determine the quality of the keywords and the resulting recommendations, we produced keywords from different sources with the different strategies. We then took samples from the resulting keywords for evaluation.

Therefore, we state the following criteria for keywords:

- **Relevance** How relevant are the keywords for the product?
- **Weighting** Does the weighting conform to the relevance of the keyword?
- **Level of Abstraction** Do the keywords have an appropriate level of abstraction? A keyword has to be specific enough to represent a potential user interest. For example, the keyword *love* would be to verbose, it would generate a lot of non-relevant matchings. On the other hand, a keyword must not be too specific. For example, the ISBN number of a book would hardly ever generate a matching.

If the extracted keywords looked promising, we run the simulation with the tweet test set and analyzed the resulting recommendations.
7.1.2 Results

What source to take? The results for the different sources are:

- **Product Name** Obviously the product name is too short to provide enough data for keyword extraction. But together with other sources it can usually add one or two valuable keywords.

- **Product Description** The product description is usually a quite promising source for keyword extraction. It is long enough to provide enough data and has a good level of abstraction.

- **Reviews** Reviews tend to be very verbose, they usually lead to highly non-relevant keywords. Also reviews are only available for books, what makes them inappropriate for a wide usage.

- **Technical Details and Product Features** Technical details and product features are too specific, as they mainly contain the size, weight or other very-specific figures. They also often contain only bullet points and are not available for all product categories.

- **Comments** Comments are even worse than reviews in terms of generating non-relevant keywords. People write about all sorts of unrelated topics in the comment section.

- **Tags** We found tags to be very promising for keyword extraction. They have a very good level of abstraction and the relevance is even ensured by the customers, since it is indicated how many customers proposed the same tag.

Based on our observations, we decided to use the product name and the product description for keyword extraction with Yahoo! Term Extraction, AlchemyApi and tf-idf weighting.

What strategy to take? The results for the different strategies are:

- **AlchemyApi and Yahoo! Term Extraction** Both strategies lead to either very specific keywords or very unrelated ones. Also the weighting does not conform to the relevance. We were able to improve the results a bit by cleaning the input data (e.g. delete Amazon specific keywords) but the results were still not satisfying.

- **Tf-idf** With the tf-idf strategy, only the top 3-4 keywords were relevant for the product. These are too few keywords, to lead to a reasonable number of recommendations. Also, the keywords were rather specific and it was not possible to define a threshold to indicate from which position on a keyword is relevant.

- **Tags** Keywords from tags produce the best recommendation results. Relevance and an appropriate weighting are ensured by Amazon customers. The level of abstraction is rather high for tags but not significantly more than for other strategies. One shortcoming however, is that not all products have a sufficient number of tags.
- **Tags with Synonyms** Adding synonyms to tags does not lead to better recommendation results, for products with only a few tags. It can even harm the relevance of recommendations, for products with a lot of tags.

7.1.3 Conclusion

Many sources, from a typical Amazon product website, contain either very product specific details or highly unrelated information. The best sources are *Product Name*, *Product Description* and *User Generated Tags*.

Even though Yahoo! Term Extraction, AlchemyApi and a tf-idf strategy can find some good keywords, the resulting recommendations are not very good. This is because the quality of the keywords varies strongly.

Tags however, are very promising. For our application, we decided to discard products with less than 10 tags.

7.2 Performance Experiments

7.2.1 Test Setup

The performance tests were executed on a cluster where each machine was running Debian 2.6.26-25lenny1 (64bit) on two 2.4 Mhz Dual Core AMD Opteron 280 processors with 1MB L1 cache and 6GB main memory. Cassandra was started with the following JVM Options: -Xms1G, -Xmx2G, -Xss128k. That means, the JVM was started with 1 gigabyte initial heap memory (-Xms1G) and 2 gigabytes as maximum heap memory (-2Xmx2G). The stack size for each thread was 128 kilobytes (Xss128k).

The measurements were running for 5-10 minutes trying to insert as many tweets as possible. Every 30 seconds the actual throughput was recorded and an average was calculated at the end.

7.2.2 Querying vs. Pushing

If the score for a user-product matching changes, the system has to decide if the score is high enough for a recommendation. We came up with two solutions for this (1) we query Cassandra for the threshold every time and only send out a recommendation when the score is high enough or (2) we push out the new score every time and leave the decision to the presentation tier.

The intention of this experiment is to see if querying Cassandra, and therefore not pushing out the score, can improve the performance.

To simulate this, we took the tweet id, which is uniformly distributed in our simulation. With the following condition, we can simulate the matching of a tweet:

```java
if ((tweetId % 10) < threshold) { match }
```

Listing 5: If-condition to simulate a tweet match

Therefore we have implemented three Map-Reduce tasks:

- **Empty Map-Reduce Task** This task is executed on every insertion of a tweet but it does not perform any operation. It’s performance is used as a reference value.
• **Pushing Map-Reduce Task** This task pushes the tweet id over a socket connection to the presentation server, whenever a new tweet is inserted.

• **Querying Map-Reduce Trigger** This task queries Cassandra for the threshold and pushes the tweet id to the presentation server if the condition in Listing 5 is true.

The test was performed for 10 minutes, on 3 cluster nodes, for matching rates between 0% and 100%, in steps of 20%. Figure 15 shows the results of this test. We see that querying Cassandra for the threshold harms the performance even for a matching rate of 0%. So our conclusion is that pushing the parameter out is better than querying it instantly.

![Figure 15: Querying vs. Pushing test results](image)

### 7.2.3 Is it better to have one big Map-Reduce Task or several small ones?

We conducted this experiment to test if it is better to have a small number of Map-Reduce tasks, which then have to do more computation, or more tasks, which do only little computation. Therefore, we prepared three different implementations of the matching algorithm.

- **6 Map-Reduce Tasks**
  The first implementation has 6 Map-Reduce tasks, which all do very little computation.

- **3 Map-Reduce Tasks**
  The second implementation has only 3 Map-Reduce tasks. Because the tasks now do more computation, they also have to query Cassandra – for the extra data they need.
• Non Querying
Because we assumed that querying Cassandra is painful for the performance, we also provided a third implementation, which does not do any extra querying at all.

![Graph showing comparison between different numbers of Map-Reduce tasks](image)

**Figure 16: Comparison of different numbers of Map-Reduce tasks test results**

The test was performed for 10 minutes, on 3 cluster nodes, for matching rates between 0% and 30%. Figure 16 shows the results of this test. We see that it is better to have several small Map-Reduce tasks. Like in the previous experiment, the reason for this result is that extra querying highly affects the performance of a single Map-Reduce task. The fact that the non querying implementation shows the best performance further supports this claim.

### 7.2.4 Scalability
The last test was the scalability test. This test was performed for 5 minutes on 1, 2, 3, 4, 5 and 6 cluster nodes. For all settings with more than two cluster nodes, the replication factor was set to 2.

Figure 17 shows the results of this test. We see that the system scales very well. Also with only 5 nodes, we can already handle more than 1’000 tweets per second, which is about the workload twitter has on average.
7.3 Comparison with Stream Processing Engines

High-performance processing of massive data-volumes – this is the showpiece of every stream processing engine. These engines are optimized to handle hundreds of thousands of events per second, on an ordinary desktop computer. So, it’s natural to analyze how our use case could be implemented in such a stream processing engine and compare it to the Triggy implementation.

To achieve such a high performance, stream processing engines have to make several assumptions and restrictions. One – very rewarding restriction – is to let the system operate only in main memory – and thereby take disk-IO out of the performance calculation. Another one is to restrict the computational algebra to very simple operators – which then can be optimized in a very effective way. All these restrictions bring limitations in terms of expressiveness, scalability, fault-tolerance and a number of other properties.

The goal of this section is to analyze 4 state-of-the-art stream processing engines and to identify potential properties of our use case that make an implementation in a stream processing engine difficult or even impossible.

In the following sections, we will describe the chosen engines, followed by a possible implementation of our use case in one of them (Esper [43]). Based on this implementation, we will define a list of requirements that have to be fulfilled by a stream processing engine, to allow the implementation of our use case. Finally, we provide an analysis of the requirements for every system.

7.3.1 Selection of the Stream Processing Engines

The stream processing engines, we use for comparison, are:

- Esper/EesperHA [43]
  Esper is an open-source stream processing engine, available for Java and .NET. It provides a rich interface to implement user-defined functions and
complex data windows. EsperHA is a distributed, closed-source, commercial extension of Esper which adds scalability and resiliency.

- **IBM InfoSphere Streams** [42]
  IBM InfoSphere Streams is a commercial, distributed stream processing engine. It comes with its own programming language, an IDE and a number of supporting tools.

- **Borealis** [2, 37]
  Borealis was a research project to develop a distributed stream processing engine until summer 2008. Although it is no longer an active project, the engine is still a good state-of-the-art example.

- **Yahoo! S4** [31, 48]
  S4 is a platform that allows the development of distributed stream processing applications. It was released by Yahoo! in October 2010 under Apache 2.0 licence.

### 7.3.2 Implementation in Esper

Although Esper is not a distributed engine, we chose it to show a potential implementation of our use case for the following of reasons. (1) It is open-source. (2) It has a rich interface for user-defined functions and complex user-defined windows. That gives it a powerful model to implement applications. (3) We hope that the high-availability extension EsperHA fulfills our requirements in terms of scalability and fault-tolerance.

Figure 18 shows our implementation in Esper. It shows the different streams, windows and queries we created. We will also show some example events after the following description of the important steps:

- **S1 – Tweets Event Stream**
  The tweets come into the engine as events in the Tweets Event Stream. We have tweet-events of the form:

  \[
  \text{Tweet-event} = \{ \text{tweetId}, \text{username}, \text{text} \}
  \]

  Listing 6: Form of a tweet-event in the Esper implementation

- **Q1 – Matching Queries**
  For every combination of product and keyword, we have a query which consumes the tweet-events from the Tweets Event Stream and parses the text for the keyword.

  As an example, a query for the product *Canon 3x* and the keyword *pictures* looks like this:

  $$\text{As an example, a query} \> \text{for the product } \text{Canon 3x} \> \text{and the keyword } \text{pictures} \> \text{looks like this:}$$

---

11 A *stream* is a queue of events. One can put events into a stream and submit queries that retrieve events from it. A stream does not save events, when there is no query (left) for an event it is discarded.

12 *Windows* are queues of events. In contrast to streams, they can save events based on different conditions such as the time or counts. There exist also infinite windows which just keep all events.

13 *Queries* are the basic processing elements in Esper. With a query, one can retrieve events by its properties, create events and insert events into streams and windows.

14 An *event* is the basic data unit. It consists of a number of properties to store the data. Events can be created, queried for and putted into streams and windows.
insert into MatchingWindow(tweetId, username, product, keyword, count)
select tweetId, username, "canon3x", "pictures",
    countMatchings("pictures", tweet-text)
from TweetEventStream
where countMatchings("pictures", tweet-text) > 0
Listing 7: A query to find a matching and insert it into the next window

The matching-events, which are inserted into the next window, have the following form:

Matching-event = {tweetId, username, product, keyword, count}
Listing 8: Form of a matching-event in the Esper implementation

• W1 – Matchings Window
The Matchings Window is an infinite window. It keeps all matching-events until they are deleted by a query. Whenever a new matching-event comes into this window, a score query is applied to it.

• Q2 – Score Queries
For every combination of product and keyword, we have a query that consumes all matching-events for this combination and sums up the counts from the different tweets of the same user. Then it calculates a score, based on the weights of the keywords.

As an example, a query for the product Canon 3x and the keyword pictures looks like this:

insert into ScoresWindow(username, product, keyword, score)
select username, product, keyword, computeScore(sum(count), 1.0)
from MatchingWindow
where product = 'canon3x' and keyword = 'pictures'
group by username
Listing 9: A query to sum up and score matching counts per user, product and keyword

The score-events, which are inserted into the next window, have the following form:

Score-event = {username, product, keyword, score}
Listing 10: Form of a score-event in the Esper implementation

• W2 – Scores Window
The Scores Window is a count based window. It only keeps the newest score-event for every combination of username, product and keyword.

• Q3 – Sum up queries
The last query sums up the scores from different keywords, for every user and product. This can be done with only one query that looks as follows:
```
insert into RecommendationsStream(username, product, score)
select username, product, sum(score)
from ScoresWindow
group by username, product
```

Listing 11: A query to sum up the scores from different keywords for a user-product recommendation

The recommendation-events, which are inserted into the next stream, have the following form:

```
Recommendation-event = {username, product, score}
```

Listing 12: Form of a recommendation-event in the Esper implementation

- **S2 – Recommendations Stream**
The Recommendations Stream holds all recommendation-events. Since it is a stream, the events are only kept as long as there is a query to handle it.

- **Q4 – Recommendation and deletion queries**
For every product, we have a query, which compares the score in the recommendation with the threshold and deletes all corresponding events from the two windows if the score is high enough.

For example, a query to delete all events from the scores window – on a recommendation of a Canon 3x – would look like this:

```
on Recommendation-event(score >= canon3x.threshold) delete from ScoresWindow
where ScoresWindow.username = username and ScoresWindow.product = product
```

Listing 13: Query to delete events from a window on recommendation
For all combinations of products and keywords:
Insert a new event into the scores window with the score calculated based on the occurrences of the keyword.

Whenever a new event comes into the scores window:
Sum up the scores from all events containing the same user and product as the incoming event (Aggregate over keyword) and send a recommendation event.

If the score is above the threshold for this product, delete all events from the two previous windows and send out the recommendation.

Figure 18: Matching-Algorithm implementation in the Esper Stream Processing Engine
To illustrate the previous explanations, we describe a short example. Imagine a user *Michael* who publishes three tweets: (1) *Did some cool pictures today http://ti.com/ksdj7sdkj, I really like taking pictures...* (2) *I'm at a photography exhibition.* (3) *Check out the pictures of my bike ride. http://ti.com/jkdj5sj*

Furthermore, let's say we have a *Canon 3x* in our product database. This product has two keywords (*pictures* and *photography*) assigned, with the weights 1.0 and 0.8. Figures 19 - 21 show the events which occur in the different streams and windows. Also we define the score-threshold for a *prius* to be 2.5.

To begin, figure 19 shows the state of the engine after the first tweet was inserted. The two occurrences of the keyword *pictures* caused a matching-event to be created by a query in step Q1. Furthermore – because of this matching-event – a query at step Q2 took the event and calculated a score, based on the weight of the keyword. This query then created a score-event and inserted it into the scores window. In step Q3 the sum-query took this single event and created a new recommendation-event, which was inserted into the Recommendation Event Stream. The queries in step Q4 were not executed – since the score of the recommendation was below the threshold of the product.

\[
\{\text{tweet1, michael, „Did some cool pictures today http://ti.com/ksdj7sdkj, I really like taking pictures...“}\}
\]

**Figure 19:** State of the Esper streams and windows after the insertion of the first tweet

Figure 20 shows the state of the engine after the second tweet was inserted. The Matchings Window now contains two events. One for the keyword *photography*, which was found in the tweet-text and the old event, since this is an infinite window. Also the Scores Window has now two events. The old events in the two streams disappeared, since streams do not keep events.
Figure 20: State of the Esper streams and windows after the insertion of the second tweet

Figure 21 shows the state of the engine after the third tweet was inserted. We see an additional matching-event in the Matching Window.

In the Scores Window, the score-event for the keyword pictures was replaced by a new one.
7.3.3 Observations

Based on the implementation in Esper, we can identify some specialties of our use case. One is that we have a complex type of windows. We cannot foresee for how long we need to monitor the event or how many events we will have. So, we cannot use traditional count- or time-based windows. In Esper, the solution is easy – given the fact that Esper allows queries which delete events from windows. A suitable stream processing engine must either provide similar functions or some sort of complex windows, which allow to specify this. We also see that these windows will grow very big – so they are not suited to be held in main memory – a potential engine has to account for that and save the overflowing data to persistent storage.

Besides the complex windows, we make use of user-defined functions, to match the keywords and to calculate the score. So we need either an interface to implement our own functions or these functions have to be provided by the engine.

Lastly, we see that our implementation will have tens of millions of queries (two for every combination of product and keyword). We see this huge amount of queries as potential problem for stream processing engines.
Based on these observations, we can now define the requirements for a stream processing engine to allow the implementation of our use case:

- **Scalability**
  We need a distributed streaming engine to ensure horizontal scalability.

- **Fault-Tolerance**
  We cannot accept the loss of all data, so the engine must be fault-tolerant. However, since Triggy also doesn’t ensure full fault-tolerance (intermediate values between map and reduce could be lost), we weaken this requirement and say. We accept to loose the effect of a tweet once in a while.

- **Expressiveness**
  We need to be able to express our complex windows and keyword matching.

- **Referenz-Data Retrieval/Provenance**
  There must be a possibility to retrieve reference-data after a recommendation.

- **Handle a big Number of Queries**
  We have a lot of keywords and products. As we already stated, this causes a big number of queries. The engine must be able to handle that.

- **Resiliency of a single Cluster Node**
  We keep the data in the windows for a long time (for days or even months). The engine must be able to swap some of the data to disk.

### 7.3.4 Analysis

The following table provides an overview of our analysis.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Esper</th>
<th>EsperHA</th>
<th>IBM InfoSphere Streams</th>
<th>Borealis</th>
<th>Yahoo S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Fault-Tolerance</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Referenz-Data Retrieval</td>
<td>&quot;yes&quot;</td>
<td>&quot;yes&quot;</td>
<td>&quot;yes&quot;</td>
<td>&quot;yes&quot;</td>
<td>&quot;yes&quot;</td>
</tr>
<tr>
<td>Handle a big Number of Queries</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Resiliency of a single Cluster Node</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**Scalability**  Since we have chosen the systems according to this requirement, it is fulfilled by all our systems, but Esper.
Fault-Tolerance

- Esper – No, Esper is neither distributed nor has a database.
- EsperHA – EsperHA is fault-tolerant. However, it only saves the state of built-in functions and windows. While this is sufficient for our implementation, it might be a problem for more sophisticated ones.
- IBM InfoSphereStreams – Yes, InfoSphereStreams can do checkpointing.
- Borealis – Yes, Borealis does replication of nodes.
- Yahoo S4 – No. However, checkpointing is planned to be implemented.

Expressiveness

- Esper – Yes. As we can see from the fact that we provided a running example.
- EsperHA – No. According to a developer of EsperHA, EsperHA could help with the persistence of the windows. The window distribution—more precisely, the transfer of data between nodes—would be left to the developer. We explicitly stated that this is not a suitable solution.
- IBM InfoSphereStreams – Yes. According to a developer of InfoSphereStreams, one could express our application within this engine.
- Borealis – No. According to a former developer of Borealis, the underlying algebra does not allow the deletion of events from a window or the definition of a window that would fit our needs.
- Yahoo S4 – Yes. We did not implement it, but from investigating the programming model of S4 we believe that one could do it.

Reference-Data  It holds for all systems, that one can theoretically do it, by adding functions to the application, which save the data to a database during intermediate steps. However, that means that one has to either save everything or foresee the usage one wants to make out of this data.

Handle a big Number of Queries  Although we believe that this is difficult to achieve, we think that a modern engine can do this with some help by the programmer. Also, we have to account for the fact that there may be a better implementation, which does not use that many queries.

Resiliency of a Single Cluster Node

- Esper – No. As stated on Espers webpage, it can handle as many data as the system has memory.
- EsperHA – Yes. This is exactly what EsperHA adds to Esper.
- IBM InfoSphereStreams – No. As a developer of InfoSphereStreams stated: “This is not supported automatically by the system. Right now, the developer has to anticipate the memory requirements and plan accordingly.”
• Borealis – No. We did not found any information that Borealis can do that in the documentation.

• Yahoo S4 – No. We did not found any information that S4 can do that in the documentation.

7.3.5 Conclusion

We see that none of the analyzed stream processing engines can fulfill all the requirements. Therefore, we conclude that the implementation of the use case we described for online advertising is impossible to implement, using state-of-the-art stream processing engines. As most critical properties of such a use case, we can point out:

• We need a window that defines the endurance of events based on other properties than just counts or the time. We showed a solution with an infinite windows and queries that can delete events. Another imaginable solution would be a window that defines the endurance of its events with some user definable predicates.

• Our windows will grow very large, though having data only in main memory is not a suitable solution.
8 Conclusion

8.1 Summary

During this thesis, we have developed a system for scalable real-time product recommendation, based on user’s activity in a social network. Therefore, a general use case for this problem was defined first.

The examined social network was Twitter. It has been analyzed how the data from twitter can be retrieved and how the tweets can be processed, to learn a user’s interests. The product database was built with Amazon products. Thereby, a system to automatically download products and complementary product information directly from Amazon.com as well as with the Amazon Product Advertising web service has been developed.

For the downloaded products, different keyword-extraction strategies were tested. It has been showed that user generated tags are a valuable resource to extract keywords that identify topics that are related to this product.

For the recommendations, a flexible, incremental and scalable model has been defined. This model was implemented on top of Triggy, which extends the key-value store Cassandra with incremental Map-Reduce tasks for push-style data processing. Three performance experiments were conducted for this implementation. The two main results of these experiments are that one should avoid intensive querying to Cassandra whenever possible and that the system scales well on a cluster.

In addition to the performance experiments, the model was implemented in the state-of-the-art stream processing engine Esper. Then, the stream implementation was compared with the Map-Reduce implementation and a list of requirements which have to be fulfilled by a stream processing engine, to allow the implementation of the matching-model was defined. For state-of-the art stream processing engines were analyzed and it has been showed that for every engine at least one requirement is not fulfilled.

Lastly, a demo application to present the approach was implemented. This application can simulate the Twitter behavior of a user and has a sample product database to show the recommendations.

8.2 Open Issues

For each perspective (recommendation & systems), there stays one main open issue.

From the recommendations perspective, one can further improve on the extraction of interests from tweets. Tweets have a very limited language and are of short nature. If one wants to determine a user’s interests and current context, tweets are not enough for a very good accuracy. However, the flexible nature of the matching-model allows adding additional sources of information – such as locations – very easily.

From the systems perspective, resetting the counters could be done in a better way. As already stated, with the implemented mechanism it is possible that the effect of a tweet that follows right after a recommendation could be lost. However, this is not critical for the presented application and a solution for this problem has been described briefly.
Bibliography


56


[34] OpenNLP. http://incubator.apache.org/opennlp/.


