HAAGA-HELIA
University of Applied Sciences

Realtime Web Analytics

João Cardoso

Bachelor's Thesis
Business Information Technology
05.05.2011

HAAGA-HELIA
University of Applied Sciences

Abstract

05.05.2011

Business Information Technology

Author João Cardoso	**Group** BITE
Thesis title Realtime Web Analytics	**Number of pages and appendices** 27 + 3
Supervisor Tero Karvinen	

Tracking what is happening on a website in realtime is invaluable. The objective of this thesis was to start and launch the first version of Snowfinch, an open source realtime web analytics application.

The thesis report contains up-to-date fundamentals of web analytics; reasoning behind the most important and difficult technical decisions in the project; product development methodologies; and an overview of the resulting application.

Understanding visitors is the key to a site's success. Research shows that the amount of insights provided by clickstream data is small in comparison to the amount of data gathered. Even though there is much more to web analytics than clickstream data, the fact is that it is the only data that can be collected and analyzed exclusively by software, thus the need to optimize its use.

With the current use and importance of social media, being able to act quickly is a business differentiator. Visualizing data in realtime is crucial to a fast response. Choosing an appropriate data store plays an important role in the overall performance, responsiveness, and development time of the system.

The development work was sponsored by Kisko Labs Oy. Snowfinch will be used both internally and in client projects. During the development period, it has tracked over half a million page views and approximately two hundred thousand visitors.

Snowfinch may be used by anyone as it is, or it can act as a solid foundation on which to build customized functionality. Being free and deployable inside a private network offers a competitive advantage over commercial alternatives.

The fundamental technologies used are the Ruby programming language, the Ruby on Rails web framework, and the MongoDB document-oriented database. The project was released under the MIT license, a permissive and free software license approved by the OSI (Open Source Initiative).

Keywords
analytics, realtime, ruby, rails, mongodb

Table of contents

1. Introduction

Original ideas don't exist. Chances are that most of us are stuck in a business with too many competitors. The simplest way to differentiate yourself is to provide the best service and experience. In the web, that translates to truly understanding your users and listening to them.

This thesis work focuses on understanding. The main outcome is an open source realtime web analytics application that is capable of collecting and analyzing data, providing good insights on visitors and their trends.

Doing all of this realtime is more important than one might realize. Social networks accelerate the spread of information and products. The problem is that this virality doesn't usually last more than a few hours. How many times have online stores lost sales because they didn't act fast enough? If a product goes viral, offer a discount. Make sure that undecided visitors have no excuse not to spend their money. Have a blog post that is getting many visitors from a discussion on some other site? Join it, answer people, get involved!

Anyone who understands the importance of the ideas introduced above, cares for this application. It goes by the name of a bird, Snowfinch.

This document supports the developed application, explaining the basics of web analytics, their importance, what has been implemented, why it matters, and technical details on the implementation and its methods.

Work has been sponsored by Kisko Labs, a web development shop located in Helsinki, Finland. Some of the features have been developed as part of the sponsorship, as they are useful to them and their clients.

On the technical side, the application is built with the Ruby programming language, the Ruby on Rails web framework, and the MongoDB document-oriented database. The application stack is composed of three components:

– A JavaScript tracker that is embedded in the pages being tracked. Every time a page is loaded it makes a request to the collector, passing a unique visitor identifier and page information.

– A collector that receives data sent by the JavaScript tracker and stores it in a MongoDB database using an easy to query data model. It can run automatically on top of the web application, or in standalone mode for optimal performance and better scalability.

– A Ruby on Rails web application where Snowfinch users can manage sites and have access to all gathered information and insights.

Once and up and running, it is no more work to track a site, than it is with Google Analytics. One difference is that data is kept by the Snowfinch user, thus enabling full ownership and control. This is important for cases where company policy requires all data to be kept in-house, behind company firewalls.

Many web analytics products exist, so why create another one? Most current solutions are commercial, but, most importantly, this is a topic of my liking. There is still plenty of room for innovation in the field, and I believe I have some good ideas.

2. Web analytics

Snowfinch focuses on the collection, analysis and display of clickstream data. This is basic and easily available data, but there seems to be a disagreement regarding its value.

In an article by Phippen, Sheppard & Furnell (2004, 285), it is mentioned that the basic clickstream data, such as page views and visitors are inaccurate and misleading. The only argument that still holds is that some spiders and bots will generate page views. In my opinion, this is not a problem at all. There are freely available lists of User-Agent headers sent by such robots, thus filtering them is possible. The other argument was about the use of frames in sites, which is plain bad practice. While I disagree that this is good reasoning for the lack of value present in clickstream data, there are good opinions out there.

Clickstream data essentially answers the questions *what* and *when*, but not *why* or *how* (Weischedel & Huizingh 2006, 463). Kaushik (2010, 2–7) agrees with this view, and takes the idea further, stating that when it comes to clickstream data, we end up with a lot more data than insights, and insights are the gold.

I strongly believe that answering the question *why* should be ignored at one's peril. Some work is required to get this data. Two ways – and both should be used – of answering this question are experimentation and testing, and the voice of the customer. Getting the voice of the user is crucial for a successful site. One may know what users look at, but not if it is what they were looking for. By conducting surveys and getting constant feedback, the level of customer satisfaction will rise, and so will the performance of the site. (Waisberg & Kaushik 2009, 3–4.)

A/B testing plays an important role in understanding users and what works best, both for the users and for the business. This testing technique is characterized by displaying two different versions of one or more pages to the users, and measure which version performs better. Results for each version can be stored by a web analytics tool. (Nielsen 2005.)

Analytics tools can do the clickstream part on their own, and some may support the gathering of data resulting from experimentation and testing, but there are specific tools for that as well. The same applies to online surveys.

Using a web analytics applications must not affect the user experience of a site. Data collection has to be done in an asynchronous fashion. Experiments at Amazon have showed that slowing down pages by 100 milliseconds decreases sales by 1%. A similar experiment at Google has demonstrated that slowing down search results by 500 milliseconds reduces revenues by 20%. (Kohavi, Henne & Sommerfield 2007.)

This brings us to the conclusion that one of the strengths of clickstream data is that it can be gathered automatically by an analytics product, without any extra work. For this reason, the next logical step is optimizing the use of clickstream so that we can get as much value out of it as possible. Data analysis plays an important role here, especially focusing on the identification of loyal visitors (Sen, Dacin & Pattichis 2006, 91).

3. Tools

When the project started, I was already familiar with the tools that were used. The biggest reason behind the technological choices was that I have enough experience with them to be comfortable developing a project of this size in such a short period of time. Do not underestimate them, though. They are my favorite tools for a reason, and – even though my opinion is biased – the results certainly prove their worthiness.

3.1. Ruby

Ruby is a fairly different programming language. It originated from Japan and focuses not on performance, but on the programmer: happiness is Ruby's main goal.

While the language supports multiple paradigms, object-orientation is at its core. In Ruby, everything is an object. Many other object-oriented languages aren't as pure as Ruby, where every primitive type is an object than can even be extended by the developer. (Ruby community.)

Ruby allows developers to write programs that are usually shorter than in most other programming languages. Less code usually means less problems, and less code to maintain. This is, many times, due to an extremely powerful feature known as metaprogramming, allowing for code to be created at runtime. In other words, a developer is able to write code that writes code. Many great creations that come from the Ruby community, such as Ruby on Rails, would not be possible without metaprogramming.

3.2. Rack

Rack introduces Ruby to the web. It provides a common API between Ruby application servers and Ruby web frameworks. An application using any framework built on top of Rack, such as Ruby on Rails, can be deployed on any of the supported application servers. In addition, separate applications built on any supported framework can be put together and deployed as one. This allows for the different frameworks to be used, side by side, depending on the requirements. As it turns out, this specific feature makes Snowfinch easier to deploy and more performant than if only Ruby on Rails would have been used. (Neukirchen 2007; Rack contributors 2011.)

A part of Snowfinch is coded directly on top of Rack without using any web framework, for optimal performance. Rack does a great job at providing a simple API to handle requests and responses, requiring the developer to implement only one method.

3.3. Ruby on Rails

The web development scene was changed in 2004, when Ruby on Rails was launched. Rails was – and still is – a full featured framework packed with best practices in software development. It has inspired many other frameworks in a variety of programming languages. However, as the original, it has received much support from a growing community, resulting in a plethora of tutorials and documentation, and has become a rock solid web framework.

Applications using Rails follow the model-view-controller (MVC) architecture: models encapsulate business logic; views are the presentation layer providing the body of the response (e.g. HTML or XML); and controllers handle incoming requests and pass data from the models to the views (Rails contributors 2010).

Tools for automated testing are provided out of the box, and testing is taken very seriously by the community. Rails developers usually embrace test-driven development and/or behavior-driven development.

The Ruby on Rails framework contains several components including an object-relational mapper (ORM) and others that account for functionality such as the ability to send emails, utilities and extensions to the Ruby language, and managing data on external web services (Rails contributors 2010). The framework is easily extensible, and many common needs in web applications are solved by open source components that can be installed in a Rails application.

3.4. MongoDB

"MongoDB is a powerful, flexible, and scalable data store." (Chodorow & Dirolf 2010, 1.) It is not, however, only buzzwords, but a high quality product that suits web development particularly well.

MongoDB is a document-oriented database, allowing users to have a flexible schema. Documents are stored in collections, and there is no requirement that two documents in the same collection have the same schema. It may sound chaotic to someone who is used to relational databases, but the developer is always in control. MongoDB's high performance and easy learning curve are some of the factors that make it so appealing. (Lerner 2010a, 18–20; Lerner 2010b, 18.)

The database is designed for scalability, performance and high availability, featuring replica sets that can be used to distribute loads and provide automatic failover. A replica set can have many nodes, where one is the master node. If a master becomes unavailable, another node will be automatically promoted to master. Furthermore, auto sharding is available to provide horizontal scalability when the volume of data is too large that it becomes slow on a single machine. Sharding is a way of partitioning data and storing chunks on multiple machines. By combining sharding with replica sets, MongoDB can provide a high performant system with no single point of failure. (10gen 2010a; 10gen 2011a; 10gen 2011b; 10gen 2011c.)

Querying the database is done in JavaScript or using a driver for a programming language. Dynamic queries let the developer find data using any criteria. Data processing and aggregation can also be done by the database, using map/reduce. (10gen 2010b; 10gen 2011d.)

Locking and transactions are not supported by MongoDB for scalability and performance reasons. It is, however, possible to do atomic update operations on single documents. Modifying documents using these atomic modifiers is very performant. There is no need to query the document, and there is also no need for a response from the server, so the update returns immediately (making the application faster), while reducing network traffic. (10gen 2011e; 10gen 2011f.)

I believe that MongoDB is a great fit for web analytics. Atomic updates executed in a "fire and forget" manner make data collection easy to develop and fast. Automatic sharding solves the problem of dealing with enormous amounts of data. Dynamic queries account for rapid development, and map/reduce can be used for complex operations on large sets of data.

4. Product overview

When using Snowfinch for the first time, the user is prompted with a sign in form. On a new installation, a user is automatically created as described in the project's README file.

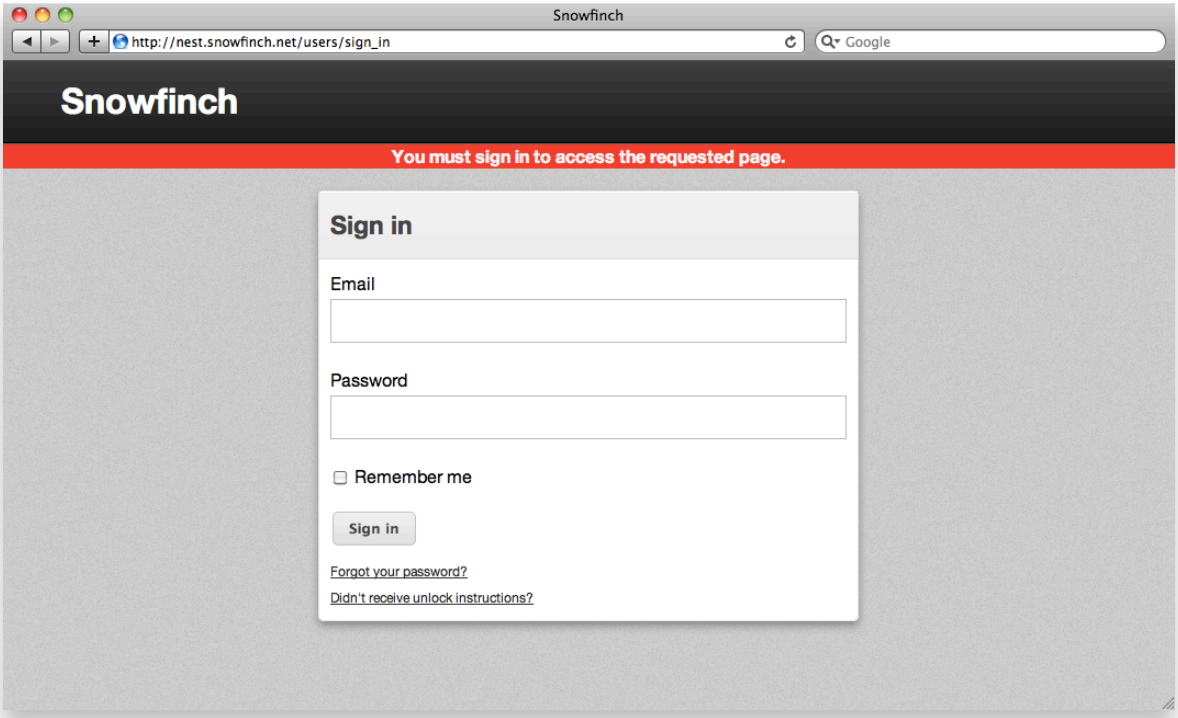

Figure 1. Sign in page with an alert message

The site navigation bar shows the root elements of the application (sites, users, account), clearly pointing out the active element. In addition, the sign out button is placed last in the navigation.

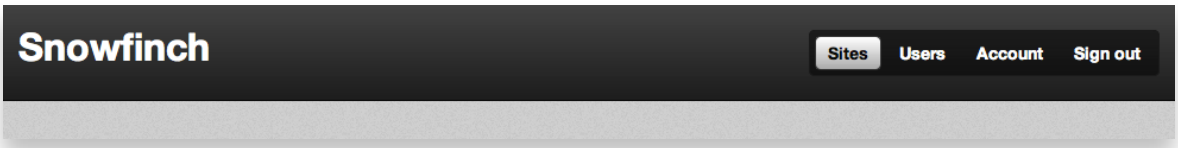

Figure 2. Header with navigation with Sites highlighted

Upon sign in, the user is prompted to create a site, by entering the site's name and the time zone. All analytics data will be displayed according to the selected time zone.

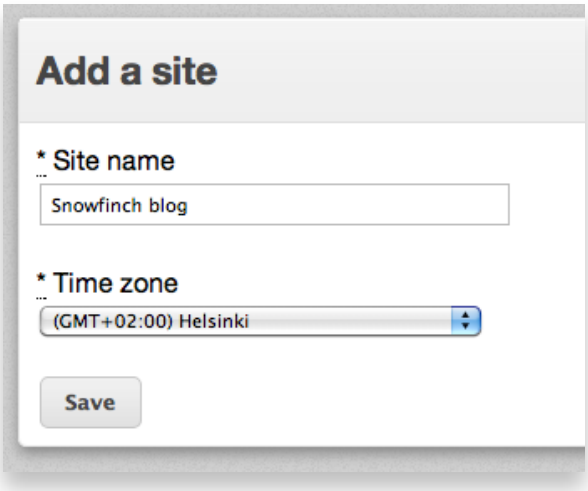

Figure 3. Form for adding a new site

To be able to track a site, a small JavaScript snippet must be inserted into the site, as instructed by the application. The snippet is presented as soon as a site is created, and can easily be copy-pasted into the appropriate location. This procedure is very common among web analytics products.

Figure 4. Instructions on adding the tracking code to a site

Once the tracking code is deployed and the site receives a visit, the previous screen is replaced with the site dashboard, a chart with the number of page views for each hour of the current day (orange line) and the previous day (grey line) for quick comparison. In addition, three counters at the top display the number of visitors on the site at the moment, the total amount of page views and unique visitors for the current day. Both the chart and the counters are automatically updated every second, allowing the user to follow how the site is performing presently.

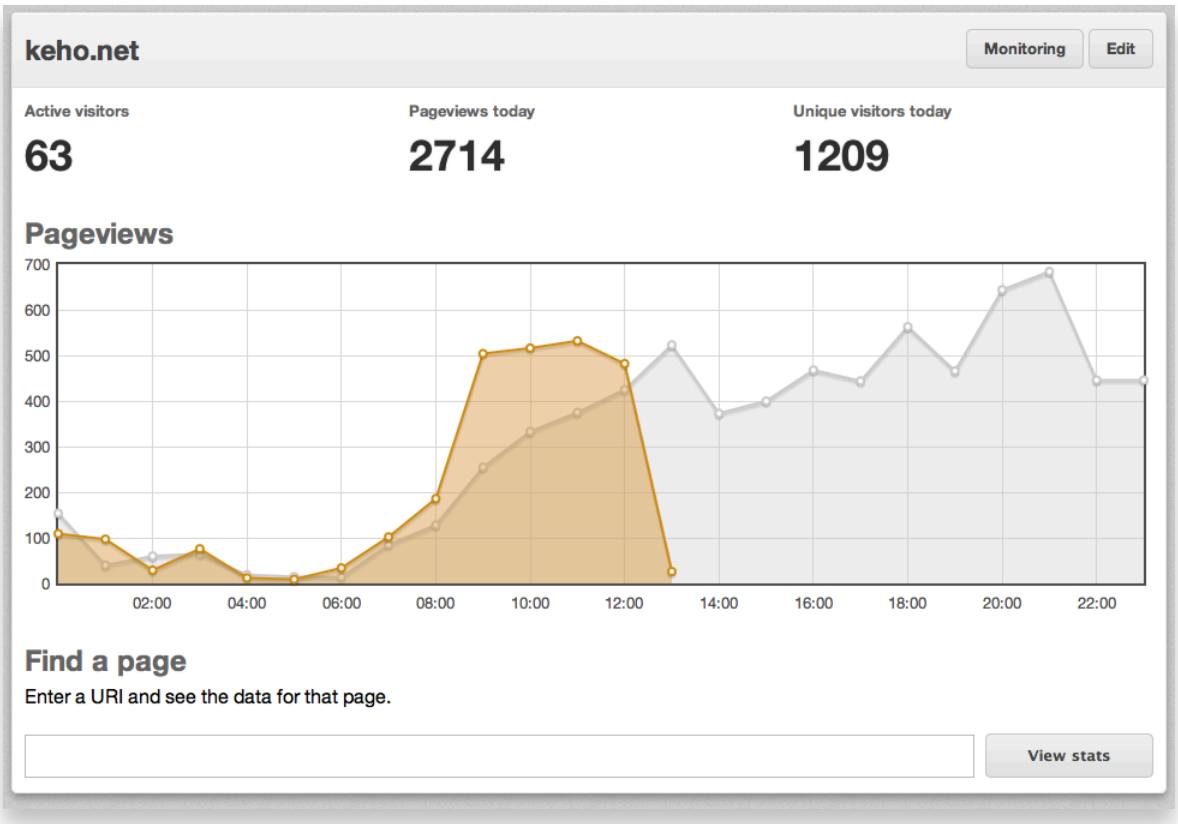

Figure 5. Site dashboard

It is possible to drill down to a specific page and see how well it is performing. By entering the URI of any page, a view similar to the site dashboard is displayed. A notice is shown to the user when there is no data for the requested page.

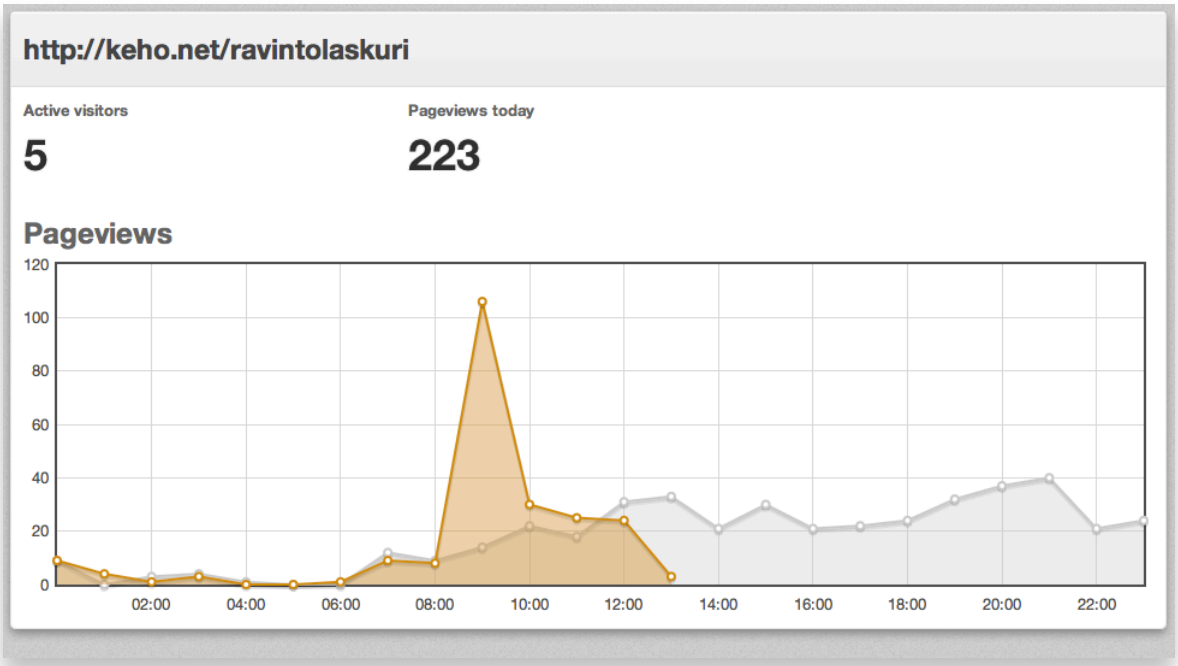

Figure 6. Page dashboard

Tracking campaigns is common functionality. Snowfinch takes the idea a bit further, and ships with a monitoring feature that has the concept of sensors. Two kinds of sensors are currently implemented, but the implementation is flexible enough to accommodate more in the future. The first type of sensor is triggered by the query string of a URI. A key-value pair is defined and any URI inside the site that contains the specified key-value pair is counted as an entry. This sensor allows for the usual campaign tracking.

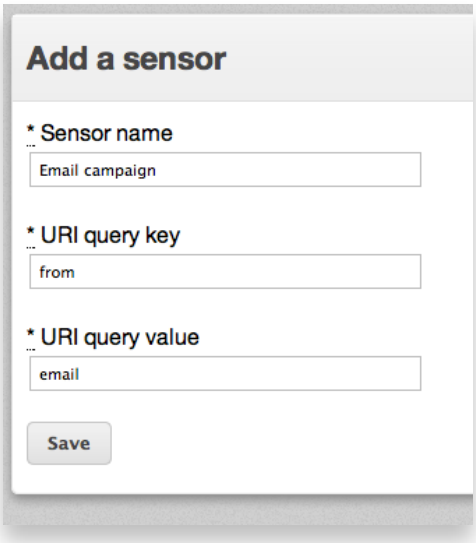

Figure 7. Adding a query based sensor

The second kind of sensor is based on the page referrer (the previous page). The hosts for several referrers can be added, and each entry to the site from any of the listed hosts will be recorded. Such functionality can be used, for example, to track how many entries the social media presence of a site brings.

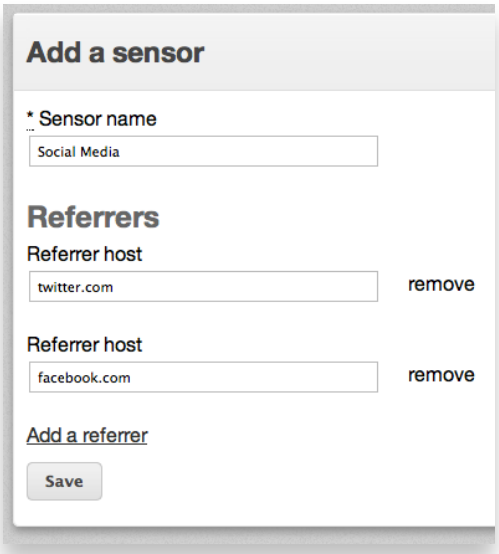

Figure 8. Adding a referrer based sensor

For each sensor, a chart is displayed, similar to the ones for a site or page, showing the number of entries.

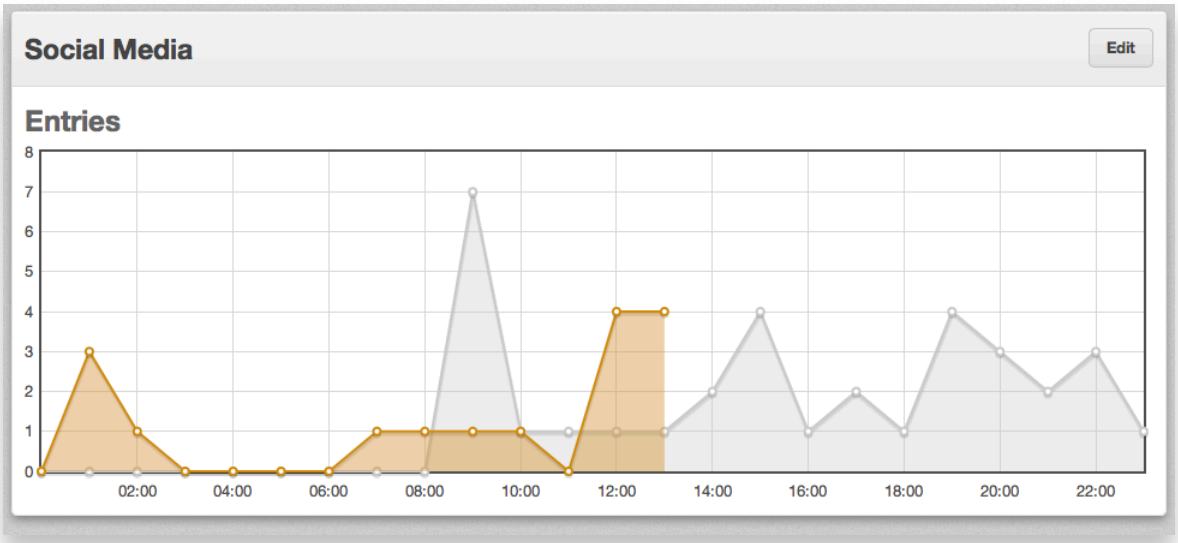

Figure 9. Sensor dashboard

Both sites and sensors can be edited, using a form similar to the one used to create each one. The edit page for a site also contains the tracking code, in case it is needed later on by the user. Edit pages also contain a button to remove the site or sensor.

New users can be created by entering an email address. The application will email the new user with a randomly generated password that can only be changed by the user, in the account page. For added security a user must always provide the current password when updating any account information.

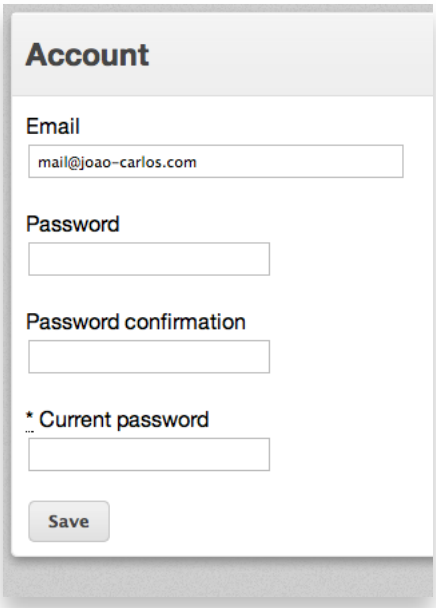

Figure 10. Account page

Users all have the same privileges and can access everything in the application. Authorization is outside the scope of this thesis work. The same applies to user interface design and usability. I did my best, however, to ensure that the user experience is not bad.

5. Implementation

5.1. Development methodologies

The project implementation was done according to Kisko Labs' development methodologies, which include but are not limited to: status updates provided often to involved parties; quick prototyping of uncertain functionality to study its feasibility, and make changes early on; workshops with interested parties to discuss features and implementation; test-driven development for clean and correct code with automated unit and acceptance tests; use of battle tested libraries and tools that are tested to work.

5.2. Tracking code

Most web analytics products track clicks on pages the same way: a JavaScript snippet is inserted into each page that needs to be tracked. A very simple approach is to perform a GET request to the server, passing all data to be stored as query parameters in the URI. The request needs to happen in the background when a page is loaded, and without interfering with the visitor's browsing experience.

To track visitors and visits, a cookie is used containing a unique random identifier. A very small, RFC4122 compliant, UUID (Universally unique identifier) version 4 generator in JavaScript was implemented to create the visitor identifiers. If the visitor does not have the cookie containing the identifier, one is generated and stored.

The tracking code must generate the URI with the query parameters containing all the data that the server needs to store. For Snowfinch, that means the URI of the page that is being viewed, the referrer (URI of the previous page, if any), the visitor's unique identifier, and the site identifier.

It may sound like Ajax is a good solution, but it is an overkill. The response is unnecessary, and the code would have to handle different browser implementations. The simplest way to execute the request is to instantiate a new Image object and set its source to the generated URI. To provide a proper response, the server will send an empty GIF.

In the end, all a site owner needs to do is copy a JavaScript snippet that the Snowfinch application provides when creating a new site in the system, and paste it right before

the end of the body tag. The snippet will, asynchronously, create a script element and place it inside the head element of the page. The script that will have its source set to a JavaScript file that performs the described functionality. Site or page specific data (site identifier at the moment) is passed to tracker code via a JavaScript object. By being loaded last, Snowfinch will never affect a site's performance, even if the server is down.

5.3. Data collection architecture

One important component of Snowfinch is the ability to collect and store clickstream data. There are several possible architectures, and finding a good balance between ease of deployment, speed and scalability is a requirement.

Here I discuss three possible architectures, their advantages and drawbacks, and the reasoning behind my choice. Each architecture builds on the previous one, thus becoming more complex.

All presented architecture diagrams contain an arrow pair on top of the component that handles the GET request performed by the JavaScript tracker, depicting a request and response. Text in italics is used to present the parts that are not readily available (need to be developed).

Note that no presented architecture is asynchronous, so the response only happens after everything else is done. The quicker data can be processed and stored, the more requests can be handled in any given period of time.

5.3.1. Monolithic Rails application

The easiest possible architecture is doing everything inside the Rails application. In other words, the Rails application that powers the user interface for analytics reports would also have a controller action to collect and store analytics data.

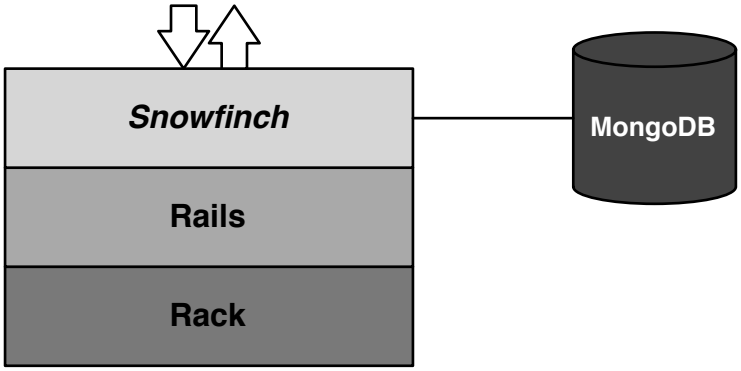

Figure 11. Monolithic Rails application

There are major drawbacks with this approach. First of all, since the action is called every time there is a click on a tracked web page, it should be performant enough, as there will be many in high traffic sites. Ruby on Rails is a great web framework, but the niceties it provides are not necessary for such a simple action, so all we end up with is unnecessary overhead. There is also the scalability issue. Scaling horizontally would mean that the whole application would have to be deployed, when maybe only the data collection part needs to be scaled.

This brings us to the next solution: decoupling the data collection part from the Rails application.

5.3.2. Rack application for data collection

As we want speed, the best we can do, while staying in Ruby land, is code the data collection component (collector) directly on top of Rack. As Rails 3 fully embraces Rack, we can even mount the collector on the Rails application.

If the collector is created as a Ruby gem it would be easier to either mount on the application or run one to multiple instances separately for better performance. As more traffic is expected, more instances of the data collector can be running.

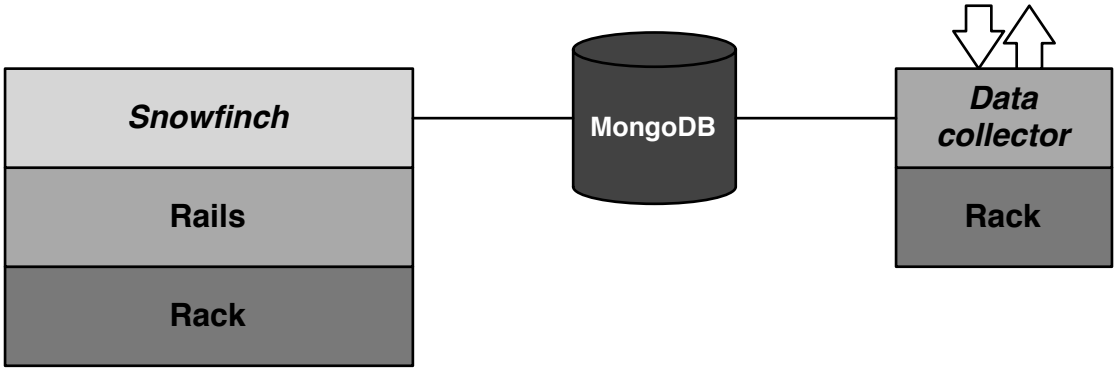

Figure 12. One Snowfinch process and one data collector process running, both connected to a MongoDB instance

At the moment, database operations done when a request comes in are writes to a MongoDB database using fire and forget queries, and one read. To reduce complexity for the time being, I decided not to use a non-blocking web server. This may or may not change in the future, but for now there is a read operation that may start performing poorly under load, and it happens on every request.

The next architecture tries to handle requests as quickly as possible, by separating the request handling and the data storage parts.

5.3.3. Separate request handling and storage

Handling requests and storing data in MongoDB are two separate concerns, so they can be split up. To quickly handle requests, we should do the bare minimum. Since the query part of the request URI contains all the data that is needed, we can simply pass it to the storage component. For easier distribution of processing and scalability, it is best to have a shared data store for the URI query.

Redis is a great fit for many reasons. It natively supports lists, where pushing and popping values is extremely easy and fast, making it behave like a FIFO (First In, First Out) queue. In addition, Redis supports blocking the connection on a pop operation when the list is empty. As soon as value is pushed to the list, the client that has been waiting the longest will get it. Such behavior reduces the code complexity associated with fetching the next element to be processed/stored.

The storage part can be implemented as a daemon, executing a blocking pop operation on the Redis server. Once the data from a request is available, we store it into MongoDB, just as in any of the two previous architectures.

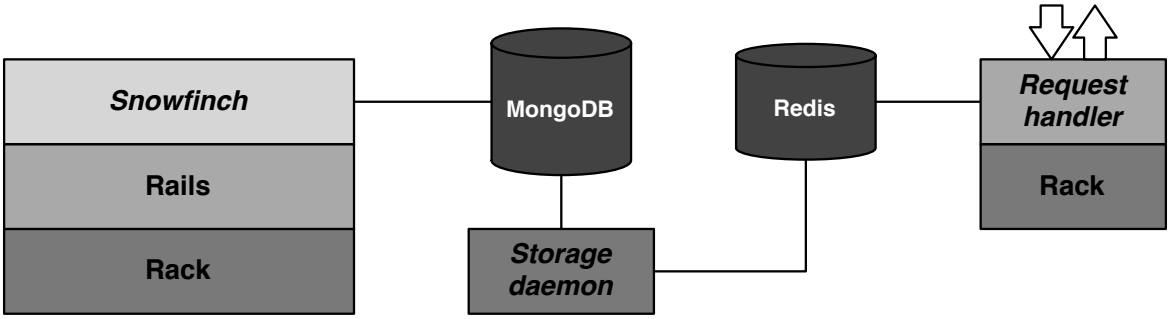

Figure 13. Redis acts as an intermediate data store between handling a request and processing its data and storing it in MongoDB

Any component of this architecture can be scaled individually as needed.
This offers a great advantage: less hardware can handle more traffic. By handling requests as quickly as possible, tackling traffic peaks is somewhat easier. With this architecture the number of requests that can be handled on a given period of time should always be higher than the amount that can be stored in MongoDB. During a traffic peak, data might not be available in realtime while the queue fills faster than it empties, but at least no requests should be lost.

5.3.4. Chosen architecture

Due to the current circumstances, I have decided to choose the second architecture. The required implementation time is closer to the first architecture than the third, and the biggest performance gain is seen when moving from the first to the second architecture. It provides a good balance between simplicity and performance, and is much easier to deploy than the third architecture.

Nobody needs to stick with my implementation, though. Swapping the provided collector with a custom made one is a fairly simple task. The data model must be respected, so that the Rails application knows its way around the data. There is only one caveat: the algorithms that sanitize and hash a URI are implemented in the collector and used in the Rails application. The Rails application expects something to implement `Snowfinch::Collector.hash_uri` and

`Snowfinch::Collector.sanitize_uri`. For that reason, it is easier to keep the original collector as a dependency, and use the same methods in the custom collector. While the actual implementation of these methods matter, they must be the same in both the collector and Rails application, or otherwise no data can be found.

5.4. Data model

The data model used to store analytics data affects the overall system performance. While storage capacity is cheap, input and output (I/O) operations are not. Using a data model that uses as little amount of space as possible, and having the most common accessed data stored in a single document is what I am striving for.

In MongoDB: The Definitive Guide, by Chodorow & Dirolf (2010, 169–170), the reader is presented with a simple schema for storing page views, where each document is similar to:

```
{
  "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)",
  "url" : "/foo",
  "views" : 5
}
```

For every hour that has any views for a specific page, a document will be created if it does not yet exist. Subsequent page views only need to increment the *views* counter by one. MongoDB features an upsert option that can be passed to any update operation that will create a document if no existing document matches the criteria. This features certainly reduces the amount of code and complexity required to create and update many of these small documents.

Snowfinch supports tracking of multiple sites, so there is the need for a site identifier. Also, to support subdomains, the full URI of the page is stored, not just its path. With the required additions, a single document would be:

```
{
  "site" : ObjectId("4d755dc3eca0261597000001"),
  "hour" : "Tue Mar 8 2011 9:00:00 GMT+0200 (EDT)",
  "uri" : "http://blog.snowfinch.net/post/3254029029/uuid-v4-js",
  "views" : 20
}
```

The *site* field is a reference to a document in a *sites* collection, a collection containing the time zone of a site. The data in the *sites* collection is also stored in a relational

database, making the Rails development simpler. It is also in MongoDB, because it is used by the collector, and it is much easier to deal only with MongoDB than having to also support a relational database.

When looking a collection full of documents containing the presented schema, repetition becomes clear. The *site* and *uri* fields are repeated over and over again. However, that is not the only issue. To display the page views for current and previous days (a common scenario), between 25 and 48 documents have to be queried, depending on the time of the day. To calculate the amount of page views during a month, more than 700 documents would be needed. Maintaining a similar collection for the totals of each day would decrease the previous number, but there is a more elegant solution. After all, a schema-less database was selected for a reason. After some thought and research on how other people have solved this problem, I ended up storing documents in the following format:

```
{
  "s" : ObjectId("4d755dc3eca0261597000001"),
  "u" : "http://blog.snowfinch.net/post/3254029029/uuid-v4-js",
  "h" : "1fTvAhefD5kPDkss3RmKa2KYa1L",
  "y" : 2011,
  "c" : 30,
  "2" : {
    "c" : 30,
    "27" : {
      "c" : 20,
      "22" : { "c" : 15 },
      "23" : { "c" : 5 }
    },
    "28" : {
      "c" : 10,
      "0" : { "c" : 10 }
    }
  }
}
```

It isn't self explanatory, so let's walk through it. First of all, keys have been shortened to one character whenever possible. The first five keys stand for *site*, *uri*, *hash*, *year* and *clicks*. The *hash* field is the SHA-1 digest of the URI, encoded in base 62. It is used by the Rails application to identify a page and can be used in other MongoDB collections to refer to a specific page. The *clicks* field contains the total amount of page views for a given *year*.

The nested hashes contain the number of page views for a month, day and hour. The example contains data for the month of February, that had 30 page views in total. The

27th day had 20 page views: 15 between 22:00 and 23:00, and 5 between 23:00 and midnight. The next day (28th), had 10 page views, all between 00:00 and 01:00.

Months, days, and hours are added to documents as needed, so if a page is only viewed once during the period of year, its document will be much smaller than the one of a page that is accessed often. All of this is possible because MongoDB's increment modifier can be used on a deeply nested hash, even if the keys are nonexistent.

I decided to have one document per page, per year. The biggest reason behind this choice was that it felt like good organization. MongoDB does have a limit on the size of each document, which used to be four megabytes and recently got increased to sixteen with the release of version 1.8.0. Even though it is unlikely that any document would ever reach that size, the world is full of technological mistakes based on the unlikeliness of events.

The *page_counts* collection stores the documents containing the number of page views per page, per year as described. A similar collection named *site_counts* contains documents with the same structure, except that they account for the page views across an entire site. Therefore, no URI or its hash are stored.

For the monitoring functionality a *sensor_counts* collection is kept. It is also similar to the *page_counts* collection, but contains a sensor identifier instead of any page URI information. Sensor information is stored in the *sites* collection, and is used to check whether any request matches any sensor data.

To track visitors, each click is recorded in the *visitors* collection. As the daily unique visitors are displayed for a site, each document contains the visitor UUID sent by the JavaScript tracker, the date, the site it refers to, and the number of clicks from the visitor for the specific date:

```
{
  "c" : 8,
  "d" : "2011-03-06",
  "s" : ObjectId("4d70c6d11e36940536000001"),
  "u" : "682fc061-7ba1-410f-8d57-adb9a9ffb1a2"
}
```

To know how many unique visitors a site had in one day, one must simply get the count of documents for a specific site and date.

Active visits are stored in the *visits* collection, according the following schema:

```
{
  "c" : 2,
  "h" : 1299370585,
  "p" : [
    "eh1ByWS63b0xs6ZRUw0SFhTJ5KG",
    "3TdjLOhYFWVdBPbWYbzbHMkyS73"
  ],
  "s" : ObjectId("4d70c6d11e36940536000001"),
  "v" : "682fc061-7ba1-410f-8d57-adb9a9ffb1a2"
}
```

The keys are, in order: number of clicks (page views) for the visit; timestamp of the last click (heartbeat) in seconds since the Unix epoch; an array of page URI hashes, matching the hash in the *page_counts* collection; the site identifier; the visitor UUID.

The heartbeat is updated every time a new click is set. A visitor is considered active if the latest heartbeat has occurred in the last 15 minutes. Therefore, a new visit document is created if none exists where the heartbeat has happened in the past 15 minutes for a given site and visitor UUID.

The page hash is added to the array if it is not yet present, and is used to query the amount of active visitors for a single page.

Every single document in MongoDB that is stored by the data collector is created and updated by the same operation, by using the upsert option and atomic modifiers.

6. Conclusion

With the right set of tools, it is possible to build a quality realtime web analytics application from scratch in a short period of time, as proven by this thesis work.

The quality of the delivered application is high, as it contains clean, readable code that is fully covered by automated tests. There is no doubt that the lack of a robust test suite would have impaired the resulting application. As with any software development, certain changes unexpectedly broke parts of the application, and it is likely that such bugs would not have been found if it weren't for automated tests.

Being able to visualize a site's traffic in realtime while launching a campaign or making changes allows for mistakes to be dealt with sooner, and shows which modifications work. Owning such data surely gives a competitive advantage to any business.

Kisko Labs, the thesis sponsor, will benefit from using Snowfinch. Newly developed web applications will be tracked by Snowfinch and available to their clients.

There is no better way to learn about web analytics than to develop a product around it. It is important to know the limitations and where the focus needs to be, especially when the available resources are scarce. I strongly believe that understanding what is happening right now to be able to respond quickly is what matters.

The chosen tools turned out to be as good as expected. Absolutely nothing got on the way while developing Snowfinch. Once again I must praise MongoDB for its excellent features and painless usage.

7. Further development

With Snowfinch being an open source project, the end of this thesis work is just the beginning. Knowing where traffic is originating from is important to discover and identify potential markets and recognize the importance of current ones. With that in mind, tracking the traffic sources is most likely the next feature to be implemented.

The current implementation focuses exclusively on what is happening on the current day and compares it to the previous day. However, since old data is stored, it is possible to display it. Some parts of the code are ready to accommodate this feature already, so it is mostly user interface work that needs to be done.

Web applications usually contain authentication mechanisms and store user specific data. In certain scenarios it may be convenient to know specifically which users are doing what. By adding custom visitor data (e.g. name and/or email address) to the JavaScript tracker, it could be possible to record that information. This allows, for example, the identification of loyal visitors. Another scenario may be that a user has visited a product page many times but has not purchased the item. Perhaps offering a discount would be a great way to improve a customer's image of the seller.

For A/B testing to be successful, it is necessary to know its results. With support for adding custom visit data through the JavaScript tracker, Snowfinch could not only track the results of the testing, but also inform the tracked application of which test samples have been shown to a specific user. In doing so, implementing A/B testing becomes easier as the application being tested would not need to implement as much functionality.

Data collection is fast at the moment, but since data is being retrieved from the database on every request, there is some time spent by the collector waiting on this blocking I/O operation. A fully asynchronous data collector would be able to handle more requests, by not waiting while data is being fetched from the database.

Bibliography

10gen 2010a. Sharding and Failover. URL: http://www.mongodb.org/display/DOCS/Sharding+and+Failover. Quoted: 03.05.2011.

10gen 2010b. Querying. URL: http://www.mongodb.org/display/DOCS/Querying. Quoted: 03.05.2011.

10gen 2011a. Replication. URL: http://www.mongodb.org/display/DOCS/Replication. Quoted: 03.05.2011.

10gen 2011b. Sharding. URL: http://www.mongodb.org/display/DOCS/Sharding. Quoted: 03.05.2011.

10gen 2011c. Sharding Introduction. URL: http://www.mongodb.org/display/DOCS/Sharding+Introduction. Quoted: 03.05.2011.

10gen 2011d. MapReduce. URL: http://www.mongodb.org/display/DOCS/MapReduce. Quoted: 03.05.2011.

10gen 2011e. Atomic Operations. URL: http://www.mongodb.org/display/DOCS/Atomic+Operations. Quoted: 03.05.2011.

10gen 2011f. Updating. URL: http://www.mongodb.org/display/DOCS/Updating. Quoted: 03.05.2011.

Chodorow, K. & Dirolf, M. 2010. MongoDB: The Definitive Guide. O'Reilly Media, Inc. Sebastopol, CA, USA.

Kaushik, A. 2010. Web Analytics 2.0. Wiley Publishing Inc. Indianapolis, IN, USA.

Kohavi, R., Henne, R. & Sommerfield, D. 2007. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers not to the HiPPO. KDD '07 Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 959–967. URL: http://dx.doi.org/10.1145/1281192.1281295. Quoted: 04.05.2011.

Lerner, R. 2010a. MongoDB. Linux Journal, Issue 193, pp. 18–20.

Lerner, R. 2010b. Advanced MongoDB. Linux Journal, Issue 194, pp. 18–22.

Neukirchen, C. 2007. Introducing Rack. URL: http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html. Quoted: 03.05.2011.

Nielsen, J. 2005. Putting A/B Testing in Its Place. URL: http://www.useit.com/alertbox/20050815.html. Quoted: 04.05.2011.

Phippen, A., Sheppard, L. & Furnell, S. 2004. A practical evaluation of Web analytics. Internet Research, Volume 14, Number 4, pp. 284–293. Emerald Group Publishing Limited. United Kingdom. URL: http://dx.doi.org/10.1108/10662240410555306. Quoted: 20.03.2011.

Rack contributors 2011. Rack project page on GitHub. URL: https://github.com/rack/rack. Quoted: 03.05.2011.

Rails contributors 2010. Getting Started with Rails. URL: http://guides.rubyonrails.org/getting_started.html. Quoted: 03.05.2011.

Ruby community. About Ruby. URL: http://www.ruby-lang.org/en/about/. Quoted: 03.05.2011.

Sen, A., Dacin, P. & Pattichis, C. 2006. Communications of the ACM, pp. 85–91. ACM. New York, NY, USA. URL: http://dx.doi.org/10.1145/1167838.1167842. Quoted: 21.03.2011.

Waisberg, D. & Kaushik, A. 2009. Web Analytics 2.0 Empowering Customer Centricity, Part II. SEMJ.org, Volume 2, Issue 2. URL: http://www.semj.org/documents/Web_Analytics_20_SEMJ.pdf. Quoted: 10.03.2011.

Weischedel, B. & Huizingh, E. 2006. ICEC '06 Proceedings of the 8th international conference on Electronic commerce: The new e-commerce: innovations for conquering current barriers, obstacles and limitations to conducting successful business on the internet, pp. 463–470. ACM. New York, NY, USA. URL: http://dx.doi.org/10.1145/1151454.1151525. Quoted: 21.03.2011.

Appendices

Appendix 1. Source code

Complete source code with automated tests is available at the following addresses:

– https://github.com/jcxplorer/snowfinch

– https://github.com/jcxplorer/snowfinch-collector.

Appendix 2. Snowfinch project README

```
Snowfinch
=========

Snowfinch is a realtime web analytics application written in Ruby, and
using MongoDB. It provides everything you need to track and visualize
analytics from multiple sites. While it may not be as full featured as
commercial alternatives, it is free and released under the MIT license.

Currently, Snowfinch supports: tracking of pageviews, active visits, and
visitors for a site; pageviews and active visitors for a given page;
monitoring based on a URI query key-value pair, or based on any given
number of referrers (think campaigns or tracking the number of visits
from social media sites). It's not much, but this is only the beginning.
Take a look at the Roadmap section to see what's coming.


Getting started
---------------

You will need recent versions of Ruby (1.9.2 recommended) and MongoDB.

If you don't have Bundler installed, install it with:

    gem install bundler

Clone the repository:

    git clone https://github.com/jcxplorer/snowfinch.git

Edit the following files to suit your needs:

* `config/database.yml`
* `config/snowfinch.yml`

Some information is stored in a relational database, so you can use any
that is supported by Rails. By default it will use PostgreSQL. Edit the
`config/database.yml` and `Gemfile` files accordingly if you want to use
a different database.

Install the application dependencies by running `bundle` in the
application directory.

Run `rake db:setup` to create a database and an initial user.

You are now ready to launch the application using your favorite Ruby
application server such as Passenger or Unicorn. Sign in with the email
address _user@snowfinch.net_ and password _snowfinch_. Don't forget to
change those credentials on your account page!
```

High performance and scaling

Are the pages you are tracking getting too many hits? Good for you! Fortunately you don't need to deploy entire instances of Snowfinch in multiple hosts. All data collection and storing is a done by a Rack application available as a gem. Just install the `snowfinch-collector` gem, and deploy using a `config.ru` file similar to the following example:

```
require "snowfinch/collector"
Snowfinch::Collector.db = Mongo::Connection.new.db("snowfinch")
run Snowfinch::Collector
```

Now you can deploy that on several hosts and get the web scale fix you were looking for. Throw some MongoDB shards at it as needed.

By default `snowfinch-collector` is mounted in the Rails application at "/collector" for easier deployment. If you run your own instance, remember to make sure that the MongoDB database is the same that the Rails application is using, and configure the URI to the collector in `config/snowfinch.yml`.

Just running one separate instance of `snowfinch-collector` will perform better than when mounted in the Rails application, as there will be no overhead from the Rails router and middleware.

Roadmap

* Display data for any given period of time (currently the past 2 days).
* Collect and display referrer information.
* Custom visitor tagging.
* Geographic location of visitors.
* Filter by custom page metadata (e.g. for A/B testing).
* Asynchronous data collector.

Reporting bugs

If you think you found a bug, please file an issue on GitHub. Please try your best to provide steps on how to reproduce the issue you are experiencing. Failing tests are highly appreciated!

Contributing

If you want to contribute, fork away and send a pull request. If you want to be sure something will be merged before spending time on it, feel free to contact me. Don't worry, hard work won't be thrown away.

Always try to provide tests with your pull requests. If you're not sure on how to test something, mention it on the pull request so that you can get some help.

I appreciate if you try to stick to the existing coding style, but I can always refactor later on. If you can stick to 80 characters a line, please do. If you don't yet, try it. It makes you write better code.

Copyright and licensing

```
Copyright 2011, João Cardoso. Snowfinch is released under the MIT
license.
```

Appendix 3. The MIT License