UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Robert Joonas

Tracking Time on Page in Web Analytics

Bachelor's Thesis (9 ECTS)

Supervisor Uku Täht
Supervisor Helle Hein

Tartu 2022

Tracking Time on Page in Web Analytics

Abstract:

This thesis explains some known Web analytics issues with tracking *time on page* and attempts to build a new functionality to make the metric more accurate and reliable. In the course of this work, a new solution is developed on top of an existing Web analytics software called Plausible Analytics. The functionalities programmed by the author were written in JavaScript on the front end and in Elixir on the Backend. The development process is described step by step, bringing up the most important functionalities built and the reasoning behind some implementation choices. Finally, the thesis assesses the performance of the delivered solution by gathering data from the Plausible website.

Keywords: Web Analytics, time on page, Beacon API

CERCS: P175 - Informatics

Lehe külastusaja arvutamine veebianalüütikas

Lühikokkuvõte:

Käesolev bakalaureusetöö selgitab veebianalüütika levinud probleeme seoses lehe külastusaja arvutamisega ning üritab neid probleeme lahendada uue tarkvaralahendusega. Töö käigus integreeritakse innovatiivne lahendus olemasoleva veebianalüütika tööriista Plausible Analytics rakendusse. Autori loodud uued funktsionaalsused on programmeeritud keeltes JavaScript (eesliidese jaoks) ja Elixir (tagarakenduse jaoks). Töö kirjeldab järk-järgult uue lahenduse arenduskäiku, tuues välja kõige olulisemad funktsionaalsused ja põhjendades mõningaid valikuid implementeerimisel. Lõpuks annab käesolev töö valminud lahendusele ka hinnangu, kasutades selleks Plausible'i ettevõtte veebilehelt kogutud andmeid.

Võtmesõnad: Veebianalüütika, lehe külastusaeg, Beacon API

CERCS: P175 - Informaatika

Table of contents

1. Introduction

The majority of businesses today have websites for various purposes [1]. For a website owner, Web analytics is the key to understanding how their website is actually used by people browsing the Web. The purposes of knowing the details about the site usage can be very different. For example, a site owner might want to see how many users are taking the desired actions on their page and where they are coming from to make better business decisions. Also, analytics insights are helpful in many aspects (such as identification of bottlenecks in the design and the monitoring of website availability) of improving the user experience [2].

There are many analytics services that provide their users with different usage reports with different metrics and properties. This thesis explicitly focuses on the *time on page* metric, its concerns, and building an innovative tracking solution to tackle the problems. Apart from the details about the *time on page* metric, Chapter 2 will also run the reader through the basics of Web analytics, including front-end tracking with JavaScript and the back-end functionality for persisting the usage information.

The known problem with the *time on page* metric in Web analytics is that it cannot be measured for the last page that the user views before leaving [3]. Google Analytics, currently being the most popular analytics service [4], also struggles with the same limitation.

In his blog post, Matthew Edgar gives clear examples of this problem in Google Analytics. To be able to calculate how long a user spent viewing a page, both start and end moments need to be available. The ending moment is defined as loading the next page, but when a visitor leaves the site, the last page viewed does not have any ending moment [5].

At the center of this piece of writing is the Web analytics software of Plausible Analytics[1]. The company was founded in 2018 by Uku Täht, the primary supervisor of this thesis. Plausible also struggles with the same *time on page* issue. Chapter 2.2 explains Plausible's current solution and its concerns with tracking *time on page*.

This thesis aims to integrate a new solution into the existing Plausible software that reports a new type of event from the client-side to determine the ending moment of a page view. Chapter 3 describes the development process on both the front-end script and the back-end application.

Chapter 4 assesses the performance of the delivered new tracking behavior with actual data gathered from the Plausible website itself by comparing the data reported by the existing and the new tracking solutions. The collected stats for both the existing and the new solution are publicly available online on Plausible's production[2] and testing[3] domains.

[1] https://plausible.io

[2] https://plausible.io/plausible.io?period=month&date=2022-03-08

[3] https://testing.plausible.io/plausible.io?period=month&date=2022-03-08

2. Background

Web analytics is the process of gathering and analyzing the traffic on a website with the aim of providing useful information to the site owner [6]. The majority of Web analytics tools use JavaScript to send analytics events from the tracked website to their application server where it gets processed and persisted [7].

An intuitive benefit of this knowledge is being able to analyze how a site is performing over time. But furthermore, by knowing the details about the site usage, one might make better business decisions. Having worked as a customer support person in a Web analytics company, the author is familiar with numerous use cases for blogs, e-commerce sites, news magazines and other types of websites, where knowing the user behavior can help improve the business. For example, by knowing what campaigns are driving more visitors to the site, the company can invest more in one specific campaign to expand its audience.

Čegan and Filip have written in 2017 that Web analytics also supports site owners in improving their visitors' user experience. It can help identify the bottlenecks and errors in the user interface design and monitor the site availability [2].

There are many services out there that offer Web analytics. Some of the most popular ones amongst them are Google Analytics, Adobe Analytics and SEMRush [8]. All tools have their own tracking methodology and provide different insights. Among the different metrics provided is also *time on page* which is surfaced in many analytics tools.

2.1 Importance of *time on page* in Web analytics

For some site owners, it is important to know the average amount of time that the users spend viewing a single page on their site.

As Adam Steele has written in his blog post, *time on page* can provide insight into how engaging the content appears to the user. A lower average page view duration might mean that the content is not interesting enough [9].

Moreover, the *time on page* metric will express whether a page is attracting the right kind of visitors or false leads (people who mistakenly end up on the page) [10].

2.2 Tracking *time on page* in Plausible Analytics

Plausible Analytics[4] is an open-source Web analytics software founded in 2018. It focuses on privacy-friendly website tracking and provides its users with a simple yet actionable dashboard with the key metrics in Web analytics - including *time on page*.

The Plausible application software is at the core of this thesis as the work focuses on building improvements in the application to track *time on page* more accurately. To make the topic easier to follow for the reader, this chapter also explains some terminology used in Chapter 3.

[4] https://plausible.io

2.2.1 Plausible script and sending events on the client-side

To better understand tracking *time on page* at the application level, it is best to be familiar with the concept of sending analytics events. This subchapter will share some details about how analytics events are sent from browsers.

Following the industry standard [7], Plausible uses JavaScript to send analytics events from the browser to its application server. For simplicity, let's say that the application consists of a back-end application and a front-end JavaScript file called *script.js*. There are also other ways to install Plausible on a website, but these are out of scope for this work.

The *script.js* file contains all the logic required to automatically send analytics events from a webpage. By default, as soon as this script is loaded and executed on a webpage, it makes an HTTP POST request to Plausible's application server. Each such request sent from the browser automatically includes a user agent and an IP address to uniquely determine a visitor (see Figure 1). The body of this POST request (see Figure 2) also contains important event parameters such as the name of the event, *domain* (the website to which this event belongs), the URL to determine the exact page path on which the event happened, and some additional properties that we are not focusing on in this thesis.

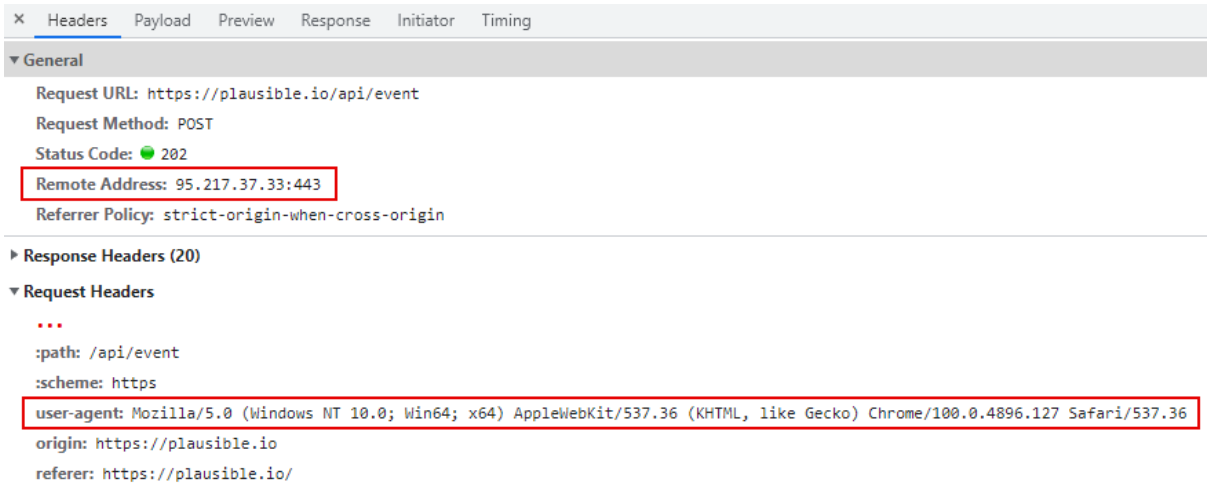

Figure 1. Request header

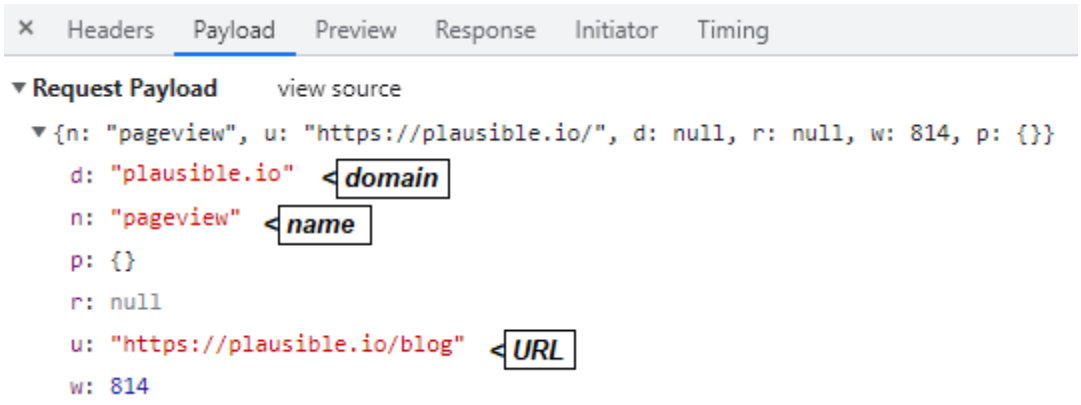

Figure 2. Pageview event request body

Every time a new page is loaded on the website where Plausible is installed, the browser executes the JavaScript code again and sends another *pageview* event to the server. With the current tracking method, every such *pageview* event marks the end of the prior page view in that session (if it exists). Sessions in Plausible (and in other analytics services) represent a set of timestamped events. Understanding sessions is necessary to comprehend both the current solution of tracking *time on page*, and the development work described in Chapter 3. Therefore, the following subchapter will give an overview of how a session is created in the application.

2.2.2 Using sessions to calculate *time on page*

The application back-end is responsible for structuring all the data that is continuously sent from browsers and persisting this information in the database. The data is stored in two structures: *events* and *sessions*. Both these structures have their own database table in which they are kept. The *events* table has a *session_id* foreign key field that determines exactly one session for an event.

Tracking *time on page* in the existing solution is session-based, meaning that the event structures do not include any information about how long a page was viewed. Instead, the *time on page* value is calculated as the time difference between a *pageview* event and its subsequent event.

In peak times, Plausible receives hundreds of events in a single second. For all these events, to find the subsequent event, it needs to know which session they belong to. As sessions are basically a group of events sent by one visitor, Plausible first has to uniquely determine the visitor for the incoming event. This is done with the help of the IP address and user agent fields in the request header. Then, these values are used to generate a unique *user_id* for the visitor.

But one visitor might still visit multiple websites that are tracked with Plausible simultaneously. As visits on different websites should not be interpreted as the same session, Plausible also uses the *domain* field sent with the event request body to uniquely determine a session.

For handling the mentioned logic, Plausible has a module called *Session.Store* that implements an Elixir GenServer. This module is like a constant process running in the application that keeps the state of all ongoing sessions. They are kept in memory as a map data structure, with tuples of *user_id* and *domain* as keys and the session objects as values. To not bloat the memory, sessions are dropped from the state map once 30 minutes have passed from the last session event. This means that the session is forgotten, and there is no way to add any more events to it.

When an event is received, the application creates an event object containing the *user*_id and the *domain* fields. As these fields combined make up a unique key for the session, the application will try to find a session with this unique key. If such a session is present in the memory, the event will get its ID as the *session_id* value and be tied with this session.

Otherwise, if there is no such session in the store, a brand new session will be initiated, for which this event will become the first one.

Figure 3 is an illustration of a session that consists of three events. The *path* attribute represents the exact path of the webpage on which the *pageview* event happened, and the simplified timestamp marks the time when this event was received. There is also the *session_id* field which illustrates that all these three events are a part of the same session.

Having an illustration of a session with its events, it's a good place to define a term used in the next two subchapters – exit page. The page */subscribe* in Figure 1 is the exit page of this session, pointing to the fact that the user left the site after viewing this page.

Figure 3: Simplified session example

Following the example of Figure 3, to know how long the */blog/post-1* page was viewed, the application uses the timestamp value to calculate the time difference with its subsequent page view on the path */subscribe*. This is currently done with specific database queries that join the events and sessions tables which will not be further explained as it is out of scope for this work.

2.2.3 Limitations in the current solution

A session-based *time on page* tracking is not the ideal solution because the duration of exit page views cannot be calculated. On the plausible website in the year 2021, there was a bounce rate[5] of 62% which can also be seen on the public dashboard[6]. This means that roughly 6 out of 10 visits only included one exit page view and did not include any information about the view duration.

Furthermore, even the sessions where multiple pages are viewed will miss out on the exit page view duration. The proportion of missing data is even more significant for SPA-s (single-page applications). This is because SPA-s allow users to see all the content on the website without requesting the whole pages from the server [11], and without reloading the page, the JavaScript code sending the analytics events will not be executed.

[5] Bounce rate is the percentage of visitors that leave after viewing the first page.

[6] https://plausible.io/plausible.io?period=year&date=2021-05-10

There is no default value for *time on page*. Instead, all the missing values will be ignored when calculating the average. This, in turn, makes the session-based *time on page* metric unreliable.

Depending on the nature of a page, the time a visitor actually spends on it just before leaving might vary, but cutting this large amount of data out of the equation will still make the *time on page* metric very biased.

2.2.4 The idea of improvement with Beacon API

For tackling the issue and reporting a *time on page* value also for the exit page views in a session, the idea was that a new type of event representing the end of a page view could be sent from the client-side. Clearly, narrowing it down to just the exit pages was impossible as there is no way to know whether a visitor is planning to view more pages or not. So every *pageview* event had to have an ending timestamp to use in the calculation of its duration.

While the idea of the new approach was simple - capture the moment when the page is closed and report the end event, there was one main challenge with the implementation - executing event requests to completion when the page has been unloaded.

The current approach uses the *XMLHttpRequest* JavaScript object to send the POST requests to the application server. When closing the browser tab or window, these requests are not guaranteed to run to completion, as the browser automatically cancels all the ongoing requests when a page is unloaded. With the use of *XMLHttpRequest*, a known workaround to reliably send the data is delaying the page unload until the request has received a response back from the server. While this is an excellent way to reliably send all the necessary information, it does not justify hurting the user experience by creating a delay in closing the page.

According to MDN Web Docs [12], Beacon API is used to send non-blocking requests to a web server and run these requests to completion without expecting a response back from the server. The documentation even mentions that the main use case of Beacon API is sending analytics events to the server.

As this seemed to solve the issues mentioned, the plan was to implement ending page views with Beacon API requests.

3. Development

This chapter describes the development process of integrating a new Beacon API solution with the existing Plausible front-end script and making several modifications to the existing Plausible back-end application to accept a new type of event and handle the new way of calculating time on page.

3.1 Front-end script development

There were several steps to implementing the Beacon API tracking solution on the front end. This chapter describes the development process and explains some reasoning behind the implementation choices based on Web research.

3.1.1 Script extension

Plausible has the default *script.js* JavaScript file, which includes only the basic *pageview* tracking functionality. It also allows a convenient way for injecting other specific tracking functionalities into the script via script extensions. For example, two existing script extensions are *script.local.js* (to track localhost visits) and *script.manual.js* (to allow manual triggering of *pageview* events). Plausible compiles all the extensions into separate JavaScript files that the clients can use by specifying the desired functionality in the filename.

As the integration of Beacon API was also expected to add extra complexity and increase the size of the script, it was feasible to add this functionality as a *script.beacon.js* extension.

3.1.2 Capturing the end of a page view

The goal of this integration was to be able to send events from the browser at the end of a page view. The initial step was to determine what browser events can be relied on to initiate the Beacon API requests. To know this and understand how Beacon API is meant to be used, the author did some research on the Web.

A blog post by Tero Piirainen [13] criticized the performance of Beacon API and showed its high failure rates on many major browser versions. However, the discussion in the comment section of the post also contained very useful information from another point of view. Ilya Grigorik explained in his comments [13] that the main reason for failure in the study was not the Beacon API itself but wrongly relying on the *beforeunload* browser event. Instead, he claims that a combination of *visibilitychange* and *pagehide* events should be used to initiate the requests.

In his own blog post from 2015, Grigorik [14] explains which browser events should be used to reliably capture leaving the website. He writes that on mobile platforms, *pagehide, beforeunload,* and *unload* events cannot be relied on, as an active application can transition into a background state and be closed without ever triggering the events. Instead, he recommends relying on the Page Visibility API. Based on his own testing, in the post Grigorik [15] also shows a matrix (see Figure 4) with different mobile and desktop situations and what events are fired in these situations.

		pagehide	beforeunload	unload	visibitilityChange visible -> hidden
Chrome desktop	Close visible tab	y	y	y	n (bug)
	Close background tab	y	y	y	y
	Navigate away	y	y	y	n (bug)
Chrome mobile	Home-button, swipe away in app switcher	n	n	n	y
	Task-switch, swipe away	n	n	n	y
	Navigate away	y	y	y	n (bug)
Firefox desktop	Close visible tab	y	n	y	y
	Close background tab	y	n	y	y
	Navigate away	y	y	y	y
Firefox mobile	Home-button, swipe away in app switcher	n	n	n	y
	Task-switch, swipe away	n	n	n	y
	Navigate away	y	y	y	y
Safari desktop	Close visible tab	n	y	n	n (bug)
	Close background tab	n	y	n	y
	Navigate away	y	y	n	n (bug)
Safari mobile	Home-button, swipe away in app switcher	n	n	n	y
	Task-switch, swipe away	n	n	n	y
	Navigate away	y	n	n	n (bug)
Edge desktop	Close visible tab	y	y	y	n (bug)
	Close background tab	y	y	y	y
	Navigate away	y	y	y	n (bug)

Figure 4. Matrix of browser events firing in different situations [15]

Based on the matrix, to minimize the failure rate and make it as reliable as possible on different browsers and devices, the author chose to use a combination of *visibilitychange*, *pagehide* and *beforeunload* events for capturing the end of a page view.

3.1.3 Information to send with the Beacon requests

Having chosen the browser events to listen to, the author also needed to figure out what information should be sent to the server with these requests to determine which page view is ended.

The most straightforward approach was introducing a new field for the event object, called *event_id*. This did not exist before because there was no need to uniquely determine an event. The value for this field had to be generated. To keep the front-end script as fast and optimized as possible, the *event_id* generation had to be done on the back end. This will be further explained in the next chapter.

With the *event_id* generation done on the back end, it had to be returned as an HTTP response from the server to be able to send the same value back to the application once the page view has ended.

3.1.4 Implementation

Finally, coming down to the front-end logic, the author implemented the following logic:

- capturing the *event_id* response received from any *pageview* request and storing that value as a global variable named *lastEventId* in the script
- adding event listeners for *visibilitychange*, *pagehide*, and *beforeunload* events to call an *enrich* function (see Figure 5)
- implementing the *enrich* function that sends an *enrich* event to the server with the *lastEventId* value in a stringified JSON object (see Figure 6)

With these functionalities in place, the *script.beacon.js* script extension was ready. All the changes regarding the front-end script are publicly available for everyone to see in the corresponding pull request in Plausible's Github repository[7].

```
document.addEventListener("visibilitychange", enrich);
document.addEventListener("pagehide", enrich);
window.addEventListener("beforeunload", enrich);
```

Figure 5: Adding event listeners

```
function enrich() {
  if (/hidden|unloaded/.test(document.visibilityState) && lastEventId) {
    navigator.sendBeacon(endpoint, JSON.stringify({n: "enrich", e: lastEventId}))
  }
}
```

Figure 6: Enrich function

3.2 Back-end changes in the Plausible application

Having the functionality for reporting *enrich* events on the front end, the back-end application still needed many modifications to handle the new behavior. This chapter will explain the changes made by the author and the reasoning behind the changes.

3.2.1 ClickHouse and *CollapsingMergeTree* migration

Plausible uses the ClickHouse database management system[8] to allow efficient database communication. According to their website, ClickHouse is 100-1000 times faster than traditional database managemnet systems [16].

[7] https://github.com/plausible/analytics/pull/1679/files#diff-13b3b9d1ff01184b9b792bd47f981dd6571e7f91d15a6c6baf90e658d5ea3ac7

[8] https://clickhouse.com/

Among other table engines supported by ClickHouse, it provides the *MergeTree* and *CollapsingMergeTree* engines which are both used in Plausible.

The *MergeTree* table engine is designed for quickly inserting large amounts of data into a database table [17]. The events table in the Plausible database is a *MergeTree* table.

CollapsingMergeTree inherits all the functionality from *MergeTree*, and in addition, collapses pairs of rows in the background, following a specific algorithm [18]. This gives a very efficient way for updating the table rows. The sessions table in the Plausible database is a *CollapsingMergeTree* table.

In the sessions table, updates are required on every event that belongs to an already existing session. This is because there are fields (e.g., the number of events) that are meant to be overwritten in the session object with every new event. To make use of the *CollapsingMergeTree* engine and update the rows of the sessions table, the application needs to store all session objects in memory that may require a database update. This is because an update in the *CollapsingMergeTree* engine requires the insertion of a cancel row, which deletes the previous instance in the table, and the insertion of a state row to represent the instance with the new duration value. In essence, both the state and the cancel row represent the same session object with all the different fields. This is why the update process requires the use of the entire object and not just the unique identifier of this object.

Tracking *time on page* for every individual page view meant that each event instance had to contain the duration information. Therefore, the author added a new field to the event object called *duration*. To actually make use of this, it had to be possible to update the duration value. Therefore, the author migrated the events table from the *MergeTree* to the *CollapsingMergeTree* table engine, following the example of the sessions table.

3.2.2 Storing recent page view events

To be able to update entries in the events database table, they also had to be stored in the application memory. In terms of database communication, the new way of storing events was expected to be very similar to how it is done with sessions in the *Session.Store* module.

Following the example of *Session.Store*, the author added a similar module called *Event.Store*. On every *pageview* event received, the current solution just constructs the *event* object and inserts it into the database. With the addition of the *Event.Store* module, the application also had to store the constructed *event* object inside the state to make future updates possible.

To avoid overloading the application memory, as with sessions, the time that the events could be kept in memory also had to be feasibly limited. An obvious choice for this limit was 30 minutes, like with sessions.

3.2.3 Updating page view duration with *enrich* events

As explained in Chapter 3.1, *pageview* events were meant to be updated with the *enrich* event. This event represents an update to an already existing *pageview*, containing the

event_id value corresponding to the *pageview* to update. In the new solution, upon every *enrich* event received, the *event_id* value was to be used for finding the corresponding *pageview* from the state of *Event.Store*. Having found the *pageview* event to update, the next step was to calculate a new duration value for the *pageview*.

When a *pageview* event is first constructed into an object, a timestamp field with the value of the current time is added to it. This represents the moment when this *pageview* event was received. Similarly, a timestamp had to be added to the *enrich* event. To calculate the duration for the *pageview*, the new solution had to find the time difference between the moments that the *pageview* and *enrich* events were received, using the two timestamp values. The final step was to update the *pageview* event object in the database and give it a new duration value.

To update a row in the ClickHouse *CollapsingMergeTree* table, a cancel row and a state row have to be inserted. The difference between the two comes from an additional field called the *sign*. The value of this field is -1 for a cancel row and 1 for a state row. The steps to update a *pageview* were the following

- Create a cancel row by adding the *sign* field with the value of -1 to the *pageview* object.
- Insert the cancel row into the database.
- Create a state row by adding the *sign* field with the value of 1 to the *pageview* object and overwrite the duration field of this object with the new duration.
- Insert the state row into the database.

Having completed all four steps with two database insert queries, the update is finished. With one row in the events table before and two more rows added to it with the update, the events table contains three rows representing one event. These three rows are then merged into one single row in the background with the merging algorithm in the *CollapsingMergeTree* table engine.

With all these functionalities developed, the *Event.Store* module was ready to handle updates on *enrich* events. The full code of the *Event.Store* module written by the author can be found in the Plausible Github repository[9].

3.2.4 Updating page view duration with subsequent page views

The new solution for *time on page* tracking was considered an addition and not the new default behavior. Therefore, it had to maintain the functionality of being able to end page views with the following *pageview* events in the same session.

To achieve this, the author added some additional logic for handling *pageview* events, which also included some modifications to the existing *Session.Store* module.

[9] https://github.com/plausible/analytics/blob/3edf8102e63c224d25b33fc81b6b84d4f3203261/lib/plausible/event/store.ex

Upon receiving a POST request representing a *pageview*, it is constructed into an object as the first step. The event object is then passed to the *on_event* function in *Session.Store* module to assign it to a session. This is done by returning the corresponding *session_id* from *Session.Store*. Figure 7 illustrates this information flow in the application with some pseudocode.

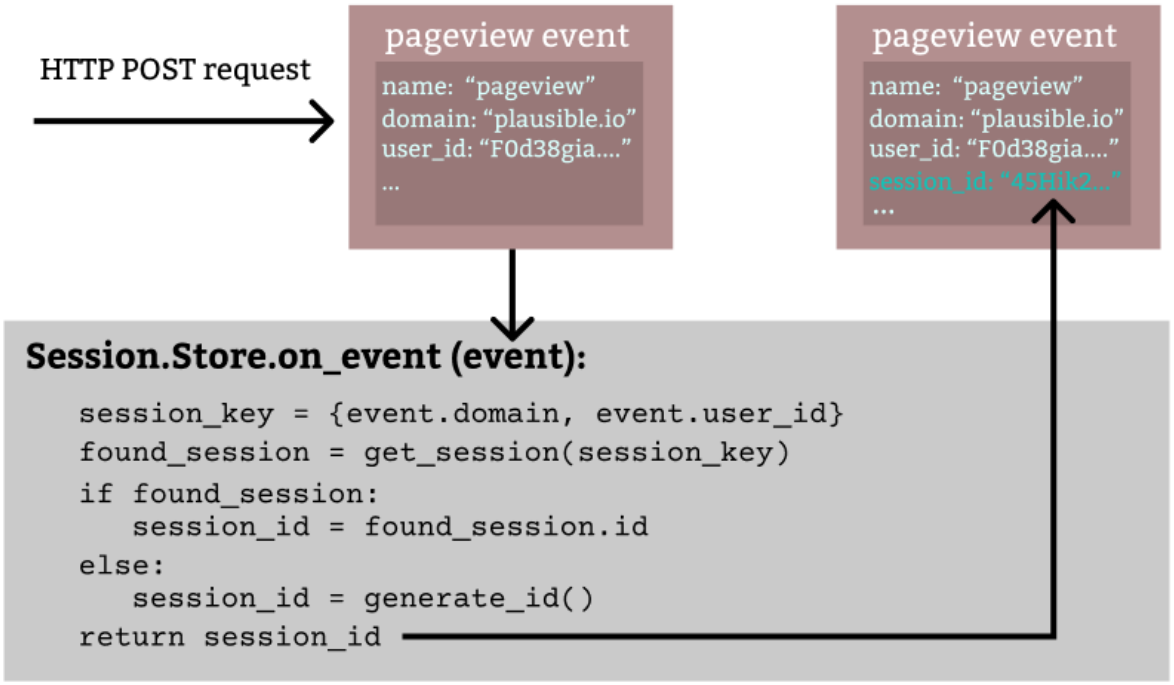

Figure 7: Assigning session ID to an event object

To be able to update the latest *pageview* in a session, the *on_event* function also had to return the *event_id* of the previous event in the session. The author made this possible by adding an additional field to the session object called *last_event_id*. In *Session.Store*, this field was assigned or updated every time a new event was received for a session. This meant that the *event_id* of the previous event was always available from every session and could simply be returned from the *on_event* function.

Having implemented the logic for returning the *last_event_id* value, the *Event.Store* module also had to make use of it. This implementation came down to a single function called *maybe_enrich_previous_session_event*. Based on the *last_event_id* value, it found the corresponding event from the state map and updated it only if its existing duration value was 0. In other words, it had not been enriched.

This behavior maintained the existing functionality of updating page views with subsequent events as a fallback mechanism in case the *enrich* event has failed to update the duration. All

the modifications to the existing *Session.Store* module made by the author can be seen in the corresponding Github pull request[10].

4. Preliminary assessment of results

To get feedback about how the new solution would perform in the real world, it had to be tested on an actual website. For this purpose, the new tracking script was included on the Plausible website itself, where it gathered data throughout March 2022.

The idea was to compare the data reported by the new Beacon API solution with the data of the original solution at the same time. This was not the ideal way of testing the new solution as there was no actual source of truth to compare with, but for the first step of the assessment, it was enough to see what differences it makes in the stats.

To have a convenient way of viewing the two reports side by side, the new tracking solution was set to report events to Plausible's testing domain. This gave the ability to present the new set of stats in the exact same format as in the official stats dashboard.

4.1 Results

In its dashboard, Plausible shows the Top Pages report, which breaks down the usage metrics by page path. The number of visitors and *pageviews*, bounce rate and *time on page* are all surfaced for every page path in the details section of the Top Pages report on the Plausible dashboard. These views can be seen on the publicly available dashboards on both testing[11] and production[12] domains.

To have a nice visual of how the *time on page* value had changed between different pages with the new solution, the author exported the top 100 pages with the most visitors from both testing and production dashboards and created a line chart comparing the *time on page* values. Figure 8 shows the comparison where each data point represents a page path on the Plausible website visited in March 2022.

[10] https://github.com/plausible/analytics/pull/1679/files#diff-d5eb51c240465d3bed8c54b1c86d2ca4770f2a7b2a62478f3198c794b9037a24

[11] https://testing.plausible.io/plausible.io/pages?period=month&date=2022-03-08

[12] https://plausible.io/plausible.io/pages?period=month&date=2022-03-08

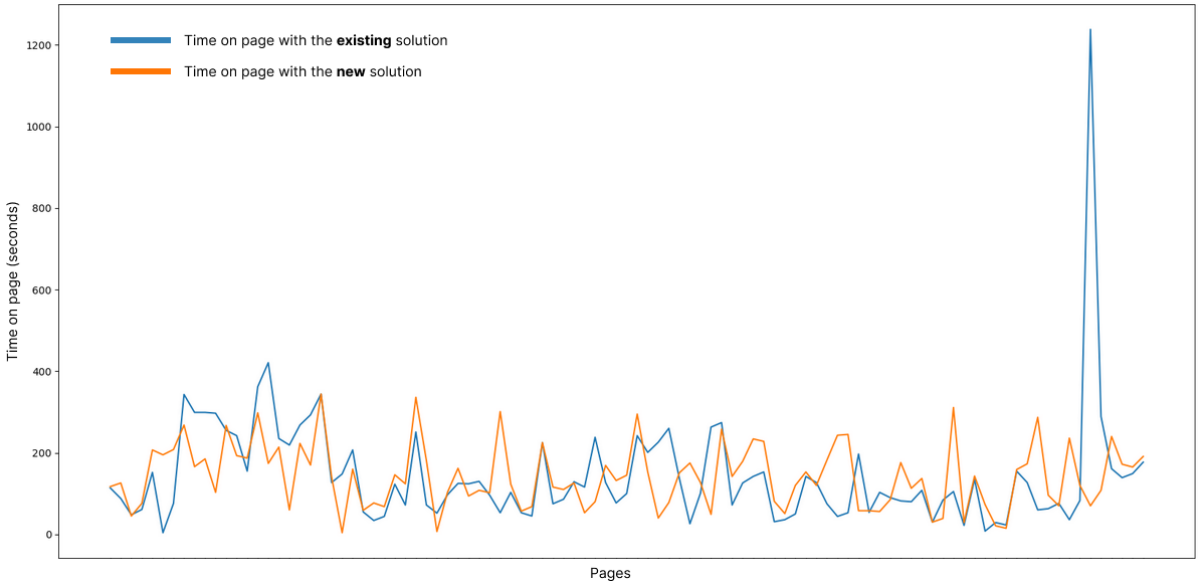

Figure 8: Time on page comparison chart

4.2 Discussion

Due to infrastructure differences between Plausible's testing and production domains, the bounce rate and the total number of visitors and *pageviews* also showed some differences, which are most likely not caused by the new tracking solution. With this noise in the data, the initial assessment did not have any strict expectations or hypotheses. This is also why the assessment of the results in this thesis does not attempt to discover all the duration differences for every page path in the report.

There are future plans to gather data in a way that allows only seeing differences in the tracking solutions without any additional noise, which should reveal some better quality information about the performance of the new solution.

Due to ignoring a significant amount of data, some *time on page* values reported by the existing solution are too extreme and not credible. The best example from the Plausible website is the */sites* page. The purpose of this page is to render a set of links that can be clicked on to navigate to a stats dashboard. As the dashboard pages themselves are not tracked, there will be no subsequent *pageview* events sent until the user navigates back to a tracked page. This means that all the time a user spends viewing their Plausible dashboard is interpreted as viewing the */sites* page.

Consequently, the *time on page* for */sites* tends to show an extreme average of about 20 minutes (see Figure 8) with the existing solution. In reality, the user is only meant to select a site on this page and navigate elsewhere. Therefore, the duration of 70 seconds reported by the new solution seemed credible, and the evaluation result was considered a success.

While the initial assessment did show some expected results and was considered a success, the delivered new solution is still not ready to go to production. The main issue with the new approach is tremendously increasing the server load with the number of requests. For every *pageview* event, there can be several *enrich* events reported from the client-side as a user can

minimize the window or change the browser tab multiple times during one page view. As serving *pageview* and *enrich* events have approximately the same volume in terms of server resources, the server load is expected to increase at least two times with the new approach, likely even more. Along with the plans to gather more reliable testing data to evaluate the accuracy of the new tracking solution, there are also plans for a server load testing phase before releasing this feature to production.

5. Conclusion

The objective of this thesis was to develop a new way of tracking *time on page* in Web Analytics that would also account for the missing values in the generally known session-based *time on page* tracking approach. This new solution was developed and integrated into the existing Plausible Analytics software. The performance of this innovative tracking approach was also measured on the Plausible website itself, where data was gathered throughout March 2022. Due to not having the ideal testing conditions, the assessment of the results in this thesis was only preliminary. Nevertheless, the results still revealed an important expected difference, and the goal of this thesis was met.

While the initial assessment results were satisfactory, the delivered feature is still not ready to be deployed to production because it still needs to be tested and analyzed more accurately. There are future plans to continue with accuracy and server load testing, which will hopefully reveal some more information about the performance of the delivered tracking approach and whether it is feasible to introduce it to Plausible's clientele.

6. References

[1] C. Onwubiko, "Exploring web analytics to enhance cyber situational awareness for the protection of online web services," 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), 2016, pp. 1-8, doi: 10.1109/CyberSecPODS.2016.7502355.

[2] L. Čegan and P. Filip, "Advanced web analytics tool for mouse tracking and real-time data processing," 2017 IEEE 14th International Scientific Conference on Informatics, 2017, pp. 431-435, doi: 10.1109/INFORMATICS.2017.8327288.

[3] Prockiw, C. (2012, May 8). *Why Average Time on Site is a Bad Metric*. Retrieved May 10, 2022, from https://www.vovia.com/blog/sem/why-average-time-on-site-is-a-bad-metric/

[4] Dolan, J. (2022, March 24). *Best Google Analytics Alternative for 2022*. Retrieved May 10, 2022, from https://reflectivedata.com/best-google-analytics-alternative-for-2022/

[5] Edgar, M. *Google Analytics Time on Page*. Retrieved May 10, 2022, from https://www.matthewedgar.net/google-analytics-time-on-page/

[6] Hughes, J. (2022, March 31). *What is Web Analytics? Your 101 on Analytics and How to Get Started*. Retrieved May 10, 2022, from https://themeisle.com/blog/what-is-web-analytics/

[7] Dilmegani, C. (2022, April 4). *In-Depth Guide Into Web Analytics in 2022*. Retrieved May 10, 2022, from https://research.aimultiple.com/web-analytics/

[8] Ahamed, S. (2022, February 7). *Top 10 Web Analytics Tools You Should Try in 2022*. Retrieved May 10, 2022, from https://www.primeone.global/top-10-web-analytics-tools-2022/

[9] Steele, A. (2021, December 22). *What is Time on Page?* Retrieved May 10, 2022, from https://loganix.com/what-is-time-on-page/

[10] Keating, S. (2022, February 18). *What is Time on Page and Why Is It Important?* Retrieved April 29, 2022, from https://jetpack.com/blog/average-time-on-page/

[11] *SPA (Single-page application)*. Retrieved May 10, 2022, from https://developer.mozilla.org/en-US/docs/Glossary/SPA

[12] *Beacon API*. Retrieved March 23, 2022 from https://developer.mozilla.org/en-US/docs/Web/API/Beacon_API

[13] Piirainen, T. (2021). *Beacon API is Broken*. Retrieved April 22, 2022, from https://volument.com/blog/sendbeacon-is-broken

[14] Grigorik, I. (2015, November 20). *Don't lose user and app state, use Page Visibility*. Retrieved April 22, 2022, from https://www.igvita.com/2015/11/20/dont-lose-user-and-app-state-use-page-visibility

[15] Grigorik, I. (2015, November 20). *Don't lose user and app state, use Page Visibility*. Retrieved April 22, 2022, from https://www.igvita.com/posts/15/xlifecycle-events-testing.png.pagespeed.ic.drjAkz8Zye.webp

[16] *Company*. Retrieved May 10, 2022, from https://clickhouse.com/company/

[17] *MergeTree*. Retrieved April 16, 2022, from https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree/#mergetree

[18] *CollapsingMergeTree*. Retrieved April 16, 2022, from https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/collapsingmergetree/#table_engine-collapsingmergetree

Appendix

1. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Robert Joonas,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, Tracking Time on Page in Web Analytics, supervised by Uku Täht and Helle Hein.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Robert Joonas

10/05/2022