WikiStats: RecentChanges Stream Analyzer

1st Jeswanth Yadagani
Electrical Engineering
Columbia University
New York, USA
jy3012@columbia.edu

2nd Sidharth Bambah
Electrical Engineering
Columbia University
New York, USA
sidharth.bambah@columbia.edu

***Abstract*—WikiStats is a robust real-time web application scaled to consume processed streams from Wikipedia's RecentChanges feed. It provides multiple insightful visualizations regarding the fluid nature of Wikipedia's article base.**

The stream processing is implemented using Apache Spark Streaming with Apache Kafka as the message broker used to pass processed messages. Using a variety of windowing, aggregation, batching, and optimization techniques the system accurately produces useful statistics from the data source. It is deployed in the Google Cloud ecosystem. Additionally, a NoSQL database is used to store long-term persistent information.

On the frontend, the ReactJS framework has been used to produce a dynamic dashboard encompassing all of the visualizations, which are primarily made using charting libraries. NodeJS and Flask APIs are implemented to support the frontend operations and interact with the processed streams and stateful data.

***Index Terms*—apache spark, stream processing, wikipedia, apache kafka, nodejs, flask, reactjs, cloud deployment, gcp, aws, heroku**

I. Introduction

Wikipedia is one of the largest multilingual encyclopedia's in the world. It is created and maintained as an open collaboration project by a community of volunteer editors using a wiki-based editing system.

Every day, new articles are being published and existing ones updated leading to a constant information flow throughout Wikipedia. Being such a large community with millions of articles on nearly any subject, it would be quite useful to analyze the true size of Wikipedia's community and the dynamic nature of the content available on the website.

WikiStats monitors the creations and changes of articles by various members of the Wikipedia community. By making use of the RecentChanges datastream, WikiStats uses cloud deployed infrastructure as well as tools, such as Apache Kafka, Apache Spark Streaming, MongoDB, NodeJS, Flask, ReactJS, and ChartJS to present a real-time, interactive analytics dashboard with short-term as well as aggregated change statistics.

II. System Architecture

A. Overview

At a high level, WikiStats uses a four-pronged approach to deliver a robust analytics platform for the recent changes committed to Wikipedia.

At the center, a stream processing backend consumes and processes the changes in Wikipedia. The real-time analyzed data is pushed into a Kafka cluster and the long-term aggregated measurements are stored in a No-SQL database, MongoDB.

This analyzed information is consumed in two web backends and shared with the web frontend on the client side.

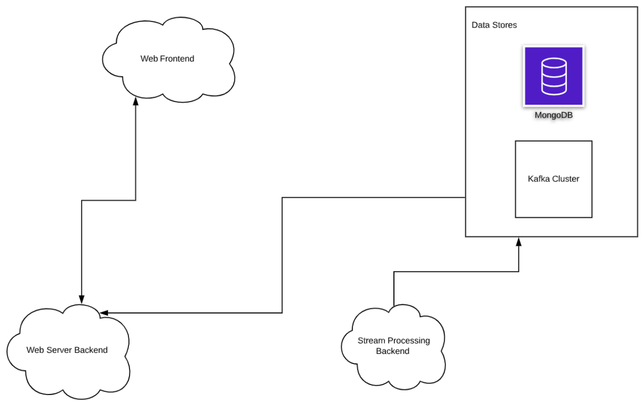


Fig. 1. Overall system architecture for WikiStats.

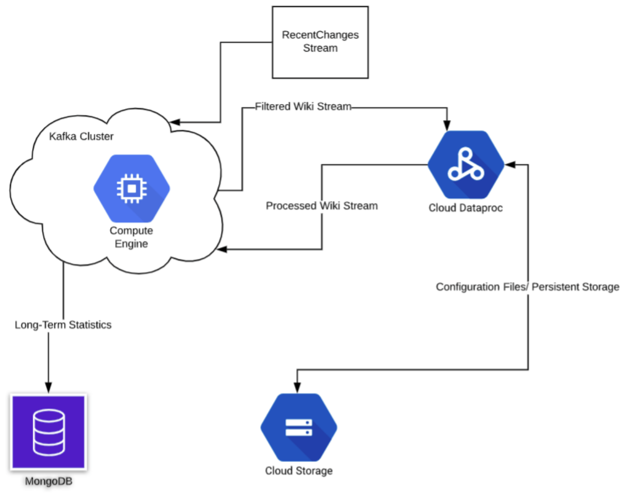


Fig. 2. Stream Processing Architecture

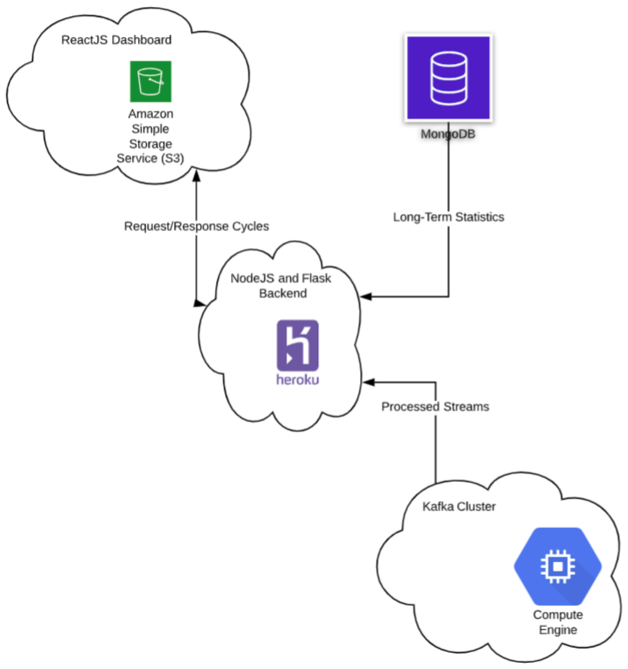


Fig. 3. Web Dashboard Architecture

B. Stream Processing Architecture

Flow of data streams in the system is depicted in Figure 2. Kafka cluster is maintained in a google compute engine with two topics namely 'wikipedia' and 'processed'. Raw stream from the wiki source is written into 'wikipedia' topic on the kafka broker.

Stream analyser spark job file is stored in cloud storage of a dataproc cluster lying at a different location. This job is called to consume the stream from 'wikipedia' topic, extract and process necessary information from the raw stream and finally write the structured information objects into another kafka topic 'processed' sitting on the same kafka broker. Few long term statistics which needs to be stored for analysis are sent to MongoDB, a NoSQL database.

C. Web Architecture

As depicted in Figure 3, WikiStats uses a multifaceted architecture to serve its web dashboard.

At its core, this involves a backend implemented with NodeJS and Flask served using the Heroku platform. These backends connect to a hosted MongoDB document cluster as well as a Kafka cluster implemented in a self-managed Google Compute Engine instance. They establish web endpoints for clients to request data from the processed Wikipedia stream. Additionally, these APIs are constructed to allow requests from any type of web client allowing for easy extensions.

Additionally, the frontend interface is deployed as a static website in an Amazon Simple Storage Service (S3) web hosting bucket. This application can be accessed by any browser on the Web and serves the visualization dashboard for the processed stream.

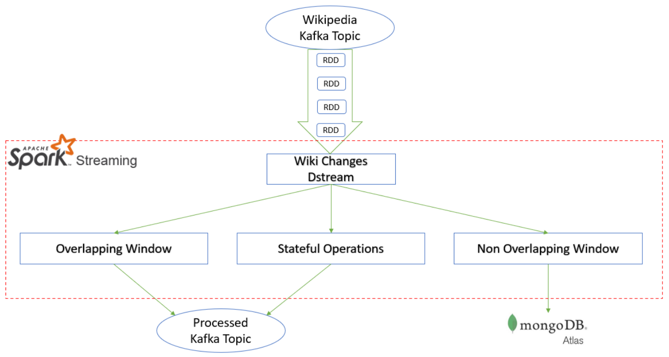


Fig. 4. Spark Streaming Pipeline

III. Wikipedia Data Stream

A. Data Source

WikiStats analyzes the recent changes committed into Wikipedia. The source of data is from WikiMedia, an entity that provides the RecentChanges stream as a publicly available EventSource endpoint.

This stream is quite high volume as it publishes an event on any change. Furthermore, the data is formatted as a JSON object and provides numerous useful fields for analysis. Particularly, the stream contains user, wiki, change type, timestamp, and many more interesting parameters. It can be found at https://stream.wikimedia.org/v2/stream/recentchange [1].

B. Data Consumption

The EventSource RecentChanges stream is consumed using the Python EventSource library, SSEClient. This allows for consumption of server sent events in a generator object. This generator is infinitely looped over and each message is parsed, filtered, and published to a raw Kafka topic.

This consumer job is enabled in the Google Compute Engine machine and is configured to run indefinitely. This is because this raw stream is a dependency for the forthcoming stream processing pipeline as well as the frontend dashboard.

IV. Stream Processing

A. Apache Spark Pipelines

The raw stream produced to the Kafka cluster is consumed and utilized with Apache Spark Streaming. Spark is chosen for its use of discretized streams (D-streams), which provide a high level functional programming API that provides extensive fault tolerance and recovery useful for high volume streams, such as the RecentChanges stream [2].

Furthermore, different insightful operations are performed with varying properties. Batch processing with size of 2 seconds is chosen for spark streaming context.

Initial stream is duplicated and paralleled into three streams as can be seen in Figure 4 in order to accommodate changing needs for drawing insights from data stream. The parallel paths are discussed in detail below.

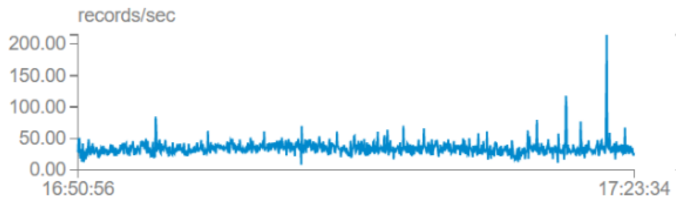


Fig. 5. Input rate from Kafka Direct Stream

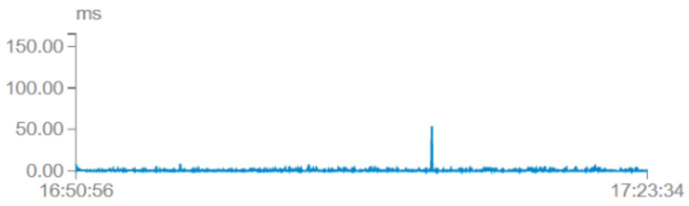


Fig. 6. Scheduling Delay by Spark Job Scheduler

1) Window: A window of duration 1hr is defined and it gathers number of changes registered in that particular duration. This information is stored in MongoDB for long time persistence and offline analysis of data. This kind of data storage can be helpful for offline learning of a model and utilizing it to perform analysis on online mode of a system.

2) Slide Window: A sliding window with a duration of 40 seconds and slide length of 6 seconds is defined to perform sliding stateless operations. Note that both duration and slide length should be a multiple of batch size. This path provides at the moment insight into the changes happening in real-time. Information about number of additions and deletions happening at the very particular instance and top users who are responsible for edits are extracted and displayed as moving forward line charts in the dashboard which moves with a speed of one unit for every 6 seconds. Percentage of changes that are caused due to bots is depicted in a dynamic manner. Popular wiki domains that are actively engaged are extracted and displayed as a pie chart. Information for leader-board of users that aren't bots in the present 40 second window is extracted. All this information is restructured into proper format, bundled as a JSON object and pushed to another kafka topic called as 'processed'.

3) Aggregate: This path in the pipeline is designed to perform stateful operations. Insights like total number of additions and deletions that happen over time, number of distinct users performing changes, top users responsible for overall changes in wiki are collected in this path of pipeline. The collected data is restructured and is written into the same kafka topic 'processed'.

B. Optimization

The processing pipeline is designed with careful consideration of operator costs and optimization strategies. Multiple tests are performed to verify redundancy elimination and optimum operator sequence in the pipeline [3].

C. Metrics

As discussed above, a batch size of 2 seconds is used for spark streaming.

1) Input Rate: Input rate into spark streaming context from kafka cluster is the number of records that are pulled from kafka topic into spark streaming context. It can be observed from Figure 5 that it averages over 34 records per second i.e. 68 records per batch.

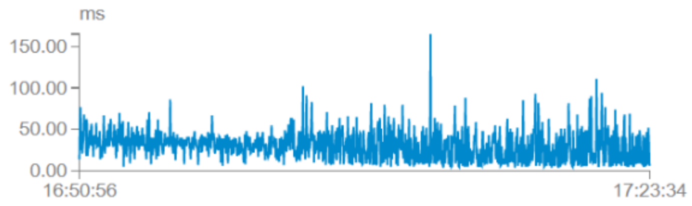


Fig. 7. Processing Time for spark pipeline

2) Scheduling Delay: Scheduling Delay is defined as the time take by streaming scheduler to submit all jobs of a batch. It appears to be very low and close to zero indicating healthy operation of system as can be seen from 6.

3) Processing Time: Processing Time is the time taken to process all jobs of a batch after receiving them from scheduler. It is essentially the major part of delays that are caused in real-time analysis of streams.

4) Total Delay: Total delay is the time taken to handle a batch of data. It is typically sum of scheduling delay and processing time. It appears to be almost similar to processing time from Figure 7 and Figure 8.

D. Outputs

Streaming pipeline has two output paths, each serving their own purpose. Firstly, MongoDB, a NoSQL database, acts as a storage system for information on which analysis can be performed for offline learning of the system. System can be put up for a full length of a month and predictions can be done using ML algorithms on the next famous domain or get insights about where to invest more infrastructure in e.t.c.

Secondly, kafka topic 'processed' is placed on the same broker as the topic 'wikipedia' lies in a google cloud compute engine. This acts as a sink for the real-time processed stream that comes out of spark streaming pipeline. Any consumer that subscribes to this topic will have access to useful real-time insights about the changes that are happening in wiki.

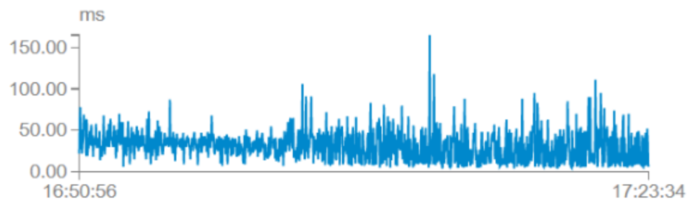
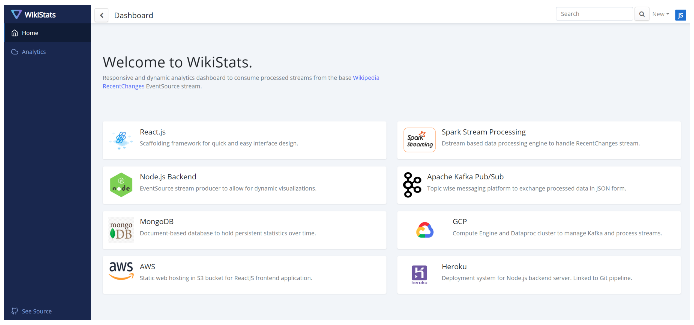


Fig. 8. Total Delay of the



Welcome to WikiStats.

Responsive and dynamic analytics dashboard to consume processed streams from the base Wikipedia RecentChanges EventSource stream.

React.js
Scaffolding framework for quick and easy interface design.

Spark Stream Processing
Dstream based data processing engine to handle RecentChanges stream.

Node.js Backend
EventSource stream producer to allow for dynamic visualizations.

Apache Kafka Pub/Sub
Topic wise messaging platform to exchange processed data in JSON form.

MongoDB
Document-based database to hold persistent statistics over time.

GCP
Compute Engine and Dataproc cluster to manage Kafka and process streams.

AWS
Static web hosting in S3 bucket for ReactJS frontend application.

Heroku
Deployment system for Node.js backend server. Linked to Git pipeline.

Fig. 9. Homepage of the WikiStats platform. Gives general information of technologies used and main data source.

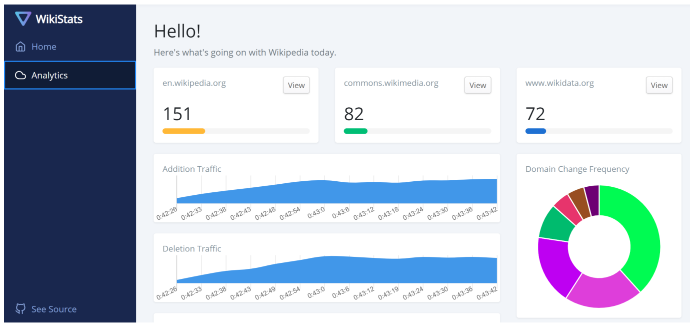


Fig. 10. First set of charts in the analytics dashboard

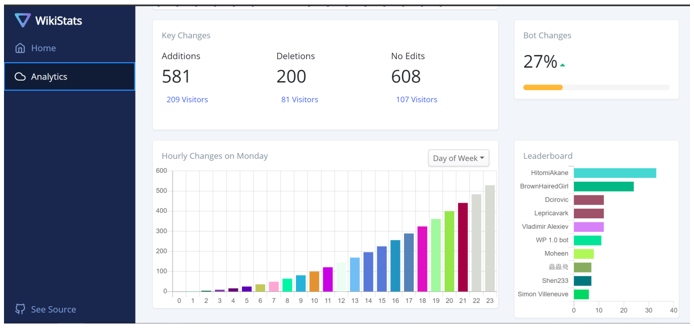


Fig. 11. Second set of charts in the WikiStats dashboard interface.

V. Analytics Dashboard

A. Backend

1) NodeJS API: After the RecentChanges stream is processed in the Spark job, the analyzed message stream is written into the "processed" Kafka topic. In order for the WikiStats dashboard to consume this information, a backend engine is required.

For this, the platform utilizes NodeJS using the Express web server framework. Using the "kafka-node" library, this web server establishes a series of JavaScript callbacks to listen for messages published to the requisite Kafka topic. Once a message is received, it is written to the appropriate route in the form of an EventSource stream.

The EvenSource protocol was chosen because it is unidirectional as opposed to WebSockets, which allow for two-way communication [4]. In essence, the client is not passing any information to the stream processor in this implementation. Rather, all analysis is being completed independently and subsequently read in the NodeJS application. A series of routes have been defined for each distinct visualization; however, this modular approach allows for easy adaptability for numerous different types of clients [5].

2) Flask API: Along with a NodeJS backend, WikiStats utilizes a Python based representational state transfer (RESTful) API in the form of a Flask application. This is used for long-term aggregated statistics.

In the context of this stage of the application, this backend connects to the hosted MongoDB document cluster and performs queries to retrieve documents requested from the client. In particular, the client system sends GET requests to the server with query parameters defining the necessary information and the Flask route returns the result in JSON format.

As with the NodeJS backend, this Flask backend is implemented in such a way so as to support numerous different types of clients and can easily be extended for future uses.

B. Frontend

1) ReactJS: The home page can be seen in Figure 9. This scaffolding and framework is built using an admin panel template in ReactJS.

ReactJS is a JavaScript library that is used to build web interfaces. It was chosen due to its implementation of reusable UI components. Particularly for an analytics dashboard, it is common for certain charts and visualizations to be used multiple times on different datasets. ReactJS is modular and allows for the same implementation to be reused.

Furthermore, ReactJS provides a robust state change system. Particularly, if a components state changes during the web applications life cycle, such as when new statistics become available, only the relevant component is reloaded rather than the entire page. This is convenient in that it allows for real-time visualizations to be easily presented on the screen without too much jitter and lag caused by constant reloading.

2) ChartJS and Bootstrap: The first half of the analytics charts are depicted in Figure 10. Additionally, the second set of dynamically generated charts can be seen in Figure 11. All of these visualizations have been build using a ReactJS wrapper for both ChartJS and Twitter's bootstrap. These libraries provide support for real-time rendering on data change using ReactJS's DOM manipulation functions referencing component mounting and unmounting.

The first visualization depicts the top 3 wikis in the current window. These are presented using a status bar showing the relative percentages of each of their influence on the recent changes. Additionally, there is a link to take a user to the associated wiki.

Next, a line chart has been generated with a filled background to present the addition, deletion, and noedit traffic in the current batch. In this real-time visualization, the older data points are being popped off the chart, while the new ones are added with their corresponding time. This yields an in-depth view of the current state of Wikipedia in terms of the size of it's article database.

Next to these traffic charts, a real-time donut chart has been rendered to present the top 7 wikis in the window in terms of recent changes. This dynamic visualization features a tooltip that provides extra information regarding the top wikis in the current batch. As new data comes in, the proportion of the fields of the donut chart change to reflect the most updated information.

Following this, an aggregate of the changes is shown from the start of stream processing in the "Key Changes" card. As expected, these numbers will continually increase, but it is useful to not the number of distinct visitors responsible for the count of changes. Clicking on this will provide a list of the top ten distinct users along with the number of edits they made to Wikipedia since the start of the stream processor.

Next to this is another card that shows the percentage of all changes that are made from bots, or fake users. This information is provided in the stream itself and can also be parsed from the name of the so-called user who has committed the change. This information is also updated in every window to provide a view of when the most bots are active.

The WikiStats dashboard also features a user leaderboard, which ranks the top non-bot users who commit changes into Wikipedia. This is presented as a horizontal bar-chart with randomly generated colors and shows who the top users are in each time window.

Lastly, a static visualization is used to present long-term aggregated statistics on the RecentChanges stream. Here, a standard bar chart is used to present the number of changes that happen on each hour of the day. The data is taken on a weekly basis and a drop-down menu is presented for the user to select which day of the week's information the chart should reflect.

VI. Cloud Deployment

A. Google Compute Engine

The Google Cloud ecosystem provides virtual machines (VMs) in the form of "compute engines". WikiStats is instantiated on a Linux-based compute engine, which serves as the Apache Kafka broker. A cloud-based virtual system is convenient as it allows variable hardware configurations as workload needs increase. This is a valuable feature since a greater portion of CPU and RAM is used when more messages are being processed and stored in the Kafka topics.

A VM instance was also chosen instead of a managed deployment because it allows for intricate management of Apache Kafka itself. For instance, WikiStats has lowered the retention period for messages in a topic so as to allow for processing of only the most recent changes as opposed to getting stuck in a backlog if too many changes arrive at once. While this is a design choice, the Google Compute engine allows for detailed setup which is not possible in a simplistic, managed environment.

B. Google Dataproc Cluster

The Spark stream processing pipeline for the RecentChanges stream has been instantiated on a Google Dataproc cluster. This allows for easy scalability as the workload increases and provides stability when numerous clients are requesting processed data. The cluster can be configured with multiple worker nodes to distribute operations on the stream.

The Dataproc cluster also provides the yarn task scheduler by default as well as the associated user interfaces for job monitoring. This includes the Spark UI, which generates throughput and latency statistics for currently running jobs. Furthermore, the cluster makes streaming jobs fault-tolerant and allows restarts to be defined on an hourly basis. This means that if, for any reason, the streaming job fails, it can be restarted without any further user configuration. Essentially, this hosted deployment provides stability and production-grade management facilities.

C. Heroku

As mentioned, WikiStats heavily relies on two independent backend web servers implemented with Flask and NodeJS. These servers are also deployed on the cloud to allow for access over the public internet. Additionally, they are written as RESTful APIs to provide access to the processed data from multiple clients.

This deployment is done with the platform as a service (PaaS) provider, Heroku. With Heroku, the code for each of the backends can be bundled with descriptions for their associated dependencies and Heroku will handle the rest. Essentially, all of the required packages and hardware infrastructure is maintained on Heroku's end and only a web endpoint is exposed to be used in client applications. Additionally, this deployment has been coupled with a Git pipeline to allow for easy updates whenever new features are added to either backend.

D. AWS S3

After the ReactJS application was completely built and configured to work with the associated NodeJS and Flask API backends hosted on Heroku, an optimized version was generated using the yarn build tool. Yarn is an alternative package manager to NPM and is built by Facebook [6]. This optimized build contains chunked JavaScript files and static assets linked to a corresponding "index.html" homepage. Additionally, there is further minification of the raw source code that allows for optimal performance in low-end environments, such as mobile web browsers. In minification, multiple JavaScript files are joined together into a single file and then compressed through the removal of whitespace and shortened variable names [6]. All of these optimizations have resulted in a lightweight frontend bundle.

This bundle was collected and pushed into one of Amazon Web Services' Simple Storage Service (S3) buckets. Once uploaded, the bucket policy was set to allow public reads of the data and the static web hosting feature was enabled [7].

This provides an HTTPS encrypted endpoint for end users to access the application.

VII. Conclusion

The system described in this report is stable, robust and deployed in a salable manner utilizing broad spectrum of tools and technologies which include Apache Spark Streaming, Apache Kafka, NodeJS, ReactJS, Flask API, MongoDB, Google Cloud and AWS.

Having different sinks for dynamic and long term data makes system flexible enough to display real time change and perform backend analysis on the stored data, thereby providing wide range of analysis on the recent changes that happen across multiple domains of wikipedia.

Acknowledgment

The authors gratefully acknowledge the contributions of Professor Deepak Turaga of the Electrical Engineering department at Columbia University for his insightful input and analysis leading to the creation of this work.

References

[1] "Wikipedia recentchanges," http://www.wikipedia.org/wiki/Special:Recentchanges, accessed: 2020-03-20.

[2] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*. USENIX, Submitted. [Online]. Available: https://www.usenix.org/conference/hotcloud12/resilient-distributed-streams-efficient-and-fault-tolerant-programming-model

[3] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, Mar. 2014. [Online]. Available: https://doi.org/10.1145/2528412

[4] K. Griffin and C. Flanagan, "Evaluation of asynchronous event mechanisms for browser-based real-time communication integration," in *Technological Developments in Networking, Education and Automation*, K. Elleithy, T. Sobh, M. Iskander, V. Kapila, M. A. Karim, and A. Mahmood, Eds. Dordrecht: Springer Netherlands, 2010, pp. 461–466.

[5] S. Pasquali, *Mastering Node.Js*. Packt Publishing, 2013.

[6] J. Ciliberti, *Creating Modern User Experiences Using React.js and ASP.NET Core*. Berkeley, CA: Apress, 2017, pp. 361–409. [Online]. Available: https://doi.org/10.1007/978-1-4842-0427-6_11

[7] D. Robinson, *Amazon Web Services Made Simple: Learn How Amazon EC2, S3, SimpleDB and SQS Web Services Enables You to Reach Business Goals Faster*. London, GBR: Emereo Pty Ltd, 2008.

[8] J. Kreps, "Kafka : a distributed messaging system for log processing," 2011.

[9] "Spark streaming programming guide." [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[10] V. Subramanian, *MongoDB*. Berkeley, CA: Apress, 2019, pp. 137–169. [Online]. Available: https://doi.org/10.1007/978-1-4842-4391-6_6

[11] K. Relan, *Building REST APIs with Flask: Create Python Web Services with MySQL*. Apress, 2019. [Online]. Available: https://books.google.com/books?id=JUmvDwAAQBAJ

Appendix A
Links

The source code created in the development of this project can be found on GitHub at https://github.com/SidBambah/WikiStats.

The NodeJS backend is available at https://github.com/SidBambah/NodeJS-Backend-for-WikiStats.

The Flask backend can be found at https://github.com/SidBambah/Flask-Backend-for-WikiStats