Real-time user clickstream behavior analysis based on apache storm streaming

Gautam Pal¹ · Katie Atkinson¹ · Gangmin Li²

Accepted: 16 October 2021
© The Author(s) 2021

Abstract

This paper presents an approach to analyzing consumers' e-commerce site usage and browsing motifs through pattern mining and surfing behavior. User-generated clickstream is first stored in a client site browser. We build an ingestion pipeline to capture the high-velocity data stream from a client-side browser through Apache Storm, Kafka, and Cassandra. Given the consumer's usage pattern, we uncover the user's browsing intent through *n-grams* and *Collocation* methods. An innovative clustering technique is constructed through the Expectation-Maximization algorithm with Gaussian Mixture Model. We discuss a framework for predicting a user's clicks based on the past click sequences through *higher order Markov Chains*. We developed our model on top of a big data Lambda Architecture which combines high throughput Hadoop batch setup with low latency real-time framework over a large distributed cluster. Based on this approach, we developed an experimental setup for an optimized Storm topology and enhanced Cassandra database latency to achieve real-time responses. The theoretical claims are corroborated with several evaluations in Microsoft Azure HDInsight Apache Storm deployment and in the Datastax distribution of Cassandra. The paper demonstrates that the proposed techniques help user experience optimization, building recently viewed products list, market-driven analyses, and allocation of website resources.

Keywords Clickstream analytics · Real-time big data analytics · Real-time data ingestion · Apache storm · Cassandra · Datastax

✉ Gautam Pal
gautam.pal@liverpool.ac.uk

¹ Department of Computer Science, The University of Liverpool, Liverpool L69 7ZX, UK

² School of Computer Science & Technology, University of Bedfordshire, Luton LU1 3JU, UK

1 Introduction

E-Commerce sites track the consumers' browsing patterns simultaneously in real-time and in batch mode. Mining *browsing motifs* to display *personalized recommendations* and near-real-time tracking of *recently viewed products* greatly enhances overall user experience and helps generate revenue. Users' *click to view* items pile up to an enormous data volume compared to a tiny percentage of clicks converted to final checkouts. Near real-time processing of a large data pool for creating unique personalized, contextual experiences need quick analysis of the inflow of data before it is even stored in the database of records. Coupled with zero tolerance for data loss, the challenge gets even more daunting. In a Big Data real-time setting, instead of waiting for data to be gathered in its totality at a long periodic batch interval, the streaming analysis leads us to detect patterns and make informed conclusions based on them as data start arriving. Big Data Streaming is the data processing paradigm designed with infinite datasets in mind. Introduced as a new category of open source project, a scalable stream processing paradigm, by Nathan Marz, creator for Apache storm [45] while developing an ingestion pipeline for Twitter. Apache Storm is a popular real-time distributed processing framework allowing users in-flight processing on the inflow of data before it is even stored in the database.

Our work is based on analyzing and predicting user *clicks* in real-time through a machine learning approach on a huge volume of heterogeneous data pool that requires a novel deployment model for stream processing frameworks and NoSQL datastore doing trade-off with Consistency, Availability, and Partition Tolerance (CAP) and also between throughput, latency, and correctness. This work describes near real-time data storage and processing approaches to analyze streams of click data based on Apache Storm and Cassandra NoSQL datastore to bring insights into consumers' browsing motifs and build the recently viewed products list at near equal to the pace the user continues to click on links.

1.1 Clickstream analysis application scenarios

This subsection describes the motivation *usefulness* of the clickstream analysis in practice.

1. **User Experience Optimization:** Clickstream helps understand the user's mindset and improve sales through better engagement. E-commerce sites collect data tracing users' trails to analyze the pages they are visiting most and in which order. Web traffic analysis reveals the path users follow while browsing. E-commerce marketers can improve on the overall users' experience by getting insights about the responsiveness of the website, how frequently users click the *back button* or the *stop button*, and the amount of data transmitted before users move on to the next page. Marketers can quantify the effectiveness of the sales volume. The analysis results show items added to the cart and removed and items purchased together. Powered by clickstream data analysis insights and market trends, mar-

keters improve the click path by optimizing the site design and effectively reducing *bounce ratio* (items added to cart but not checked out) and improve *view to purchase* conversions.

2. **Building Recently Viewed Products List:** The recently viewed products list enhances the overall shopping experience by reminding users of products they have viewed before. Processing data near-real-time is the main technical challenge on a large volume of the data pool.
3. **Market Driven Analysis:** Basket analysis produces insights into the aggregate behaviour of customer buying patterns. Similar to car drivers who can use different routes to reach the same destination, analysis of a user basket uncovers the common interest across consumers, the common path users take to conclude a purchase. This leads the way to find out the most effective path a buyer takes to search for and buy a product.
4. **Analyzing the Next Best Product:** Next Best Product Analysis (NBP) is essential for marketers to predict the next purchase by a customer. NBP analysis discovers customer buying patterns to list the items consumers tend to buy together. For instance, a customer who buys nuts *also-bought* bolts together. Once the *also-viewed* and *also purchased* behaviour is learned for a collective pool of historical users, a new customer can be offered a product together with another product he already bought. This helps to generate major revenue for the site since the analysis shows that consumers tend to buy more often from suggestions and recommendations than by their own search.
5. **Allocation of Website Resources:** Clickstream analysis uncovers the key browsing patterns which are used to distribute the resources (hardware and development time) to focus areas.
6. **Segmenting consumers at a Micro Level:** High impact personalized recommendation is achieved through micro-level customer segmentation using clickstream data clustering. Customers are grouped based on buying pattern and average cart value, which helps to provide a targeted recommendation that users are most likely to buy.

1.2 Ethical consideration

E-marketers benefits by employing users' clickstream trail for purchase prediction and recommendation but not without the cost of data privacy. The trade-off between the privacy of sensitive customer information and business value sparked several ethical concerns in e-commerce [7]. However, clickstream data used for this paper does not contain any Personally Identifiable Information (PII) such as credit card information or addresses. Users' click paths are anonymously stored in the Cassandra database.

2 Summary of contributions

The primary contributions of the paper are the new methods for stream data ingestion, in-flight processing, and distributed storage using a case study through clickstream data analysis as summarized below:

2.1 A case study through clickstream data analysis

1. The case study is based on the big data *Lambda Architecture (LA)* [18], which combines both batch and stream processing approaches. LA is used to analyze users' click data on e-commerce sites. In the real-time setting, the model builds the *recently viewed products* list, which a user previously viewed. At the batch setting, the model simultaneously analyzes *motifs of users* for computation of *personalized recommendations*.
2. This paper introduces novel techniques in clickstream data analytics to unleash key customer journeys through pattern mining using the n-grams and Student T-Test, which distinguishes between regular patterns and special sequences [40, 48]. A model is proposed to predict users' transition from one state to another based on the higher-order Markov chains.
3. Clustering clickstream sequences help generate customer segments and build personalized recommendations. A Custom-built model consolidates both batch and real-time data pools. First, the initial cluster is formed through the batch setting. Thereafter, at a streaming setting, a variation of the Expectation-Maximization algorithm with Gaussian Mixture Model maps each user click (stream event) to one of the clusters at every streaming window interval. A dimension reduction technique reduces training time significantly.
4. A model for the Apache Storm topology is presented serving near real-time responses.

2.2 Distributed storage layer

- The paper introduces a streaming data storage mechanism through Cassandra datastore to support a low-latency, highly available stream processing architecture.
- Proposed theorems describe trade-off between *low-latency* and *high-accuracy*.
- The paper provides unique insight into optimizing *Cassandra* database on a multi data centre setup for near Real-Time Responses.
- Stress test results are provided on the *Datastax Enterprise* (DSE) distribution of Cassandra.
- Experimental results are presented for clickstream data flow from *Kafka* to *Cassandra*.

2.3 Finding the research gaps

This works positions to fill the current research gap of an efficient big data real-time processing framework for e-commerce clickstream data analysis. The Hadoop and its de facto MapReduce processing mechanism are designed for batch setup. While MapReduce is capable of providing more comprehensive and accurate insight on the historical dataset, it suffers from a number of challenges like high latency, larger clusters, and bigger storage requirements. Traditional real-time setups, on the other hand, lack reliability, accuracy and are ineffective at cold start situations.

We address the existing limitations with a simultaneous mix processing approach through big data Lambda architecture. The model allows the stream processing engine to avoid cold start situations by initiating from an offset provided by the batch engine. Parallel processing through batch and stream modules ensures both low latency and high throughput requirements are served seamlessly.

2.4 Organization of the paper

This paper begins with a review of several existing big data real-time processing frameworks like Apache Kafka, Adobe Analytics, and Apache Storm and in Sect. 3. Lambda Architecture which is used for data ingestion and processing, is described in Sect. 4.1. In subsequent sections, several data ingestion mechanisms are implemented through a case study of clickstream data capture and in-flight processing. Section 1.1 provides the usefulness of the proposed framework in the clickstream analysis. The Sects. 5.1, 5.2) presents data retrieval techniques in real-time through users' clicks in a browser, mining data-in-transit before storing into big data storage. Section 5.3 outlines the strategy to minimize database latency, trivial to achieving real-time turnaround time. A case study for real-time clickstream data analysis using Storm is presented in Sect. 5.4. The application is hosted in the Microsoft Azure Cloud environment (Sect. 5.5).

3 Literature review

An efficient ingestion strategy involves extracting information from different data sources (as shown in Fig. 1) and carrying it over a network in a fault-tolerant and distributed manner. For instance, *Flume* [8, 19, 20] uses *Netcat* or *Spoolir* abstractions to read data from sources like *web services* or a directory that is streamed with new data. See Fig. 2 for a list of *abstractions* that *Flume* uses to read data from various sources. Ingested data is stored into a sink: either Hadoop Distributed File System (HDFS) or a NoSQL database. Between source and sink, data can be stored into a *passive store* for a configurable amount of time or size when a sink is busy or unavailable for fault-tolerance. See Fig. 3 for an end-to-end flow of data ingestion through *Flume*.

Fig. 1 The Figure shows different data sources supported by Flume. Flume moves data through memory or disk to finally store into a HDFS based storage location

Fig. 2 The Figure shows different data sources supported by Flume. Flume moves data through memory or disk to finally store into a HDFS based storage location

Fig. 3 End-to-end data flow using Flume. Stream is initially collected from server logs, data transits through memory channel and finally stored into HDFS in a multi-node Hadoop cluster

Apache Kafka [11, 14, 44] is a fast, scalable, durable and fault-tolerant publish-subscribe messaging system for data ingestion. It replaces traditional message brokers like Rabbit MQ, IBM-MQ because of higher throughput, reliability, and replication capabilities. It was originally developed in LinkedIn and subsequently open-sourced in early 2011. Components of the Kafka cluster include the following— (i) **Broker** is the actual Kafka process. A Kafka cluster can have one or more brokers. (ii) **Topic** are feed names to which messages are published by the producers. It can have partitions to increase the degree of parallelism. (iii) **Producer** is an application that publishes data to a topic and in a particular partition within a topic. (iv) **Consumer** is an application that subscribes to a given topic and consumes the feed of published messages. (v) **Zookeeper** is the coordination interface between the Kafka broker and consumers. For each Topic, the Kafka cluster maintains a *partitioned* log. Each *partition* is an ordered, immutable sequence of messages that is continually appended to a commit log. The messages in a partition are each assigned a sequential id number, called an offset, which uniquely identifies each message within the partition. The Kafka cluster retains all published messages whether or not they have been consumed for a configurable period of time or size. Kafka uses partition replication to ensure resilience, as shown in Fig. 4. Different *consumer groups* and *offset management* in Kafka ensure fault-tolerant distributed computing. If multiple consumers which are part of the same consumer group are hooked to the same Topic, then a message is delivered to only one consumer within the group. If a message needs to be consumed by multiple consumers, then the consumers have to be part of different consumer groups. Since the same message can be consumed by multiple consumers in a consumer group, the consumer's responsibility is to maintain a log of what has been consumed so far. The consumer maintains an offset which is a position in the log till which messages have been consumed. In

Fig. 4 Kafka *topic* is partitioned into a *leader* (main copy) of the data, and an *In Sync Replica (ISR)*. *Leader* and *ISR* are hosted into different physical nodes in a cluster where a Kafka broker is running. Assume, that the partition size is 100 mb, then for 200 mb of data load, each of the *lead partition* will store 100 mb data. Replica of *partition 1* in *broker 1* is stored into the *broker 2* as an *ISR*. Similarly, replica of *partition 2* in *broker 2* is stored into the *broker 1* as an *ISR*

earlier versions of Kafka, the offsets were maintained with Zookeeper. From V0.8.1 onwards, the offsets are maintained in a separate topic created by Kafka.

A Kafka cluster can be configured in the following topologies: single node single broker cluster, a single node multiple brokers cluster, and multiple nodes multiple brokers cluster.

Apache Kafka witnessed a wide industry adaptation empowered with real-time distributed message queue for the stream processing frameworks [10]. Ingesting a large high-velocity volume of data requires fast, fault-tolerant distributed pipelines. Apache Kafka, as a distributed client-server-oriented publisher-subscriber messaging system, replaced many traditional message queue systems like Rabbit MQ, IBM MQ because of its higher throughput, reliability, and replication capability [22].

Adobe Analytics is an industry-leading tool allowing stream data integration. Raw clickstream is accumulated from mobile apps, and websites using web service APIs [1, 23]. Adobe Analytics analyzes raw clickstream after data is stored and pre-processed in the Adobe warehouse. Adobe Analytics, however, lacks the real-time feature and works on the batch manner for a recurrent hourly or daily schedule.

Several open-source and enterprise stream processing solutions have emerged in the recent past, such as Spark Streaming, Storm, Flink, Beam, which are widely explored in web traffic and clickstream analytics [13, 17, 25].

Apache Storm is an industry-leading distributed client-server stream processing framework, written primarily in the Clojure programming language, first developed at the BackType company [24, 41]. The project was then open-sourced to Apache after being acquired by Twitter. An optimization technique for Storm's default scheduler to perform fair load balancing, especially in heterogeneous Storm clusters setup, was proposed in [12] [46] to overcome the limitations of the default round-robin scheduling, which attempts to overcome the disparity between resource requirements and availability that can lead to poor performance and resource wastage. *Stream groupings* in Storm decides workload allocation among different bolts. There are *eight* stream groupings built into Storm. The *shuffle grouping* is the default scheduler in Storm. For the production deployments, an optimized *stream groupings* is recommended, replacing the default scheduler [42]. To support design justifications in selecting specific data mining approaches and visual analytics in clickstream analysis using multi-levels granularity, in a recent work [26], researchers have proposed visualization strategies for the sequential forms, raw sequences, and design collaboration methods.

Existing research on clickstream data is not based on the public cloud deployments and can not take advantage of building on top of the state-of-the-art cloud offerings. The proposed methods, in comparison, are hosted on the Apache Storm cluster in Microsoft Azure HDInsight Cloud environment [6, 9]. As a result of cloud capabilities, the proposed model is robust in failover with no impact on cluster downtime when a worker node is down. The model can build a streaming pipeline with a host of Azure Services and configurations that are optimized for a production-ready deployment.

At the implementation level, several application areas were investigated for Storm, like the multi-sensor data fusion principle and a real-time network intrusion detection system [29]. Using the UML profile of Storm applications, performance

As shown in Fig. 5, Apache Storm stream processing pipeline cleanses and transforms data to store into Cassandra database. A recently viewed list can be built by a low-latency Cassandra query as users continue to click on different links on an e-commerce portal. Java Spring REST APIs return JSON responses to the user interface to display the *recently viewed* item list. At the same time, a simultaneous data pipeline stores the data into HDFS, and a periodic batch job performs data mining tasks to extract user *motifs* through *n-grams* and build a recommendation service by *Collaborative Filtering* [35]. Section 5.2 elaborates on the case study further.

To generalize, Lambda architecture provided a *simultaneous* and *mix processing* environment for data ingested to the system and dispatched to both the batch layer and the stream layer. The stream layer serves only low-latency queries. Data is further merged for the type of queries that require comprehensive historical data.

5 Methodology

5.1 Retrieving ingested clickstream data

Clickstream is recordings of users' click navigation trail while surfing a website [16]. The user action is captured in the client-side browser. Therefore, clickstream data is the URL generated when each user clicks on items on the site. For instance, in an online shopping portal, clickstream data may look like this: <http://smartbuy.com:8080/electronics/products?userID=id3&productName=iPhone10&price=500&location=london>. Each context parameter like *product name*, *price* and *location* is appended to each clickstream events. Each time users open a new session, a new user context is captured. The context is a uniquely derived entity built from the session object created at the application server and JavaScript layer. A context ID is appended to each user's click data. The context ID is used for identifying each user uniquely and tracking their previous browsing patterns based on the system IP users browsed from. Thus, the website need not rely on users to explicitly log in to track their identity.

5.1.1 Identifying unique users through context ID

E-commerce users can see their recently viewed products list even when they are not logged in. So, tracking each user's past interactions for a non-logged-in user requires identifying a unique user session from the application server's session ID. Every user of a site is linked with an application-server-generated *javax.servlet.http.HttpSession* object, which is persisted and fetched later for session information about that user. Consider that clicks from each unique IP belong to a unique user for each session. A *recently viewed items* list is built for a user based on a concept that essentially combines all the user's past sessions into one single list. This approach has the limitations of not being able to capture events correctly when multiple users are accessing the same IP or the same user browsing from different IPs. The limitation is avoidable, considering only logged-in users or using the session id. The final order of the recently viewed products list, sequence, and pattern extracted from the

context ID is the key determining factor for analyzing *motifs of users* and computing *personalized recommendations*.

Before developing the method for analyzing clickstream sequences and understanding users' motifs, a few concepts and measuring techniques for quantifying user behavioural patterns and motifs are defined first. The topographic constructs of clickstream sequences are described, and an estimate of users' interests like the support of a sequence, common patterns, and motifs are formally defined.

Definition 1 (*Set of Unique Sequences*) A set of ordered and unique sequences S_i represents user activities under a single browsing session persisted in a database of tables.

Definition 2 (*Subsequence*) A: $a_1, a_2, \dots, a_n$ is a subsequence of B: $b_1, b_2, \dots, b_n$ if and only if there exists $k_1, k_2, \dots, k_m$ such that $1 \leq k_1 < k_2 < \dots < k_m \leq n$ and $a_1 \subseteq b_{k_1}, a_2 \subseteq b_{k_2}, \dots, a_m \subseteq b_{k_m}$

Definition 3 (*Support of a Sequence*) Support of a sequence S_i is the count of sequences in database D which has S_i as a subsequence. If the count of matches (support) for S_i in the database is more than a defined threshold, then S_i is called a frequent sequence [37].

Definition 4 (*Set of Unique Events*) A set of unique ordered click events E_i combines to form a sequence S_i , such that $\sum_{i=1}^n E_i = S_i$

Definition 5 (*Set of Common Patterns*) A set of common patterns P_i is identified as set of multiple sequences or subset of a sequence. Hence, a pattern P_i is defined as $P_i \subseteq \sum_{i=1}^n S_i$ or $S_i \subseteq \sum_{i=1}^n P_i$

Definition 6 (*Finding the Motifs*) A user's motif (M_k) to use the site is a combination of patterns represented as *n-grams* with n representing the count of patterns such that $\sum_{i=1}^n P_i = M_k$.

The following section explains the steps for understanding user motifs through pattern mining. Hadoop ecosystem tools such as Hive have good support for native n-grams APIs, which are used to highlight commonly occurring sequences that appear more than a defined threshold as defined in Definition 3 and 5. Understanding user behaviour through n-grams is a high latency task handled through periodic batch jobs.

5.1.2 Browsing hierarchy

Users' browsing patterns are grouped into categories based on the browsing path. The parent category is divided into several subcategories. For example, the category *electronics* is sub-categorized as *television* and further sub-categorized as *3D television*. A custom JavaScript is added to the *header* of each UI JSP page, capturing

Table 1 Clickstream sequence Table

Session 1 User A	Session 2 User A	Session 1 User B	Session 1 User C
http://a	http://a	http://e	http://a
http://a/b	http://a/c	http://e/f	http://a/b
http://a	http://a/c/d		

Table 2 Item sequence Table 1

User ID	Sequence ID	Event Time Stamp	Leaf Element of the Path
U_1	1	10	a
U_1	1	20	b
U_1	2	30	b
U_1	2	40	c
U_1	2	50	d
U_2	3	20	e
U_2	3	40	f
U_3	1	10	a
U_3	1	20	b

the users' navigation patterns in terms of each click. The navigation trail creates the browsing path for each user for each session. For instance, users first open the home page *http://a* then navigate to the subcategory *b* at the URL *http://a/b* then come back to the home page again. The clickstream pattern for the user *U* is: a, b, a. Each time the user navigates to a new category and subcategory to the leaf of the tree, a new click event is captured. If a user closes the session by closing the browser and opening the home page again, the new session sequences are not included in the old pattern Items browsed under the same browsing session are the key to understanding the user motifs and building the recommendations.

Total time spent per subcategory: Along with browsing frequencies, total time spent per user per subcategory on each session reveals user browsing patterns. Time spent for each item under a subcategory are added. A parent category (like *electronics*) can be too generic to be meaningful for user segmentation. Therefore, sub-category (like *television*) is used for computing total time spent. Let d_i^j be the time spent on a subcategory c_j by a user u_i then $d_i^j = \sum_{i,j=1}^n \text{duration}(u_i, c_j)$

The sequence of events captured by the system is as follows:

Per user per element frequency is computed using Equation 1

$$\text{freq}_u^i = \text{count}(\text{user}_u, \text{item}_i) \quad (1)$$

Frequency of co-occured items i and j is given by

$$\text{freq}^{ij} = \text{count}(\text{user}_u, \text{item}_i, \text{item}_j) \quad (2)$$

Table 3 Item sequence Table 2

User	Session ID	Elements	Size	Root Cat-egory
U_1	1	[a,b]	2	a
U_1	2	[a,c,d]	3	a
U_2	3	[e,f]	2	e
U_3	4	[a,b]	2	a

Table 4 Co-occurrence Matrix

Item Viewed	Item Also Viewed	Count
a	b	2
b	c	1
c	d	1
e	f	1

The relative frequency for item i and user u =

$$\frac{\text{count of visits to item } i \text{ by user } u}{\text{Total count of visits by user } u} \quad (3)$$

$$\text{RelativeFreq}_u^i = \frac{\text{count}(\text{user}_u^i)}{\text{count}(\text{user}_u)}$$

A co-occurrence matrix M_{ij} is built between two items (i and j) across all of the user base. The matrix is the aggregated count of items *co-viewed* under the same browsing session (Tables 3, 4).

$$M_{ij} = \text{count}(\text{item}_i, \text{item}_j) \quad (4)$$

5.2 Understanding users' Motif through pattern mining

The sequence of events that co-occurred more than average reveals an important user trait. For understanding browsing pattern, *n-grams* are computed with n ranges between 2 to 7. *n-gram* captures the frequency of a contiguous sequence of n elements from a corpus of text. The sequences are sorted based on the pattern length and occurrence frequency to reveal the most popular webpages and the sequences a user is reaching the final webpages. Apache Hive APIs [2] can be used to compute *n-grams* and find the most commonly occurring sequences and their counts.

n-grams measure the popularity of a webpage sequence but do not determine if the *mean* of two sets of data are significantly different from one another. Consider a table of data with two columns as browsing sequence (category and subcategory) and an average price of a subcategory; divide the table into two, one with a price of more than \$25 and the other with a price less than or equal to \$25. To estimate the

difference in the data distribution of the two price groups *Independent two-sample Student's t-test* is used as a *Collocation* method [28]. Experiments use the production dataset obtained from data.world [3, 31] consists of data from Amazon.com fashion products and Indian e-commerce major Flipkart's product details. Table 5 shows the results for the t-test with *Amazon* and the *Flipcart dataset*. *Flipcart dataset* contains raw clickstream events in the *product_url* column along with price and other information. Amazon dataset captures the clickstream for products *also bought* along with the name of the product *bought*. *Amazon dataset* is compared for two distributions separated based on the price (price more or less than \$25). *Flipcart dataset* is split into two and compared for browsing patterns in the odd and the even months

The return values [h,p] signify if two distributions are substantially dissimilar from each other at a 1% significance level. With $h = 1$ in the *Amazon* dataset, the browsing pattern for two distributions of different price ranges but with similar click sequences are significantly different. On the other hand, with the *Flipkart* dataset, considering the click sequences, clickstream events for odd and even months are not much different.

5.2.1 Clustering clickstream sequences

Users take different paths while browsing the site. Permutations and combinations of click paths make clickstream analysis and prediction data intensive. Grouping the clickstream data helps create customer segments with users of similar interests. Clustering can be used for future click predictions for similar users.

Clickstream is clustered based upon user browsing sequences under a session, browsing frequencies on the same path, and the duration spent on each subcategory. Consider two clustering approaches for customer segmentation and prediction: a hybrid model of K-Means clustering and the Expectation-Maximization algorithm with Gaussian Mixture Model. Initial clusters are created by Apache Spark APIs K-means clustering under the batch settings for the historical dataset as a one-time job. Thereafter, at each streaming window interval, Apache Spark Streaming K-means APIs map each user click event to one of the clusters and update the cluster incrementally at a near-real-time setting [4, 33, 34]. Cumulative hybrid learning between batch and stream is described in our previous work [32]. However, K-means *hard assigns* an event to a specific cluster, limiting the probability of customers belonging to multiple user segments with varying degrees of possibility. Therefore, we can take recourse to a variation of the Expectation-Maximization (EM) algorithm with the Gaussian Mixture Model (GMM) at the streaming setting.

Table 5 Student's T-Test

Returned parameters	Amazon dataset	Flipkart dataset
h-value	1	0
p-value	$3.751e^{-07}$	0.1098

EM *soft allocates* events to distributions with a probability of any event belonging to a distribution mean, resulting in users falling into multiple customer segments with varied degrees of association. The predicted cluster is used to *suggesting* the user a set of items under the same browsing session. For instance, if the user views items a , b , c under a browsing session, then based on the sequences on the existing clusters that are similar to the user's browsing pattern, items d , e , f can be suggested to a user. The challenge is, since users expect to see the suggested items under the same browsing session, as they continue to click different items, the computation needs to return near-real-time results. As the browsing data volume is large on even a relatively moderate user base, a real-time big data ingestion and computation framework are essential.

5.2.2 Expectation maximization (EM) algorithm with Gaussian mixture model

Let us assume c_{ui}^t is the interaction between the user u and item i at time t . Then c_{ui}^t is given by:

$$c_{ui}^t = \omega p_{ui}^t + p_{ui}^t \quad (5)$$

where p_{ui}^t is the feature vector for the pair of user u and item i in a real-time setting. Vector p_{ui}^t is learnt through the historical batch dataset to provide an offset (or starting point) for the streaming process. p_{ui}^t is generally high dimensional and typically sparse. Standard learning methods [21, 36] are used to overcome the sparsity problem. Let us assume p_{ui}^t belongs to a $m \times n$ dimensional projection matrix $M_{m \times n}$. M is high dimensional and largely remains unchanged with very few cells requiring update at real-time. That is, $p_{ui}^t = M\delta_{ui}^t$. Only δ_{ui}^t is learnt at real-time which is not high-dimensional and the online model converges much faster. ω is the data sample row vector representing the weight of each of the events summing to *one*. The feature vector p_{ui}^t between user u and item i is a data frame with *three* variables as browsing sequence, frequency and duration in each sequence per customer. See an example in Table 6.

The proportion of weight is pre-estimated and defined in the weight vector ω . In the Table 6, A is the parent category (example *electronics*) and given less weight than the more specific sub-category (example *television*) which reveals more about customer browsing intent. C and D are the leaf nodes representing a specific

Table 6 User Browsing pattern

Weight	Customer ID	Browsing Element	Browsing Frequency	Browsing Duration
0.1	C1	A	10	20
0.6	C1	A-B	8	90
0.3	C1	A-B-C	400	120
0.1	C2	A	12	20
0.6	C2	A-B	15	20

item with brand and model number (example: Samsung *television* model number UE50RU7100). Since Leaf nodes are too specific, they are given less weight to avoid model overfitting.

The E-step: The expectation step aims to compute the *responsibility* for each data point for each cluster distribution. Let us create 50 customer segmentations within *one million* records in a *1million*×50 matrix. Expectation-step finds each data point's level of association to each distribution is given by Equation 6:

$$a_{ic} = \frac{\omega_c P(x_i | \mu_c, \sigma_c)}{\sum_{j \in (0, n)} \omega_n P(x_i | \mu_n, \sigma_n)} \quad (6)$$

where a_{ic} is association of the event x_i in terms of Gaussian distribution c . Total number of possible Gaussian distributions is n . The numerator represents the probability of event x_i for Gaussian distribution c multiplied by the weighting factor for each distribution. The denominator represents the sum of probabilities for n Gaussians for the event x_i .

The M-step: With the degree of association value for each event for each Gaussian, we can improve the mean and standard deviation using the association values. The mean is calculated as

$$\mu_c = \frac{\sum_{i=1}^n a_{ic} x_i}{\sum_{i=1}^n a_{ic}} \quad (7)$$

Standard deviation is given by

$$\sigma_c = \frac{\sum_{i=1}^n a_{ic} (x_i - \mu_c)^2}{\sum_{i=1}^n a_{ic}} \quad (8)$$

Weighting factor ω_c is given by

$$\omega_c = \frac{\sum_{i=1}^n a_{ic}}{p} \quad (9)$$

where p is the total number of data points. That is, the sum of level of association values for each of the Gaussian to an event divided by number of points.

5.2.3 Predicting user click

A common case study in clickstream data analysis is predicting a user's next click sequences or the last click event. The prediction could help to suggest the users' list of items as recommendations. We can use Markov chains to predict users' transition from one state to another. Markov chains were first introduced by the 19th century Russian mathematician Andry Markov [15, 31]. A Markov chain defines a stochastic process that determines the probability of the next events based upon the previous transitions and states achieved. In a Markov chain with k order, the future state relies on the previous k states. Formally defined as:

$$\begin{aligned} \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \dots, X_1 = x_1) \\ = \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \dots, X_{n-k} = x_{n-k}) \end{aligned} \quad (10)$$

$(n > k)$

Fitting into the Markov model, the probability distribution $X(n)$ of the n^{th} click is given by Raftery [38].

$$P^{(n)} = \sum_{i=1}^k \lambda_i M_i P^{(n-i)} \quad (11)$$

M_i is a $m \times n$ transition probability matrix and λ_i is the weight of each lag (time difference of two parts of a same stochastic process) in i , such that,

$$\sum_{i=1}^k \lambda_i = 1, \lambda_i \geq 0 \quad \forall i \quad (12)$$

If the final *absorbing* state [39] (checkout or add to cart) is known, then we can use the information about the final state F to predict more precisely the next click sequence as given by Equation 13.

$$P^{(n)} = F \sum_{i=1}^k \lambda_i M_i P^{(n-i)} \quad (13)$$

based on that, we can now set out the steps to predict clicks based on the Markov Model:

Creating a transition matrix: We compute a bigram transition matrix as the transition frequency between *two* websites links (items) as shown in Table 7:

To compute the transition probability, bigram values are normalized by the sum of each row as shown in Table 8.

Compute the Markov transition diagram: Table 8 graphically represented as the transition diagram (Fig. 6) with edges representing click probabilities from one item to another. The following properties are analyzed from the transition diagram:

1. **Periodicity:** If there exists a cycle.
2. **Irreducibility:** If the diagram is a complete graph (each vertex is connected to another by an edge).

Table 7 Bigram matrix

	Item 1	Item 2	Item 3	Item 4
Item 1	30	40	20	10
Item 2	40	30	10	20
Item 3	30	20	30	20
Item 3	30	20	30	20
Item 4	50	10	30	10

Table 8 Transition probability matrix

	Item 1	Item 2	Item 3	Item 4
Item 1	0.3	0.4	0.2	0.1
Item 2	0.4	0.3	0.1	0.2
Item 3	0.3	0.2	0.3	0.2
Item 4	0.5	0.1	0.3	0.1

Fig. 6 Transition diagram**Table 9** Transition probability matrix

Source	Destination	Probability
Item 1	Item 1	0.3
Item 1	Item 2	0.4
Item 1	Item 3	0.2
Item 1	Item 4	0.1

3. **Invariant Distribution:** If there exists a unique invariant distribution (given the Convergence Theorem [30]).

Considering Fig. 6, the transition diagram is classified as non-irreducible, aperiodic (as self-loops exist), and has no unique invariant distribution.

Compute the transition probability: Using the past transition matrix, probability of transition is computed from item 1 to other items using Equation 13. See Table 9.

Transition probability between two website links is given in Fig. 7. Users' *browsing depth* is the sequence or *path count* at the maximum levels (category and subcategory), which a user browses through in order to conclude their purchase or search (Fig. 8).

5.3 Minimizing database latency

The paper focuses on real-time, stream data analytics. Towards achieving a real-time turnaround time, this section outlines the strategy to achieve milliseconds range

Fig. 7 Clickstream transition probability between *from path* and *to path*. The transitions are often dominated by transitioning to the same path (such as checkout to checkout or addItem to addItem) and moving back to home

Fig. 8 Sequence count represents maximum depth (category and subcategory) a user chooses in order to navigate concluding their search. Interpreting the diagram, it is apparent, users frequently browse up to 2.5 levels on average

response by reducing the database latency. Recall from Fig. 5, in the Lambda Architecture, post-processing, Cassandra DB combines and persists the views from both stream and batch. Online Analytical Processing (OLAP) is the model for clickstream analysis and exploration, which deals with two *latency constraints*:

- *Data turn around time*: Is the latency between data ingested into the pipeline to the time data is queryable.
- *Query processing latency*: The speed at which results are returned, and reports are generated. OLAP requires interactive query speed in contrast to long-running *batch jobs* taking a few hours to days.

This section addresses the high volume of read-write operations through the Cassandra datastore and resolves the computational challenge in situations requiring to act faster, such as alerting, processing real-time, or near-real-time.

5.3.1 Data model

Distributed columnar datastore Cassandra load distributes storage and processing requirements across cluster nodes using a *partitioner* based on a *partition-key*. Data at each partition is further sorted on a *cluster-key*. Near-real-time stream processing applications depend on an efficient data modelling technique for optimal query

performance. The following are the constraints for a low latency response from a distributed system supported by Cassandra:

- A row cannot be *split across two nodes*.
- *Minimize network travel through the data model:*
We should minimize network load as much as possible. In other words, fewer the number of nodes the query needs to coordinate with, the better is the overall performance of the data model.

Illustrative example: The *click_master* table is an entity for the information such as the *session_id*, *click_sequence*, and the *click_location*. The *click_ticker* table is the second entity storing the *start_time* and *end_time*. Two tables are linked based on the *session_id* column.

click_master(*session_id*;, *click_sequence*, *click_location*)

click_ticker(*session_id*;, *start_time*, *end_time*) Constraints:

- Find duration and *click_sequence* for a *session_id* and *click_location* on a particular day.
- A row cannot be split across two nodes

Data Model: Based on the constraints, we can define the following data model.

```
CREATE TABLE clicks_by_date
(click_location varchar, session_id varchar,
click_sequence varchar, date varchar,
start_time decimal, end_time decimal
PRIMARY KEY ((click_location, date), session_id));
```

Observe, compound primary keys (*click_location*, *date*) forms the partition key which is ordered by cluster key *session_id*. The *partition key* in this instance is responsible for even distribution of keys across cluster and *co-locates* the keys based on *date* to enable faster *date-based query performance* [16].

5.4 Real-time processing using storm

Setting up a Storm Cluster. A 9 nodes Ubuntu OS was set up on Microsoft Azure HDInsight Storm cluster. In contrast to Farahabady's approach [12], this work considered a homogeneous Storm cluster setup for Nimbus or Supervisor to make the

most out of the default scheduler and load balancer. See Fig. 9. The cluster's benchmarking results against time is shown in Fig. 12, 13, 14.

Creating the Project

- Language: Java Spring
- IDE : Eclipse
- Build tool : Maven

The project structure is shown in Fig. 10.

5.4.1 Stream propagation through storm spouts and bolts

This section provides an overview of real-time event propagation through the series of *Storm Spouts and Bolts* with an objective to achieve milliseconds range response time.

Event Producer. Upstream click events propagate through a Kafka message queue to persist into a file system at real-time.

Storm Spout. Storm *Spouts* consume *Kafka topics* containing clickstream sequences and identify each unique users through a context ID.

Storm Bolts. Series of *bolts* transform each *tuple* of input as follows:

- *Item-viewed* list grouped by the context ID order by timestamp.
- Computes event hierarchy and the overall co-occurrence or *also-viewed* matrix between item pair I_i, RI_i as a sum of each unique session by a user. See Fig. 11.
- Compute user motifs through n-gram and student-t test.
- Compute customer segmentation through *K-means* and *E-M algorithms*.

9 nodes
TYPE	NODE SIZE	CORES	NODES
Nimbus	A3	8	2
Supervisor	D3 v2	16	4
Zookeeper	A3	12	3

Fig. 9 Microsoft Azure HDInsight Storm cluster with 9 nodes. The cluster consists of 2 Nimbus (master) and 4 Supervisors (worker) and 3 additional nodes for coordinating with Zookeeper servers. The storm cluster operates in speed layer of the Lambda Architecture which associated storage options such as Casandra, MongoDB or HBase

Fig. 10 Project structure with Java, Maven and Eclipse

Algorithm 1 Stream Propagation through Storm Spouts and Bolts to Create Co-occurrence Matrix**Require:** S : Co-occurrence matrix between item pair I_i, RI_i till time t Past dataset $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_t$ untill time t New dataset $\mathcal{D}_{t+1}$ at time $t+1$ n : Input feature variable k_t : Kafka topicMessage queue: *Linked Blocking Queue*Kafka Partitioner: *Consistent Random*Kafka topic k_t , DB server IP d_a ,**Ensure:** M : Updated co-occurrence matrix between item pair I_i and RI_i subscribeKafka(k_t, d_a , queue, partitioner)**for** windowed dataset $\mathcal{D}_{t+1}$ **do**Kafka-instance port $\leftarrow \mathcal{D}_{t+1}$. //Kafka port listening to stream events.Storm Spout $\leftarrow$ *Flat File Event Spout* //Storm Spouts consumes the file system data. I_i, RI_i (Storm Bolt) $\leftarrow$ Storm Spout**end for****return** I_i, RI_i

The algorithm is validated in a Storm cluster with the e-commerce data from Amazon and Flipkart (as discussed before). The statistics for throughput, Storm Topology and supervisor summary are shown in Fig. 12,13 and 14.

Fig. 11 Co-Occurrence Matrix between item pair I_i, RI_i for each unique session by a user

Fig. 12 Read throughput of the of 114 real-time mini-batches over the period of 1 hour 54 minutes. A read throughput of 300 records/sec is achieved under the streaming setting

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	8102	8102	21586.715	210	953
3h 0m 0s	150819	150819	20313.176	3591	18027
1d 0h 0m 0s	1205075	1205075	18252.193	26041	146691
All time	1373200	1373200	17976.799	29140	167680

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
\$mastercoord-bg0	1	1	197040	197040	17976.799	29140	167680				

Fig. 13 Storm topology stats for different time intervals. The statistics shows the efficiency of data processing between spouts and bolts. Failed transactions are retried automatically by the *Storm Nimbus*

Supervisor Summary

Host	Id	Uptime	Slots	Used slots	Avail slots	Used Mem (MB)	Version
wn0-gautam.1j4ve5z5m3culkrenqe25hguzb.cx.internal.cloudapp.net (log)	7b102150-6d45-4ee1-a976-9d0a5f14a3c8	1d 21h 53m 38s	4	1	3	2521	1.1.0.2.6.3.40-13
wn1-gautam.1j4ve5z5m3culkrenqe25hguzb.cx.internal.cloudapp.net (log)	d0f914ec-ce6c-4df5-8c99-77b49be17391	1d 21h 53m 37s	4	1	3	2521	1.1.0.2.6.3.40-13
wn2-gautam.1j4ve5z5m3culkrenqe25hguzb.cx.internal.cloudapp.net (log)	432448b9-49bf-4626-87dc-8cd96a532e3d	1d 21h 53m 48s	4	1	3	2521	1.1.0.2.6.3.40-13
wn3-gautam.1j4ve5z5m3culkrenqe25hguzb.cx.internal.cloudapp.net (log)	eebe0106-390a-4600-88c5-ed06652793fc	1d 21h 53m 40s	4	1	3	2521	1.1.0.2.6.3.40-13

Fig. 14 Storm supervisor summary of 4 worker nodes. The statistics shows the number of worker nodes running within their respective host and corresponding statistics are shown for *id*, *uptime*, *slots*, *used slots* and *available slots* fields

5.5 Clickstream capture and prediction model in microsoft azure cloud

Apache Storm cluster is hosted in *Microsoft Azure HDInsight* Cloud environment. Benefiting from the Azure capabilities, the stream capture model is robust in *failover* with no impact in cluster downtime when a worker node fails. The model builds a streaming pipeline with Azure Virtual Network Gateway, which consists of built-in authentication and load balancers for a production-ready deployment.

Fig. 15 Clickstream capture prediction model in the Microsoft Azure cloud. Feature and weight vectors (see Sect. 5.2.2) are extracted from clickstream in a cluster running the application server. The stream is pushed to the Microsoft Azure cloud through Apache Kafka

See Fig. 15 for the Azure deployment model. The performance metrics for the Azure cluster are captured in Figs. 18, 19.

5.5.1 Stress test

This section presents stress test results on the DataStax Enterprise (DSE) distribution of Cassandra. 3 million records are used to write first and run a mixed type load (read plus write) for another 3 million records. The server setup is shown in table 10. Refer to Fig. 16 and 17 for test results obtained through Datastax OPSCenter. Figures 18, 19 show the *average processing latency* in Azure HDInsight storage *unit 1* and *unit 2*.

6 Discussion

The proposed architecture, as depicted in Fig. 5 and consists of both a low latency real-time component as well as a high-accuracy batch counterpart, working simultaneously on the same dataset. The cost of batch and a real-time algorithm is defined by complexity, i.e., time, space, and resource utilization complexity. We can calculate the competitive ratio of a streaming algorithm against a batch algorithm. A smaller than *one* cost ratio can prove the superiority of the streaming process. However, the competitive ratio, due to time complexity, turns out to be greater than *one* in all scenarios and particularly worse in a certain dataset. This is less surprising because the streaming algorithm processes in mini-batches, without having the foresight of the entire dataset. On the

Table 10 Cassandra Setup

Cassandra Vendor	Number of Records	Replication Factor	Number of Nodes
DSE	4 Million writes	3	3

Fig. 16 Analyzing the number of requests for a given period reveals the underlying patterns about the read overhead and usage trends. Statistics are shown for (i) *Read Requests*, (ii) *Write Request Latency*, and (iii) *OS Disk Utilization*. (i) *Read Requests*: per second read request counts on all coordinating nodes. (ii) *Write Request latency(Percentiles)*: 99th, 90th percentiles, minimum, maximum and the median for a client writes. When a node accepts a client read request the period initiates, and terminates while node replies back to the client. Depending on the replication factor and consistency setting, this might include the network delay from the data replicas. (iii) *OS Disk Utilization*: CPU time used by disk I/O. The time unit is in milliseconds

Fig. 17 Statistics are shown for (i) *OS Load*, (ii) *Heap Used* and (iii) *TP Flashes Completed*. (i) *OS Load*: Operating system load average. Average data for every One minute data are parsed from `/proc/loadavg` statistics on Linux systems. (ii) *Heap Used*: Average of Java heap space utilized, (iii) *TP Flashes Completed*: Number of memtables flushed to disk since the nodes start

contrary, a real-time algorithm benefits from space and resource utilization complexity due to significantly less volume of data processing in each mini-batch. A real-time algorithm is α competitive if there are positive factors α and γ so that:

$$v_i \leq \alpha v_b + \gamma \quad (14)$$

v_i is the cost of the streaming algorithm and v_b is the cost at the batch setting.

From Equation 14, we conclude, an α competitive real-time algorithm has a cost not inferior to α times the optimal batch algorithm (v_b) plus some initial advantages (γ) assigned to an optimized batch algorithm, due to its prior knowledge of the full dataset [27]. Information about the size and schema of the entire dataset in advance is beneficial in terms of using a suitable algorithm and computing infrastructure.

In a classic trade-off between low-latency and high-accuracy, case studies dealing with the real-time dataset select low latency responses over *high accuracy* and *strong consistency*.

Fig. 18 Average latency in Azure HDInsight *storage unit 1*, representing the average delay for end-to-end requests sent to a storage operation. The latency time is the total duration required to process within Azure storage unit reading the request, dispatching response, and getting and acknowledgment

Fig. 19 Average latency in Azure HDInsight storage unit 2

The proposed model uses Cassandra DB to support low-latency requirements for the real-time component. Cassandra initiates a *read repair* to update the inconsistent data in a situation with a higher consistency setting. The client read-write processes, therefore, must wait before discrepancies are eliminated. Big data systems

that require swift turnaround time at near-real-time can keep the lowest likely value for consistency (which is, *consistency one*), due to the negative effect of consistency levels on general responsiveness. Cassandra allows tunable consistency settings, thus enabling the proposed model to act like a CP (partition tolerant and consistent) or AP (partition tolerant and available) system with regards to the CAP theorem [43].

A database is considered high consistent, if following condition is met:

$$\text{Definition 7} \quad r + w > n \quad (15)$$

r, w are the consistency levels of read and write.

n is the replication factor.

According to definition 7, for a replication factor of 3, if read plus write consistency level is 4 then consistency is considered strong on the fact that read/write operation verifies value from 2 out of 3 replicas.

Definition 8 The database is considered eventual consistent or has a weak consistency if the following condition is met:

$$r + w \leq n \quad (16)$$

r, w are the consistency levels of read and write.

n is the replication factor. According to definition 8, for a replication factor of 3, if *read plus write* consistency level is 3 or less, then the consistency is considered weak, since the read operation verifies values from 2 out of 3 replicas, and the write operation verifies from 1 out of 3 replicas. This may lead to read-inconsistent data in the case of immediate read requests after a write. However, the database *eventually* reaches consistency with additional delay in read operations [5]. *Eventual consistency* in a distributed systems allow all the replicas in different nodes to *eventually* reach the same state over time, without *locking* the data access. *Eventual consistency* is used to achieve *high availability*, since data is not locked for reading *but* can be *inconsistent* as replicas are still updating over the network.

In the clickstream analysis model, faster read and write is desirable due to the high velocity of data ingestion. Lowering both the write and *read consistency* to 1, can make the framework responsive at the cost of consistency [5]. However, we can opt for the eventual consistency when higher accuracy is required and follow as per the strategy laid down in Definition 8.

7 Conclusion

We presented a near real-time data storage and processing approach to analyze streams of data with Apache Storm and Cassandra NoSQL datastore. The model analyzes key customer footprint in terms of user clickstream trail. This study can contribute to e-commerce research and practice in terms of real-time ingestion, in-flight batch-stream hybrid processing, and storage approaches to analyze streams

of data. The proposed model analyzed key customer footprint in terms of users' clickstream trail. Click events were captured in real-time using *Kafka*, a distributed big data ingestion framework. *Apache Storm* processes data in real-time at stages (*bolts*) to create a *browsing hierarchy*, *item co-viewed matrix* and *frequency of co-occurred items (n-grams)* under a same browsing session. The relative difference of two clickstream distributions on similar click sequences was measured through *Student's t-test*. Closely related users were grouped to create customer segments using a hybrid model of the *K-Means* clustering and the *Expectation-Maximization* algorithm. Initial clusters were created at the batch mode, and the streaming APIs map each user click event to one of the clusters and update the cluster incrementally at the near-real-time setting. A model was proposed for predicting future click patterns through a *Higher Order Markov Chain*. Processed data was stored into *Cassandra database* to serve real-time responses in the future. This paper demonstrated several *Cassandra optimization techniques* on a multi-datacenter setup for serving near real-time responses. Experimental results for a Storm deployment in the Microsoft Azure HDInsight cluster provide important performance metrics. The metrics showed the data for cluster latency with database performance under stress. An approach was shown for building an optimized model for *Storm Topology* serving near real-time responses.

The proposed methods are generic and have potential beyond clickstream analysis with respect to the *Lambda architecture* which combines big data batch and stream processing approaches. In the future, this research will explore wider applications of click pattern behavioural systems and broaden the proposed models to other Human-centered computing (HCC) frameworks.

Funding This research was funded by Accenture Technology Labs, Beijing, China. Grant number RDF 15–02–35.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. <https://www.adobe.com/in/analytics/adobe-analytics.html>. Accessed: 1 Aug, 2019.
2. <https://wiki.apache.org/confluence/display/Hive/StatisticsAndDataMining>. Accessed: 1 Aug, 2019.
3. <https://data.world/promptcloud/fashion-products-on-amazon-com>. Accessed: 1 April, 2021.
4. <https://spark.apache.org/docs/latest/mllib-clustering.html>. Accessed: 1 June, 20189.
5. <https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutDataConsistency.html>. Accessed: 1 June, 2018.

6. Baheti, V. K. (2014). Windows azure hdinsight: where big data meets the cloud. in *2014 Conference on IT in Business, Industry and Government (CSIBIG)*, pp. 1–2. IEEE.
7. Baumann, A., Haupt, J., Gebert, F., & Lessmann, S. (2019). The price of privacy. *Business & Information Systems Engineering*, 61(4), 413–431.
8. Birjali, M., Beni-Hssane, A., & Erritali, M. (2017). Analyzing social media through big data using infosphere biginsights and apache flume. *Procedia Computer Science*, 113, 280–285.
9. Chauhan, A., Fontana, V., Hart, M., Tok, W. H., & Woody, B. (2014). *Introducing Microsoft Azure HDInsight*. USA: Microsoft press.
10. Dinsmore, T. W. (2016). Streaming analytics. In *Disruptive analytics*, pp. 117–144. Springer.
11. Dunning, T., & Friedman, E. (2016). Streaming architecture: new designs using Apache Kafka and MapR streams. O'Reilly Media, Inc.
12. Farahabady, M. R. H., Samani, H. R. D., Wang, Y., Zomaya, A. Y., & Tari, Z. (2016). A qos-aware controller for apache storm. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pp. 334–342. <https://doi.org/10.1109/NCA.2016.7778638>.
13. Friedman, E., & Tzoumas, K. (2016). Introduction to Apache Flink: stream processing for real time and beyond. O'Reilly Media, Inc.
14. Garg, N. (2013). *Apache kafka* (Vol. 591). UK: Packt Publishing Birmingham.
15. Gilks, W. R., Richardson, S., & Spiegelhalter, D. (1995). *Markov chain Monte Carlo in practice*. USA: Chapman and Hall/CRC.
16. Hanamanthrao, R., & Thejaswini, S. Real-time clickstream data analytics and visualization. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pp. 2139–2144. <https://doi.org/10.1109/RTEICT.2017.8256978>.
17. Hanamanthrao, R., & Thejaswini, S. (2017). Real-time clickstream data analytics and visualization. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pp. 2139–2144. IEEE.
18. Heidrich, J., Trendowicz, A., & Ebert, C. (2016). Exploiting big data's benefits. *IEEE Software*, 33(4), 111–116. <https://doi.org/10.1109/MS.2016.99>
19. Hoffman, S. (2013). *Apache Flume: distributed log collection for Hadoop*. UK: Packt Publishing Ltd.
20. Hoffman, S. (2015). *Apache flume: Distributed log collection for hadoop*. UK: Packt Publishing Ltd.
21. Huang, Z., Chen, H., & Zeng, D. (2004). Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. *ACM Transactions on Information Systems (TOIS)*, 22(1), 116–142.
22. Ichinose, A., Takefusa, A., Nakada, H., & Oguchi, M. A study of a video analysis framework using kafka and spark streaming. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2396–2401. <https://doi.org/10.1109/BigData.2017.8258195>.
23. Jarman, D. C., Zhou, Z. Q., & Chen, T. Y. (2017). Metamorphic testing for adobe data analytics software. In *Proceedings of the 2nd International Workshop on Metamorphic Testing*, pp. 21–27. IEEE Press.
24. Jones, M. T. (2013). Process real-time big data with twitter storm. IBM Technical Library.
25. Karakaya, Z., Yazici, A., & Alayyoub, M. (2017). A comparison of stream processing frameworks. In: *2017 International Conference on Computer and Applications (ICCA)*, pp. 1–12. IEEE.
26. Liu, Z., Wang, Y., Dontcheva, M., Hoffman, M., Walker, S., & Wilson, A. (2017). Patterns and sequences: Interactive exploration of clickstreams to understand common visitor paths. *IEEE Transactions on Visualization and Computer Graphics*, 23(1), 321–330. <https://doi.org/10.1109/TVCG.2016.2598797>.
27. Gerard, Maas, & Garillot, F. (2018). Learning spark streaming. In: *Learning Spark Streaming*, book section Chapter 3: Streaming Application Design. O'Reilly Media, Inc.
28. Manning, C. D., Manning, C. D., & Schütze, H. (1999). *Foundations of statistical natural language processing*. USA: MIT press.
29. Manzoor, M. A., & Morgan, Y. (2016). Real-time support vector machine based network intrusion detection system using apache storm. In: *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 1–5. <https://doi.org/10.1109/IEMCON.2016.7746264>.
30. Möhle, M. (1998). A convergence theorem for markov chains arising in population genetics and the coalescent with selfing. *Advances in Applied Probability*, 30(2), 493–512.

31. Neal, R. M. (2000). Markov chain sampling methods for dirichlet process mixture models. *Journal of computational and graphical statistics*, 9(2), 249–265.
32. Pal, G., Hong, X., Wang, Z., Wu, H., Li, G., & Atkinson, K. (2019). Lifelong machine learning and root cause analysis for large-scale cancer patient data. *Journal of Big Data*, 6(1), 1–29.
33. Pal, G., Li, G., & Atkinson, K. (2018). Big data real time ingestion and machine learning. In: 2018 IEEE Second International Conference on Data Stream Mining & Processing (DSMP), pp. 25–31. IEEE.
34. Pal, G., Li, G., & Atkinson, K. (2018). Multi-agent big-data lambda architecture model for e-commerce analytics. *Data*, 3(4), 58.
35. Pal, G., Li, G., & Atkinson, K. (2017). Multi-agent item to item contextual big data recommender system. *International Journal of Design, Analysis & Tools for Integrated Circuits & Systems*, 6(1), 58–59.
36. Papagelis, M., Plexousakis, D., & Kutsuras, T. (2005). Alleviating the sparsity problem of collaborative filtering using trust inferences. In: International Conference on Trust Management, pp. 224–239. Springer.
37. Peng, W. C., & Liao, Z. X. (2009). Mining sequential patterns across multiple sequence databases. *Data & Knowledge Engineering*, 68(10), 1014–1033.
38. Raftery, A. E. (1985). A model for high-order markov chains. *Journal of the Royal Statistical Society: Series B (Methodological)*, 47(3), 528–539.
39. Scholz, M., et al. (2016). R package clickstream: analyzing clickstream data with markov chains. *Journal of Statistical Software*, 74(4), 1–17.
40. Shuster, J. J. (2009). Student t-tests for potentially abnormal data. *Statistics in Medicine*, 28(16), 2170–2184.
41. Singh, M. P., Hoque, M. A., & Tarkoma, S. (2016). Analysis of systems to process massive data stream. arXiv preprint [arXiv:1605.09021](https://arxiv.org/abs/1605.09021).
42. Son, S., Lee, S., Gil, M. S., Choi, M. J., & Moon, Y. S. (2018). Locality aware traffic distribution in apache storm for energy analytics platform. In: 2018 IEEE International Conference on Big Data and Smart Computing (BigComp), pp. 721–724 . <https://doi.org/10.1109/BigComp.2018.00135>.
43. Thottuvaikkatamana, R. (2015). Data modeling considerations. In *Cassandra Design Patterns*, 2nd edn. Packt Publishing.
44. Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., et al. (2015). Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12), 1654–1655.
45. Xiang, D., Wu, Y., Shang, P., Jiang, J., Wu, J., & Yu, K. Rb-storm: Resource balance scheduling in apache storm. In *2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, pp. 419–423. <https://doi.org/10.1109/IIAI-AAI.2017.63>
46. Xiang, D., Wu, Y., Shang, P., Jiang, J., Wu, J., & Yu, K. Rb-storm: Resource balance scheduling in apache storm. In: *2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, pp. 419–423 (2017). <https://doi.org/10.1109/IIAI-AAI.2017.63>
47. Yan, L., Shuai, Z., & Bo, C. (2017). Multi-sensor data fusion system based on apache storm. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pp. 1094–1098. <https://doi.org/10.1109/CompComm.2017.8322712>
48. Zhou, H., Zouaq, A., & Inkpen, D. (2017). Dbpedia entity type detection using entity embeddings and n-gram models. In: International Conference on Knowledge Engineering and the Semantic Web, pp. 309–322. Springer.