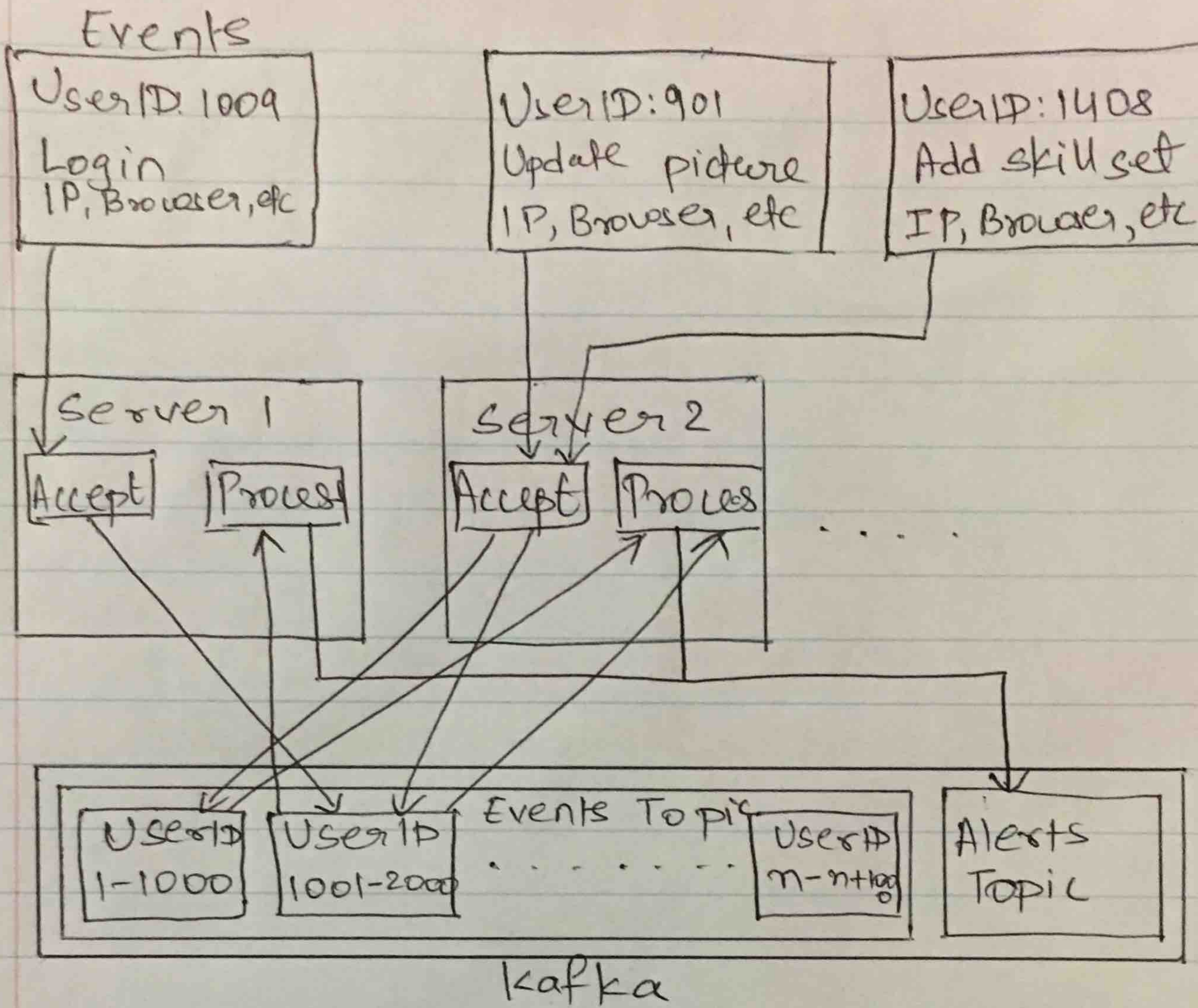


→ The figure above depicts the use of kafka to implement LinkedIn user profile experience. The normal operations possible here are login, Update profile information, add s~~k~~kill set, add new job, etc.

→ We create 2 topics here for

example.

1) Events topic

This will service all the requests for profile updates or operations. As shown in figure there are separate partitions for users, which ~~a~~ sends ~~re~~ different requests from same user to the same partition.

2) Alerts topic

This topic is basically for event logging and ingesting data into Hadoop for analytics.

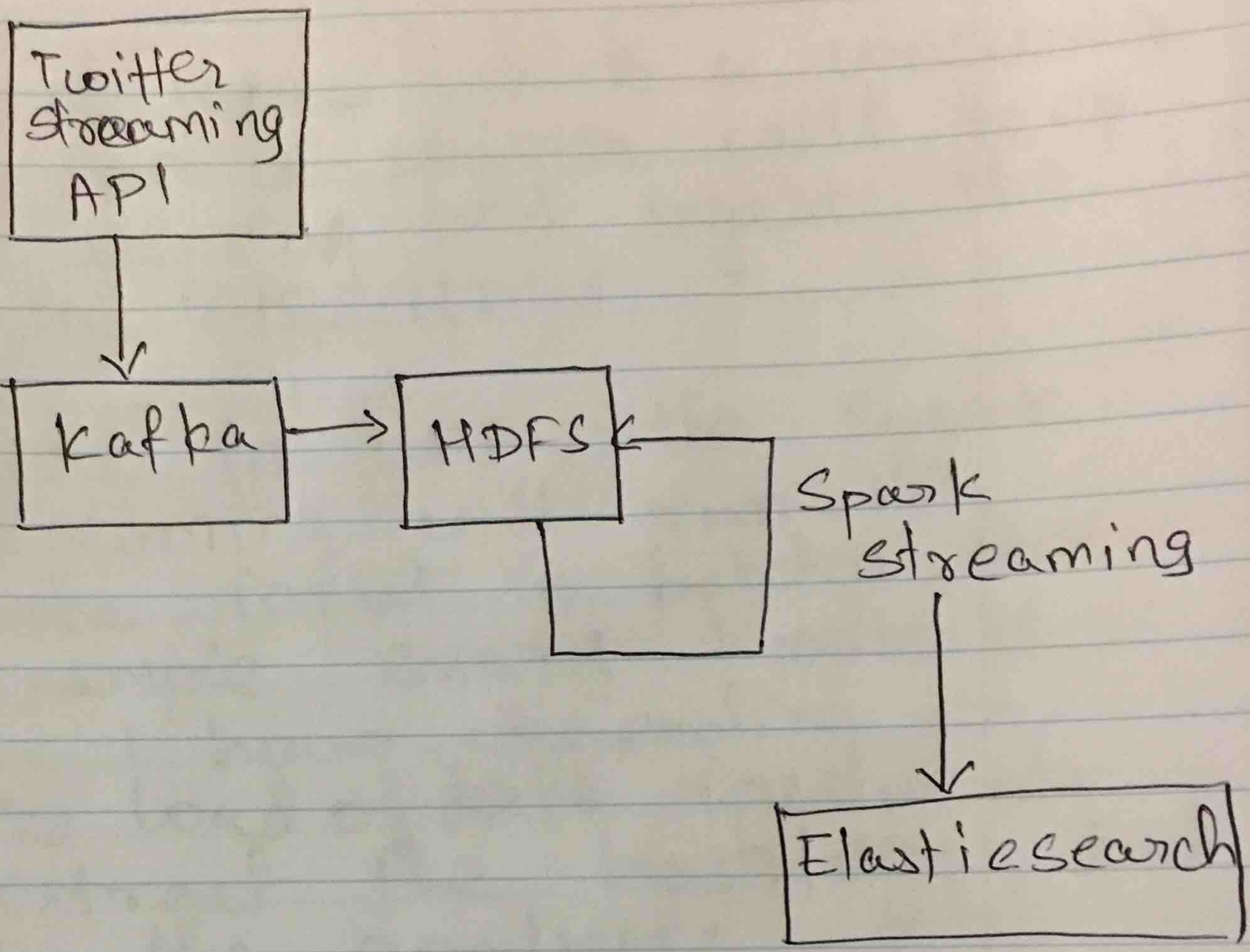
→ As shown in the figure as soon as user sends a request by performing an action, it goes to the nearest server first which then creates an event and adds into Kafka.

→ The events gets processed in

events topic. In each server there is a publisher which publishes events & there is consumer which takes responses. ~~If~~ All the log data, these consumer sends to Alerts topic where there can be another consumer to take those log and add into HDFS for analytics.

→ Since kafka is scalable, this design is highly scalable

(3)



→ As shown in the figure the twitter streaming API will be used to fetch the real time data. Suppose we use flask web app to download tweets using twitter streaming API.

→ This Flask App will keep adding tweets into kafka as events. Kafka will then publish to a topic which a consumer ~~no~~ on a server will keep consuming and ingest the data into HDFS.

→ Here in HDFS the spark streaming will run. i.e. take input in batches, for example every 1 minute or 1 hour depending on the load of data incoming extract the hashtags, and do the analytic

→ Finally the tweets will be stored in elastic search.

②

→ links = # RDD of (url, neighbors)
This will contain all the urls and its neighbors
i.e. A, C = A → C

→ ranks = # RDD of (url, rank)
This will contain the rank of each url.
i.e. A, 1

→ [url, [links, rank]] = pair
return [(dest, rank/len(links)) for dest in links]

On each iteration, have page p send a contribution of rank(p)/num_of_neighbors(p) to its neighbors.
↑
→ Contribs = links.join(ranks).flatMap (comput_contrib

→ ranks = contribs. reduceByKey
(lambda x, y: x+y) \
.mapValues (lambda x: 0.15 + 0.85 ~~**~~ * x)

ranks.saveAsTextFile(...)

↑

→ Set each page's rank to
0.15 + 0.85 x contributionsReceived

→ Then save the result in
text file.

→ The RDD generated

ranks = (url, rank).