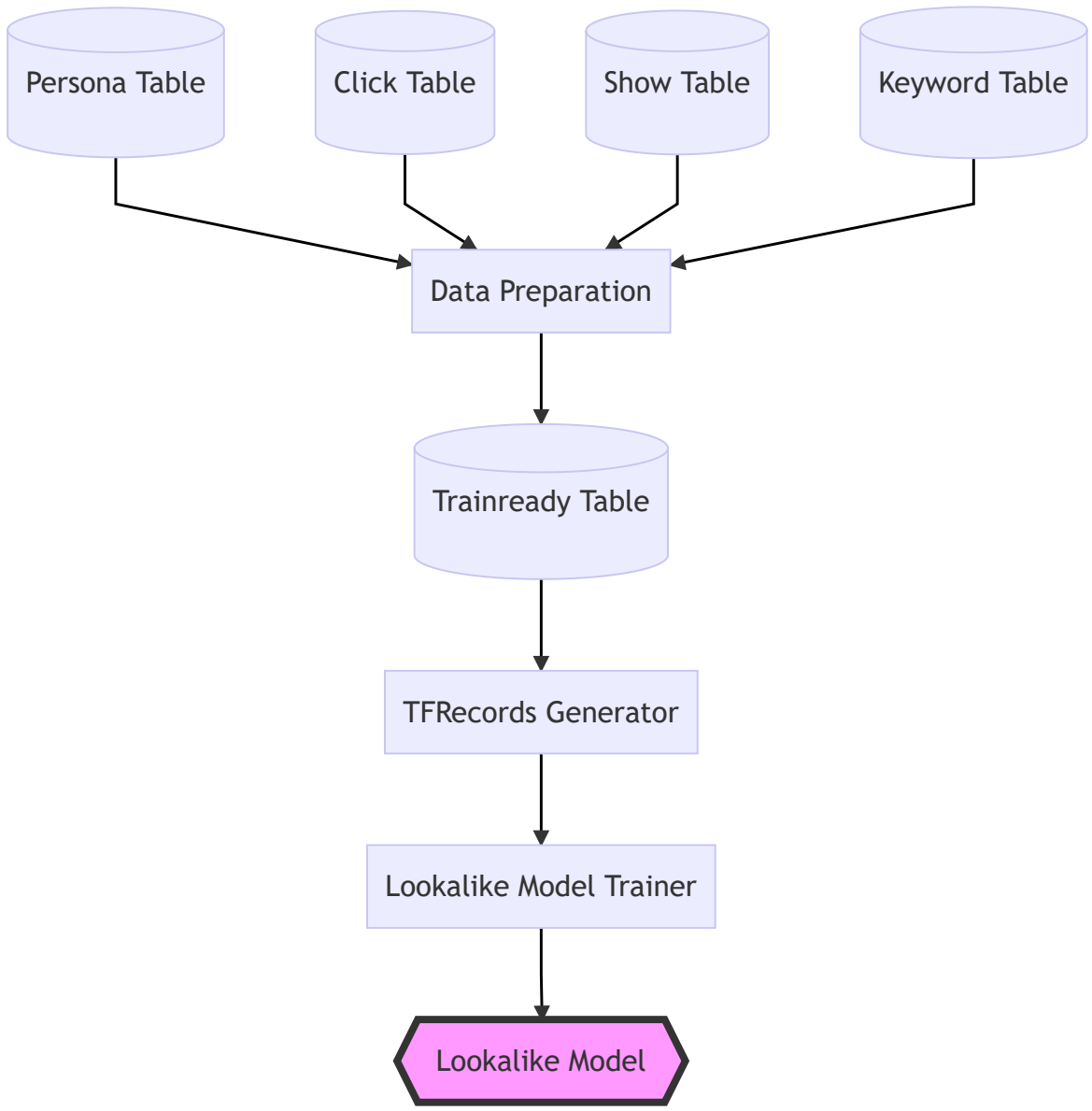
Lookalike Model and Services

Background

In Advertisement, lookalike models are used to build larger audiences from original audience (seeds). The larger audience reflects the characteristics of the seeds. This helps advertiser to reach a large number of new prospects.

Data Preparation

Lookalike Data Preparation



Overview

Data needs to be cleaned and transformed prior training. The data that is used for building a lookalike model has to posses the following characteristics:

- User ID
- User interests

The Lookalike model targets to setup the correlation between user and ads based on logged user behavior. The log data got from product is usually noisy due to various reasons, such as: unstable slot_id, redundant records or missing data. The data pre-processing is then required to clean original logs data before the data can be feed into the model trainer. Cleaning steps include transforming the logs data to new data format, and generating new data tables. This document of the data frame transformations for logs data pre-processing is designed for clarifying the whole processing steps on logs data and guiding the coding logic on the data pre-processing of DIN model.

Input data

The input tables for Lookalike model are the following log files.

- Persona table

```
>>> df=sql('select * from ads_persona_0520')
>>> df.count()
380000
>>> t1=df.take(1)
>>> m=map(lambda x:(x[0],x[1],t1[0][x[0]]),df.dtypes)
>>> for _ in m:
...     print(_)
...
### USED Fields
('did', 'string',
u'7eb9401a7ba0e377ad2fd81315b39da088b2943d69ff89c3ba56149d0a166d11')
('gender_new_dev', 'string', u'0')
('forecast_age_dev', 'string', u'3')
###
```

- Show-Log table

```
df=sql('select * from ads_showlog_0520')

### USED Fields
('did', 'string',
u'fd939b9efd2d9c512bcd360c38d32eedf03f6a11a935295c817f8e61caa2f049')
('adv_id', 'string', u'40023087')
('adv_type', 'string', u'native')
('slot_id', 'string', u'l03493p0r3')
('spread_app_id', 'string', u'C10374976')
('device_name', 'string', u'MAR-AL00')
('net_type', 'string', u'4G')
('adv_bill_mode_cd', 'string', u'CPC')
('show_time', 'string', u'2020-01-07 16:22:18.129')
###
```

- Click-Log table

```
df=sql('select * from ads_clicklog_0520')

### USED Fields
('did', 'string',
u'fd939b9efd2d9c512bcd360c38d32eedf03f6a11a935295c817f8e61caa2f049')
('adv_id', 'string', u'40023087')
('adv_type', 'string', u'native')
('slot_id', 'string', u'l03493p0r3')
('spread_app_id', 'string', u'C10374976')
('device_name', 'string', u'MAR-AL00')
('net_type', 'string', u'4G')
('adv_bill_mode_cd', 'string', u'CPC')
('click_time', 'string', u'2020-01-07 16:22:18.129')
###
```

Clean Log Data

1. **Clean Persona Table**

   This operation is to clean persona table to:

   - Have distinct did, gender and age.
   - Have did associated to only one age and gender

2. **Clean Show-log and Click-log Tables**

   The cleaning operation is performed in batches.

   Every batch is cleaned according to following policies:

   - Filter right slot-ids and add media-category
   - Add gender and age from persona table to each record of log
   - Add keyword to each row by using spread-app-id

```
>>> sql('show partitions lookalike_02022021_limited_clicklog').show(100,False)
+-------------------------+
|partition                |
+-------------------------+
|day=2019-12-19/did_bucket=0|
|day=2019-12-19/did_bucket=1|
```

```
|day=2019-12-20/did_bucket=0|
|day=2019-12-20/did_bucket=1|
+--------------------------+

>>> sql('show partitions lookalike_02022021_limited_showlog').show(100,False)
+--------------------------+
|partition                 |
+--------------------------+
|day=2019-12-19/did_bucket=0|
|day=2019-12-19/did_bucket=1|
|day=2019-12-20/did_bucket=0|
|day=2019-12-20/did_bucket=1|

>>> df=sql('select * from lookalike_02022021_limited_showlog')
>>> ...
>>>
('spread_app_id', 'string', u'C10608')
('did', 'string',
u'029d383c0cfd36dbe45ed42b2b784f06c04a6554b61c3919ea7b3681bc0fda39')
('adv_id', 'string', u'106559')
('media', 'string', u'native')
('slot_id', 'string', u'l2d4ec6csv')
('device_name', 'string', u'LIO-AN00')
('net_type', 'string', u'4G')
('price_model', 'string', u'CPC')
('action_time', 'string', u'2019-12-19 19:23:47.894')
('media_category', 'string', u'Huawei Reading')
('gender', 'int', 1)
('age', 'int', 4)
('keyword', 'string', u'info')
('keyword_index', 'int', 14)
('day', 'string', u'2019-12-19')
('did_bucket', 'string', u'1')
```

Partitions

Log tables should be **partitioned by DAY and DID** .

Remove Inactive users and Insignificant keywords

This section implements the following functionalities **(TO BE IMPLEMENTED)**

1. Exclude users whom have impressions for less than X days (5 days for example) in training period
2. Exclude users whom have total number of impressions less than Y during a period of time (100 impressions for 45 days)
3. Exclude keywords and related logs whose contribution is less than Z% (1% for example) of total traffic.

For more details of this algorithm refer to lookalike algorithm document.

Log Unification

The log unification process is performed in batches.

It processes data for each user partition separately and uses **load_logs_in_minutes** to load specific amount of log.

Here is a pseudocode for the process.

```
for each user-partition:
    start-time <- config
    finish-time <- config
    batch-size <- config
    while start-time > finish-time:
        batch-finish-time = start-time+batch-size
        read logs between start-time and batch-finish-time
        union logs
        save or append logs in partitioned tables ('day','user-did')
        start-time += batch-finish-time
```

Every batch of logs are unified according to following policies:

- Add is_click=1 to click log
- Add is_click=0 to show log
- Union show and click logs
- Add **interval_starting_time**

> This value shows the start_time of an interval.
>
> For example for the following time series and for interval_time_in_seconds=5
>
> [100,101,102,103,104,105,106, 107,108,109]
>
> the interval_starting_time is
>
> [100,100,100,100,100,105,105,105,105,105]

Here is a sample of this stage output.

```
>>> sql('show partitions lookalike_02022021_limited_logs').show(100,False)
+--------------------------+
|partition                 |
+--------------------------+
|day=2019-12-19/did_bucket=0|
|day=2019-12-19/did_bucket=1|
|day=2019-12-20/did_bucket=0|
|day=2019-12-20/did_bucket=1|
+--------------------------+

>>> df=sql('select * from lookalike_02022021_limited_logs')
>>> ...

('did', 'string',
u'02842b445b7779b9b6cd86a7cde292f15cd10128a8a80f19bea7c310c49160df')
('is_click', 'int', 0)
('action_time', 'string', u'2019-12-19 13:04:14.708')
('keyword', 'string', u'video')
('keyword_index', 'int', 29)
```

```
('media', 'string', u'native')
('media_category', 'string', u'Huawei Video')
('net_type', 'string', u'WIFI')
('gender', 'int', 0)
('age', 'int', 5)
('adv_id', 'string', u'40014545')
('interval_starting_time', 'int', 1576713600)
('action_time_seconds', 'int', 1576789454)
('day', 'string', u'2019-12-19')
('did_bucket', 'string', u'0')
```

Build Trainready Data

This phase is to

a) Aggregate data by user

b) Remove inactive users

c) Remove insignificant keywords

Data aggregation by user has 3 following steps:

1) Create attribution dictionary for each record. The attributes are ['interval_starting_time', 'interval_keywords', 'kwi', 'kwi_click_counts', 'kwi_show_counts']

2) Aggregate attribution dictionaries into a list for each user

3) Sort attribution list based on 'interval_starting_time'

4) Build feature array by extracting the attributes from the sorted attribution list.

Model Training

The model trainer reads records from Tfrecords and train the model. The built model is saved in HDFS.

Model Evaluation

The built model is compared against the previous model. If the new model performs better than the old model then it gets deployed and its name is stored in Zookeeper.

Model Deployment

Here are the steps to deploy the model.

```
1.  Pull a serving image : docker pull tensorflow/serving:latest-gpu
2.  Put the saved model in .../lookalike/<date>/tfserving directory.
3.  Run the serving image:
```

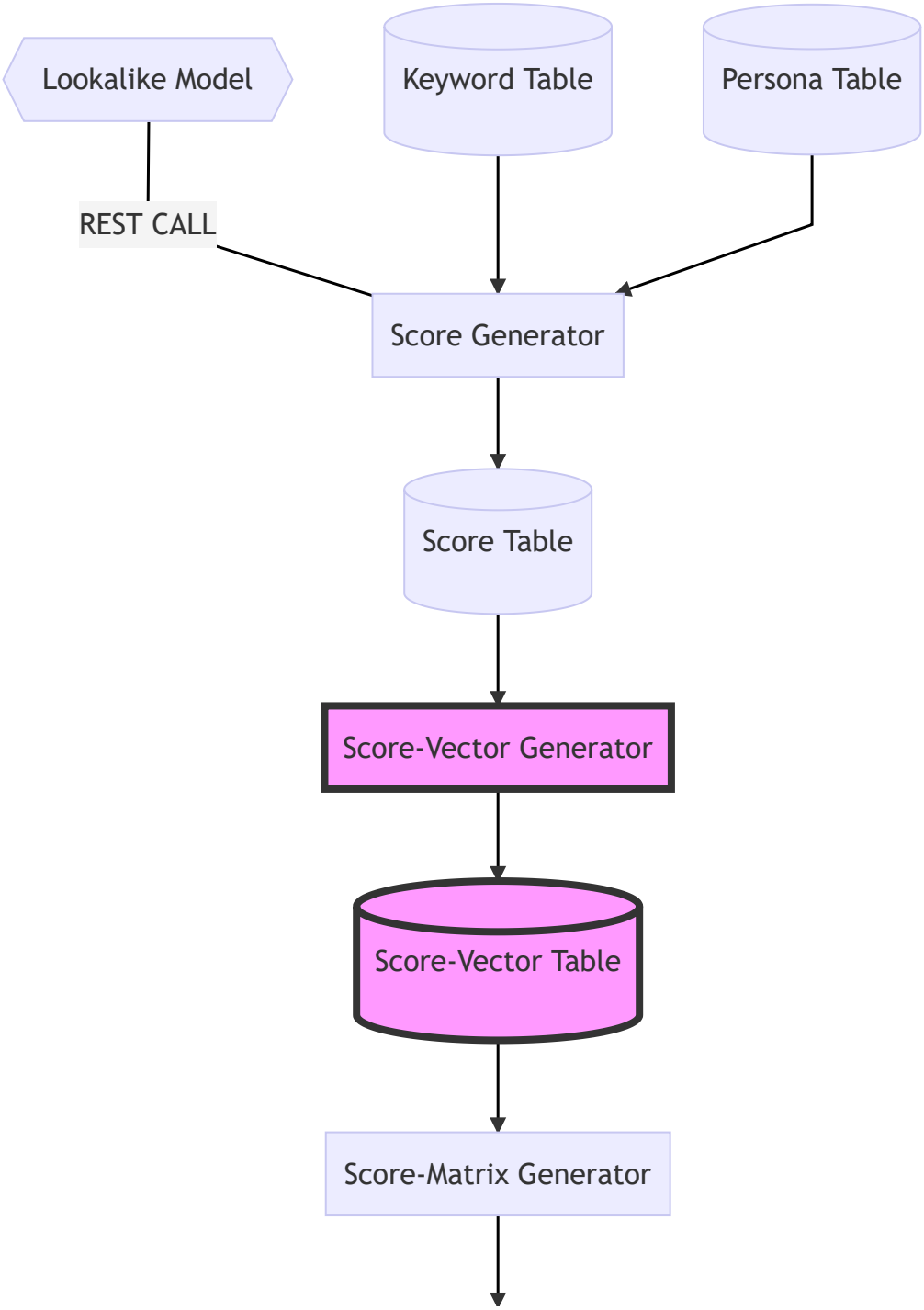
```
docker run -p 8501:8501 --mount
type=bind,source=/tmp/tfserving,target=/models/<model_name> -e MODEL_NAME=
<model_name> -t tensorflow/serving &
```

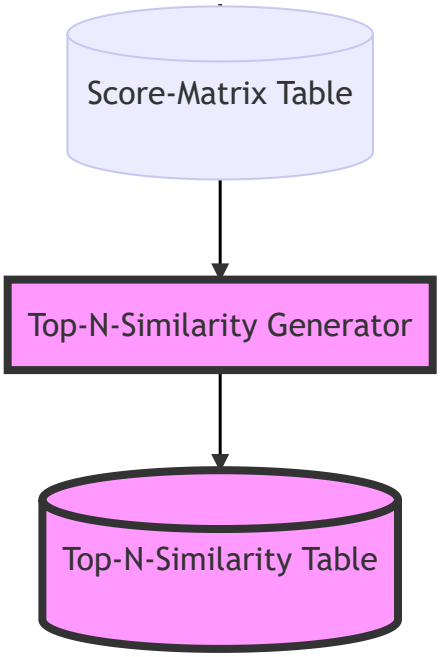
Lookalike Post Process - (Top-N-Similarity Table Generation)

This process builds tables that relates any user with its top N most similar users.

This is an offline process.

The following is a flow diagram of the process. The whole process is separated into 4 steps.





Here is an example of the final result.

Users	top-N-similarity-scores	did-bucket	similar-users
user-1	[similarity-score-11, similarity-score-12, similarity-score-13]	1	[user-did-1, user-did-2, user-did-3]
user-2	[similarity-score-21, similarity-score-22, similarity-score-23]	1	[user-did-11, user-did-22, user-did-33]
user-3	[similarity-score-31, similarity-score-32, similarity-score-33]	2	[user-did-10, user-did-20, user-did-30]

Restrictions

There is 1 restriction on the length of top-N-similarity-scores.

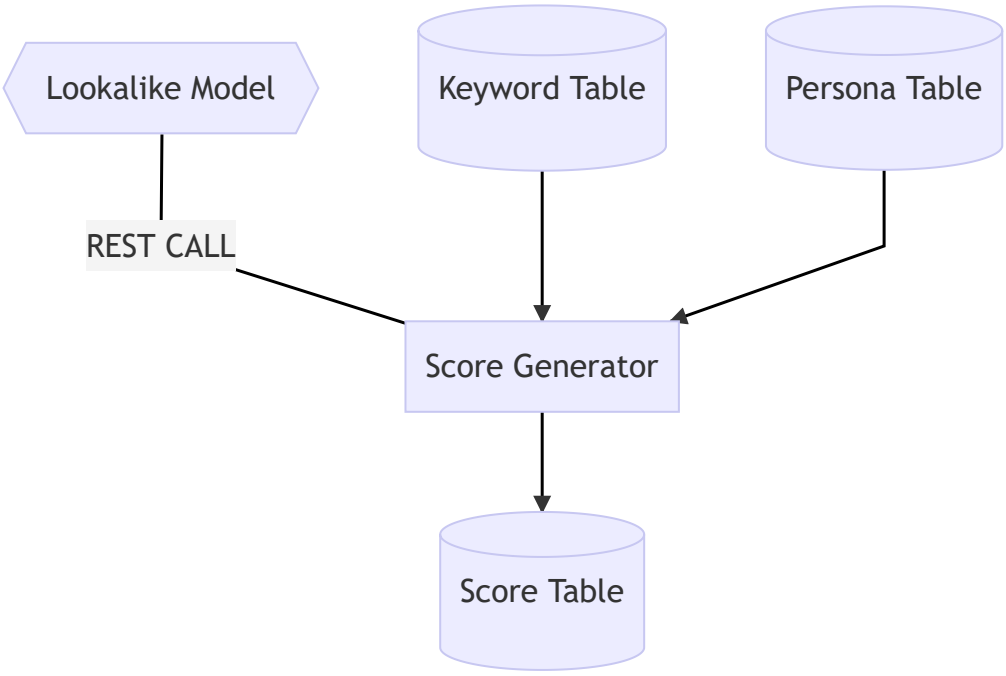
1. Size threshold: The top-N-similarity-scores array size is less than **max-similarity-vector-size (top_n)**.

Note: Score threshold is used when querying Top-N-Similarity Table, because the threshold might vary more often while the Top-N-Similarity Table is produced once per month.

Score Table Generation

This module uses Keyword Table, Persona Table and Lookalike Model to generate Score Table. The scores are ** not normalized**.

The score generator runs whenever a new lookalike model is generated. The name of the score table is stored in Zookeeper to be used by API services.



The score table has the following important columns. (The real table has more columns that are useful for troubleshooting purposes.)

	Keyword-Score	did-bucket
User-1	{kw1:norm-score-11, kw2:norm-score-12, kw3:norm-score-13}	1
User-2	{kw1:norm-score-21, kw2:norm-score-22, kw3:norm-score-23}	1
User-3	{kw1:norm-score-31, kw2:norm-score-32, kw3:norm-score-33}	2

DID-Bucket sharding strategy

The best way to create a sharding strategy is to benchmark production data on production hardware. As a general quick rule, the size of each bucket should be between 10G to 50G bytes. In our tests, the number of did-buckets is 1000, and bucket step is 10 for small cluster (5 servers) and 100 for medium size cluster (20 servers).

Score-Vector Generator

The first step of Top-N-Similarity process is to build the score-vector table. This table contains the same information as score table but in a vector format.

score-vector table is build based on a fixed array of keywords, hence the size of one vector is equal to the size of keywords.

Experience show float (np.float32) data type for score are faster than integer in pyspark and numpy world.

Users	score-vector	did-bucket
user-1	[score-11, score-12, score-13]	1
user-2	[score-21, score-22, score-23]	1
user-3	[score-31,score-32, score-33]	2

The size of score-vector table is about 100M-400M and the process is offline, with the same running frequency as of the lookalike-model trainer.

Score-Matrix Generator

The second step of Top-N-Similarity process is to build the score-matrix table. This table contains the same information as score-vector, but a group of rows are consolidated in one row in a form of matrix. This results into a fat dataframe with lots of data concentrated into one row. This approach increases the performance significantly (10 to 15 times in processing 1m users).

did-list	score-matrix	did-bucket
[user-1,user-2]	[[score-11, score-12, score-13],[score-21, score-22, score-23]]	1
[user-3,user-4]	[[score-31, score-32, score-33],[score-41, score-42, score-43]]	2

User consolidation into matrixes are based on did-bucket. All the users in the same bucket are converted into Matrix and become a row. The size of a matrix depends on the memory of nodes. The size of 10K x 10K for the matrix is used in experiments. This means that 10K users data is concentrated into 1 row.

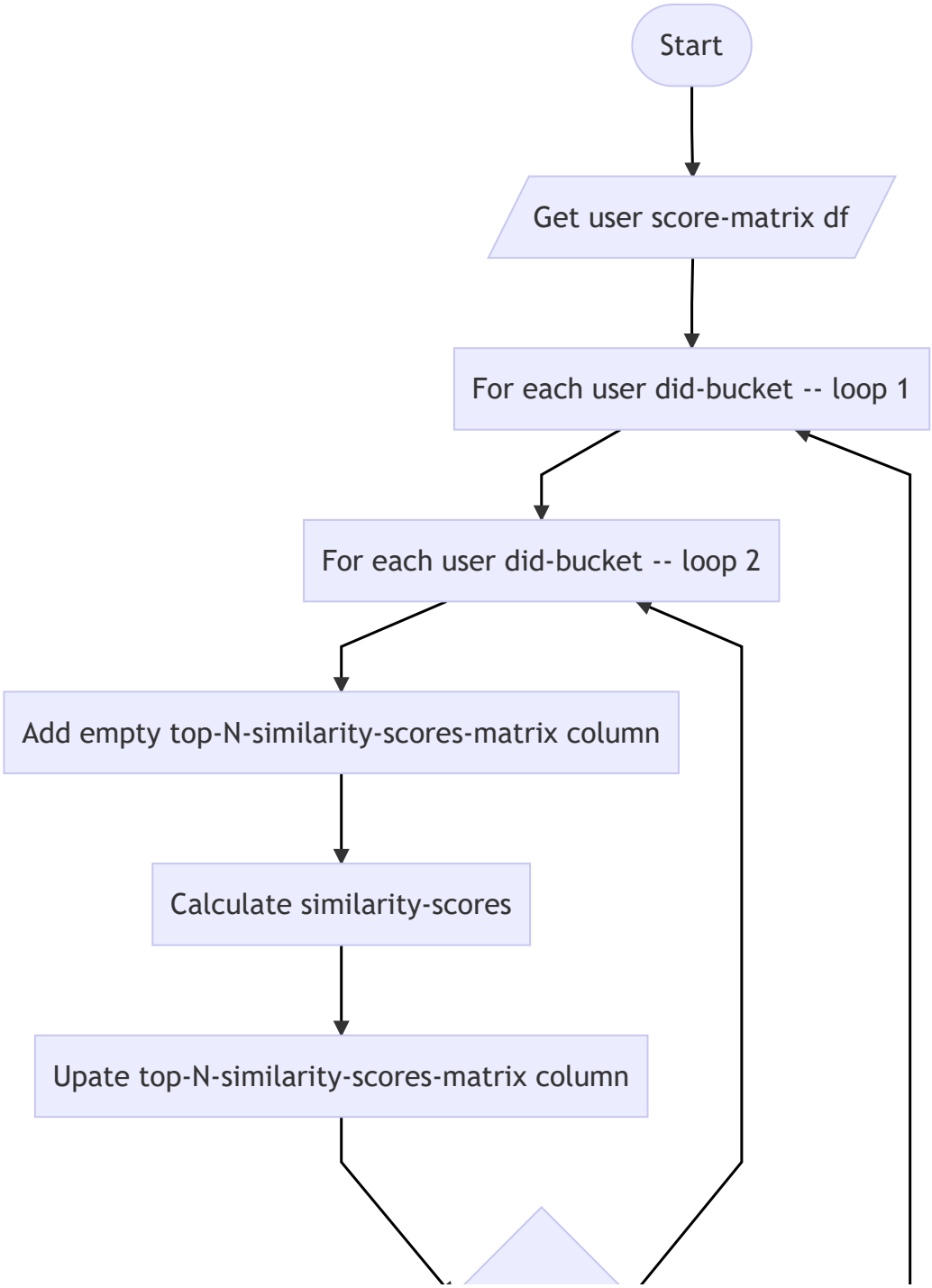
Top-N-Similarity Table Generator

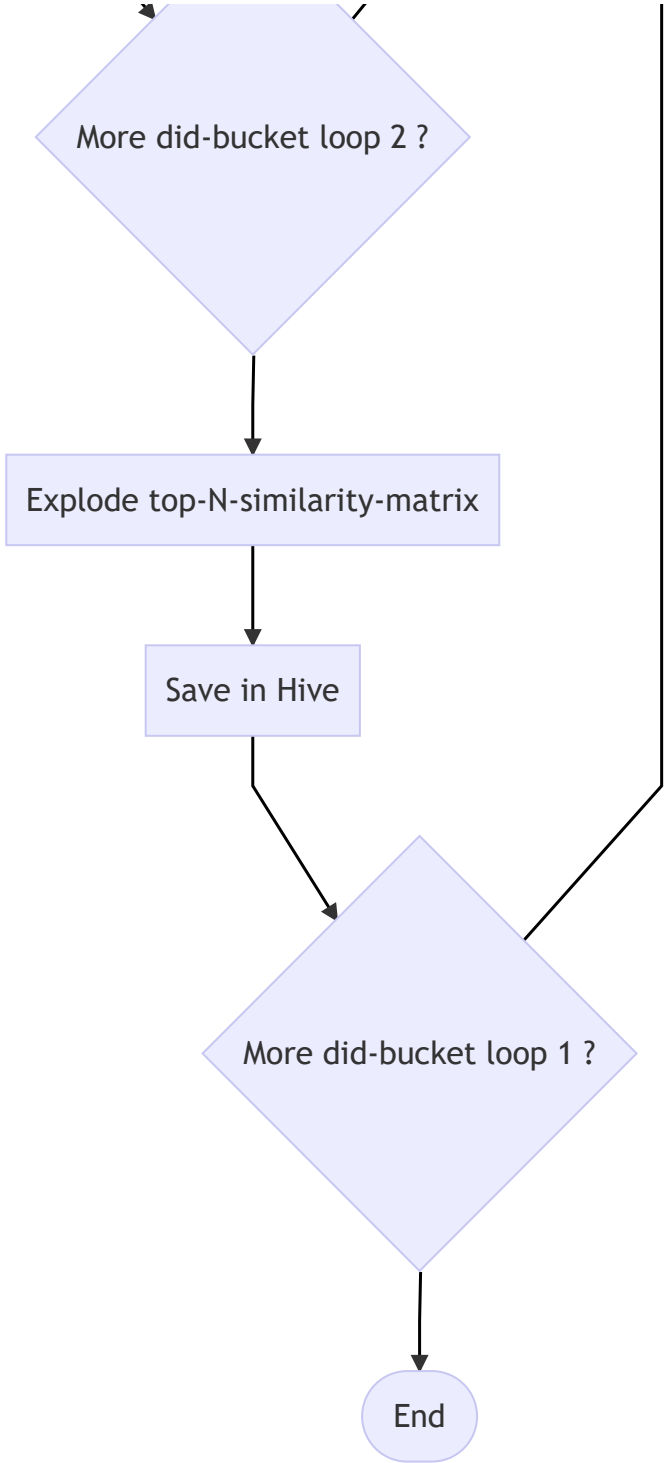
This process is to find top-n similar users for any user in the table.

The result dataframe has the following schema. This is an offline process.

	top-N-similarity-scores	did-bucket	similar-users
user-1	[similarity-score-11, similarity-score-12, similarity-score-13]	1	[user-did-1, user-did-2, user-did-3]
user-2	[similarity-score-21, similarity-score-22, similarity-score-23]	1	[user-did-11, user-did-22, user-did-33]
user-3	[similarity-score-31, similarity-score-32, similarity-score-33]	2	[user-did-10, user-did-20, user-did-30]

Similarity Generator has iterative process to produce top-N-similarity-scores. The following is the algorithm flow chart.





The possible caveat here is that the similarity score between 2 users is calculated twice, but this is better than storing the scores and reading them back. Also this calculation is on need-to-have bases, because the heap collection always keep the top N similar users and throws away unnecessary values. The other benefits of having double calculation are the simplicity of algorithm and having a full user table at the end.

Virtual Users (NOT IMPLEMENTED)

The other way to improve the performance is to use virtual users. One virtual user represents a closely group of users. Close users are such that have very close score-vector. This approach reduce the size of score-vector table and hence the performance.

Query Top-N-Similarity Table

There are 2 possible approaches to query the table depending on the size of the seeds.

m = number of seeds

n = size of similar-users threshold

1. Large size: Large size is considered more than 1000 seeds. For large size, the query will filter tables for **user-id IN seeds** and union (top-N-similarity-score,similar-users). The size of the extended users is between n and m*n

2. Small size: Small size is considered less than 1000 seeds. For small seeds, the query script broadcasts seeds to nodes and filter users which their similarity-users contain any of the seeds. This is reverse of the former approach. This approach builds larger set of lookalike users.

API Services

Abstract

This part describes the requirements, API, and high-level architecture for the Lookalike services.

List of Abbreviations

Acronym	Meaning
ES	Elasticsearch
LS	Lookalike Service
SG	Score Generator
ST	Score Table

Overview

The Lookalike Service is a real-time service whose purpose is to take an existing seed audience of users and return a group of additional users who have similar interests. This provides an expanded audience to which advertisers can target their ad campaigns.

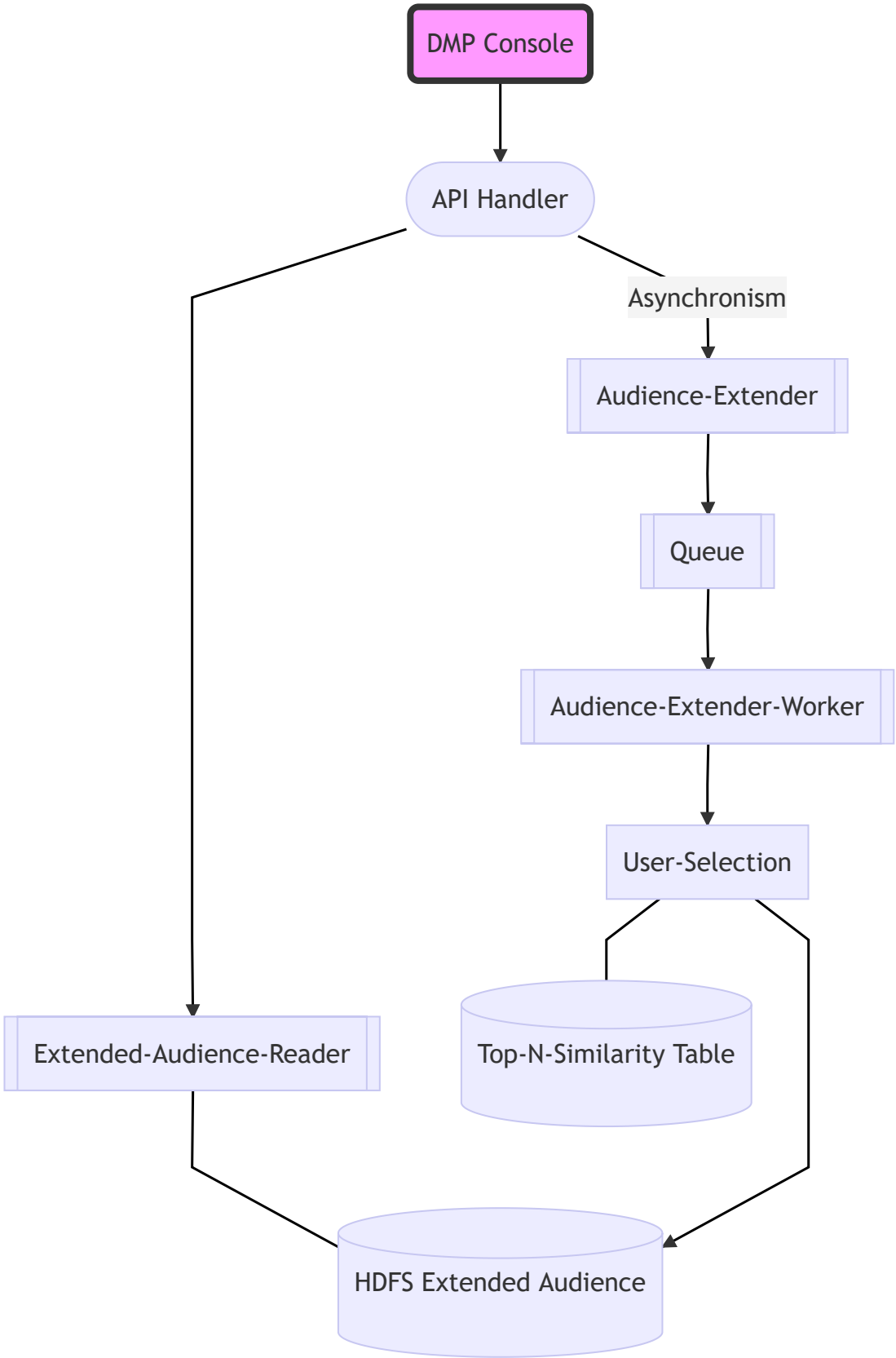
Experienced advertisers will develop an understanding of the audience that they want to target with their campaigns. This audience will be the users where the advertiser experiences the highest conversion rate for their metric of success. However, this approach can exclude a large number of their potential audience that would be similarly receptive to their advertising campaign. The Lookalike Service is intended to help advertisers expand the audience of their campaigns to users that are similar to their existing audiences that they would not otherwise know to target in their campaigns.

The Lookalike Service builds on the work done on the DIN model. From the DIN model, a correlation can be developed between users and topic keywords by the Score Generator. The Score Generator is an offline service that draws data from the DIN model and builds a score for each user for each of a given list of keywords. The score is a numeric metric for the affinity of this user for the topic represented by the given keyword. The scores for all users is stored in the Score Table.

Note: **The services are not implemented.**

Lookalike Services

System Entity Diagram


DMP Console
API Handler
Asynchronism
Audience-Extender
Queue
Audience-Extender-Worker
User-Selection
Extended-Audience-Reader
Top-N-Similarity Table
HDFS Extended Audience

User Selector

This module filter Top-N-Similarity table based on seed users and unions top-N-similar column.

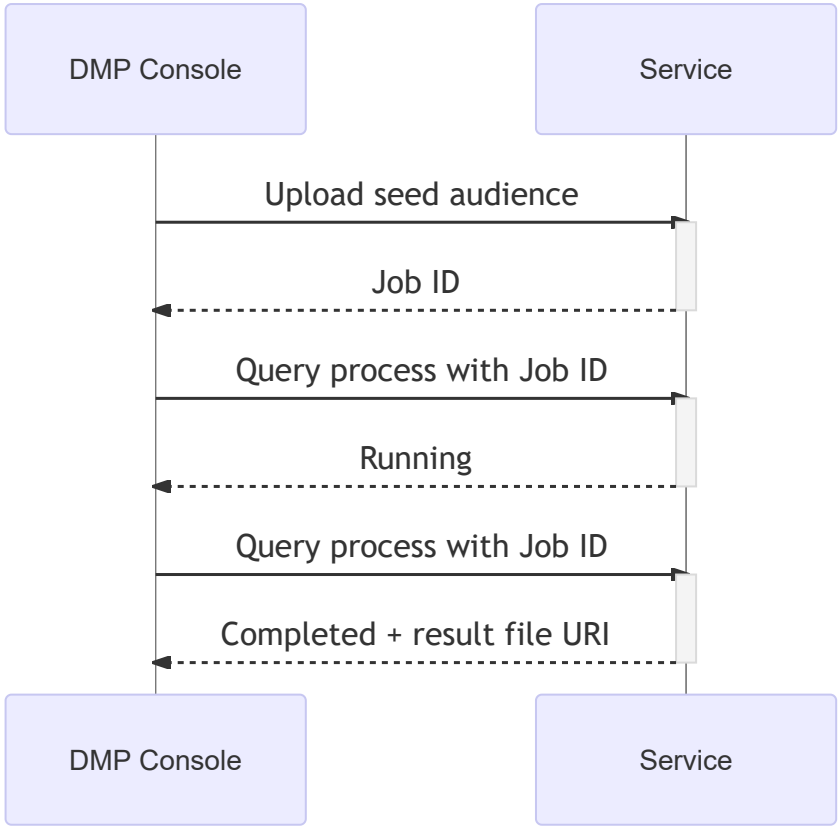
The result HDFS file carries the job-id in the path.

User-selector uses JOIN operation between seed table and user-similarity table ON did and did-bucket.

APIs

Use of the Lookalike Service is a multi-step process due to the size of the potential audiences and the scale of processing involved in generating the lookalike audiences.

Due to the scale of the computation required when generating the similarities between users (millions x 100millions), the processing of the Lookalike Service is divided into a REST endpoint to initiate the processing and a REST endpoint to poll the service on the completion of the processing.



Extend Audience API

```
Method: POST
Signature: /lookalike/extend-audience
Input:
    Multipart list of seeds (user dids)
Output:
    Job reference ID
```

- Sample output

```
{
    "job-id":"3746AFBBF63"
}
```

Implementation

This is an asynchronous call that initiates the processing of a lookalike audience and returns with a generated UID that should be used to poll for the completion of the request.

In the asynchronous process that is started:

The base directory that uploads are stored is read from Zookeeper. A UID will be generated for each upload request. The UID name will be used to create a subdirectory of the storage directory. The files will be uploaded into the subdirectory. The user IDs of the seed audience are loaded from the seed audience files.

The implementation is carried out by **Audience-Extender** service.

The top M most similar non-seed users to seeds are filtered and stored in HDFS file. M is loaded from Zookeeper.

Query Process Completion

```
Method: GET
Signature: /lookalike/status/<job-id>
Input:
    Job-ID which is unique ID string for the Lookalike Service request.
Output:
    Status of the request. Possible responses can be Running and Finished. If job
is Finished then the URI of result is presented.
```

- Sample output

```
{
    "statue":"Finished",
    "url": "../lookalike/extended-audience/<job-id>/exetened-audience-1p"
}
```

Implementation

This part is implemented by **Extended-Audience-Reader** service. This service checks HDFS for the URL of the extended audience. The service follows predefined template to construct URL from job-id. If the URL exits, it means job is finished and the module returns the url of the HDFS file.

Configuration

Configuration of the Lookalike Service will be stored in Zookeeper. The configuration parameters are:

- Hive table name for score table
- Number of clusters to group existing audience
- Number of highest similarities to include in average
- Percentage of user extension

Spark Environment Tuning

Low performance on Spark operations can caused by these factors:

1. Level of Parallelism
2. Data Locality

Level of Parallelism

Spark is about parallel computations. Too low parallelism means the job will be running longer too high parallelism would require a lots of resource. So defining the degree of parallelism depends on the number of cores available in the cluster. **Best way to decide a number of spark partitions in an RDD is to make the number of partitions equal to the number of cores over the cluster.**

There are 2 properties which can be used to increase the level of parallelism -

```
spark.default.parallelism
spark.sql.shuffle.partitions
```

`spark.sql.shuffle.partitions` is used when you are dealing with spark SQL or dataframe API.

A right level of Parallelism means that a partition can be fit into a memory of one node. To achieve right level of Parallelism follow these steps:

> a. Identify right about of memory for each executer.
> b. Partition data so that each partition can be fit into memory of a node.
> c. Use right number of executers.
> d. Respect partitions in queries

Data Locality

Data locality can have a major impact on the performance of Spark jobs. If data and the code that operates on it are together, then computation tends to be fast. But if code and data are separated, one must move to the other. Typically it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. Spark builds its scheduling around this general principle of data locality.

Calling `groupBy()`, `groupByKey()`, `reduceByKey()`, `join()` and similar functions on dataframe results in shuffling data between multiple executors and even machines and finally repartitions data into 200 partitions by default. Pyspark default defines shuffling partition to 200 using `spark.sql.shuffle.partitions` configuration.

Experiment

The project was run on the spark cluster version 2.3 with Java 8.

Spark Environment Settings

The Hadoop cluster has the '600GB' Memory and '200' V-Cores.
The following command was used for each step of the pipeline.

```
spark-submit --master yarn --num-executors 20 --executor-cores 5 --executor-
memory 8G --driver-memory 8G --conf spark.driver.maxResultSize=5g --conf
spark.hadoop.hive.exec.dynamic.partition=true --conf
spark.hadoop.hive.exec.dynamic.partition.mode=nonstrict <python-file> config.yml
```

This command engages 50% of the cluster (110 V-Cores) to carry out the operation.

Elapsed Time

The following is the elapsed time for each step of the pipeline.

STEP	INPUT TABLE NAME	TABLE SIZE RECORDS	PARTITIONS	ELAPSED
main_clean.py	ads_cleanlog_0520	1,036,183	NONE	51mins, 18sec
	ads_showlog_0520	44,946,000		
	ads_persona_0520	380,000		
main_logs.py	lookalike_02242021_clicklog	251,271	DAY,DID	4mins, 41sec
	lookalike_02242021_showlog	12,165,993		
main_trainready.py	lookalike_02242021_logs	12,417,264	DAY,DID	15mins, 0sec

Debugging for Performance Bottlenecks

One way to find a bottleneck is to measure the elapsed time for an operation.

Use the following code after a specific operation to measure the elapsed time.

```
import timeit
def get_elapsed_time(df):
    start = timeit.default_timer()
    df.take(1)
    end = timeit.default_timer()
    return end-start
```

For example in the following pyspark code, the `get_elapsed_time(df)` is called in 2 different places. Note, that the time measurement is from the beginning of the code up to the place where `get_elapsed_time(df)` is called.

```
 trainready_table_temp
    batched_round = 1
    for did_bucket in range(did_bucket_num):
        command = """SELECT *
                        FROM {}
                        WHERE
                        did_bucket= '{}' """
        df = hive_context.sql(command.format(trainready_table_temp, did_bucket))
        df = collect_trainready(df)
        print(get_elapsed_time(df))

        df = build_feature_array(df)
        print(get_elapsed_time(df))

        for i, feature_name in enumerate(['interval_starting_time',
'interval_keywords', 'kwi', 'kwi_show_counts', 'kwi_click_counts']):
            df = df.withColumn(feature_name, col('metrics_list').getItem(i))

        # Add did_index
        df = df.withColumn('did_index', monotonically_increasing_id())
        df = df.select('age', 'gender', 'did', 'did_index',
'interval_starting_time', 'interval_keywords',
                       'kwi', 'kwi_show_counts', 'kwi_click_counts',
'did_bucket')

        mode = 'overwrite' if batched_round == 1 else 'append'
        write_to_table_with_partition(df, trainready_table, partition=
('did_bucket'), mode=mode)
        batched_round += 1

    return
```

Application testing

To run the application test run.sh should be run. By running it, 4 lines of code would be run one after each other.

```
spark-submit --executor-memory 16G --driver-memory 24G  --num-executors 16 --
executor-cores 5 --master yarn --conf spark.driver.maxResultSize=8g
seed_user_selector.py config.yml "29" ;
spark-submit --executor-memory 16G --driver-memory 24G  --num-executors 16 --
executor-cores 5 --master yarn --conf spark.driver.maxResultSize=8g
score_generator.py config.yml ;
spark-submit --executor-memory 16G --driver-memory 24G  --num-executors 16 --
executor-cores 5 --master yarn --conf spark.driver.maxResultSize=8g
distance_table_list.py config.yml ;
spark-submit --executor-memory 16G --driver-memory 24G  --num-executors 16 --
executor-cores 5 --master yarn --conf spark.driver.maxResultSize=8g
validation.py config.yml "29";
```

A brief description for run.sh is as following:

> a. The first line of the code in the run.sh gets the config.yml and keyword index as an argument, create a list of seed users and write it to a hive table. The number of seed users is configurable and can be changed.
> b. The second line gets the config.yml in the argument and trainready table as an input. score_generator.py send the instances to the Rest API and write the responses to the score table.
> c. The third line gets the config file in the argument and score table as an input and create a distance table.
> d. The last line in the run.sh file gets config file and keywords index in the argument. The validation.py is calculating the number of clicks among the lookalike extended users, in the specific keywords and compare it with the number of click in the random selection.

Application Pipeline Performance

The application pipeline includes:

1. Score Table Generator
2. Score Vector Generator
3. Score Matrix Generator
4. Top-N-Similarity Table Generator

The most time consuming section is Top-N-Similarity Generator where the cross product of millions of users is calculated.

As the last experiment,

1. The elapsed-time for calculating similarity table for 10K user with 10 keywords on single machine is between 5 to 10 minutes.
2. The elapsed time for 1M user in 100 core cluster with 10 keywords in between **90 - 120 minutes**.

There are some research to use faster formula and packages for similarity calculation.