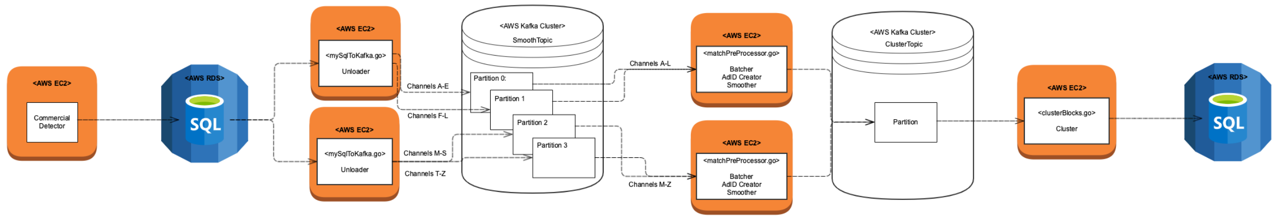
AdDB Architecture V2 (Go)

General Overview

Using the output of commercial detector match logs, which are stored in a MySQL database, 1) matches are unloaded to Kafka by channel in realtime. 2) Read Kafka data is batched by channel. 3) A MatchSetID is assigned to each match based on start and end time similarity to previous matches with the same match_idx. 4) Batched channel data is smoothed through merging, filling, concatenation, and fixing boundary jitters. 5) Smoothed data is sent to Kafka. 6) Read Kafka data is clustered by MatchSetID and duration similarity. And finally, 7) Clusters, with their corresponding smoothed matches, are stored in MySQL.

Located in the goaddb repository, in branch DEV-4968.

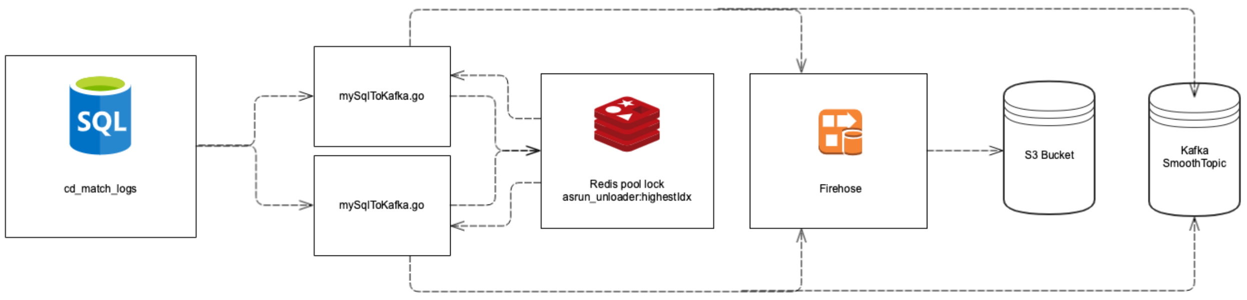


Modules

Unloader

The unloader continuously grabs cd_match_logs from the MySQL database in realtime. The mySqlToKafka.go instances, one after the other specified by the Redis pool lock, query MySQL to find its most recent entry (or highest idx value). They then query the logs between the current asrun_unloader:highestIdx and the previously queried highest idx value, with a maximum batch size specified as 5000, and they re-set asrun_unloader:highestIdx. After unlocking, the instances write these queried batches to Kafka (to continue down the pipeline) and Firehose to S3 (for storage in case of data loss).

In terms of Kafka, messages are written to Kafka partitions based on channel name (Balancer for Kafka writer is set to Hash) because the next module (smoothing) processes matches per channel.



Batcher/AdID Creator/Smoother

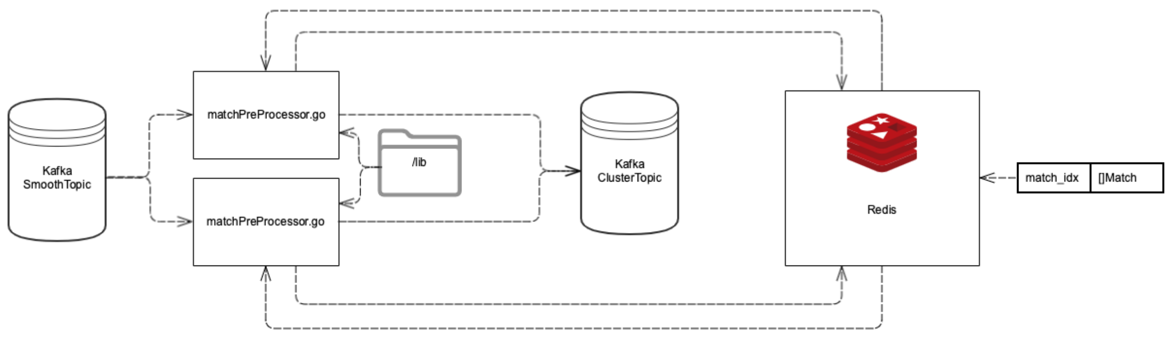
The matchPreProcessor.go Kafka consumers split reading from an equal number of partitions to maintain channel consistency. Channel messages are fetched and only committed once smoothing is completed. In case of an instance crashing, this allows fetched messages to remain in Kafka until they have been fully processed.

Once a match is fetched, it is compared to a list of previous matches grabbed from Redis based on its match_idx and either 1) assigned an existing MatchSetID of a similar match or 2) assigned a new MatchSetID when no similar matches exist and added to Redis.

After ID creation, matches are continuously added to batches by channel in channelBlocks. Each batch has to meet two requirements in order to be smoothed: a) it has to be a certain size (NumProcessBlocks) and b) an incoming match of the same channel's StartTs has to be at least a minute away from the batch's last EndTs. If this time gap is more than a day, however, the batch is considered old and is not smoothed.

Once batched, matches of the same channel are smoothed. Smoothing consists of merging, filling, concatenating, and fixing boundary jitters. Each step works to either combine matches that are similar in terms of start and end times or separate matches that slightly overlap. When a match is combined with another match, it is put in the other match's MergedMatches list for reference.

The number of smoothed matches should be much lower than the number of raw matches.

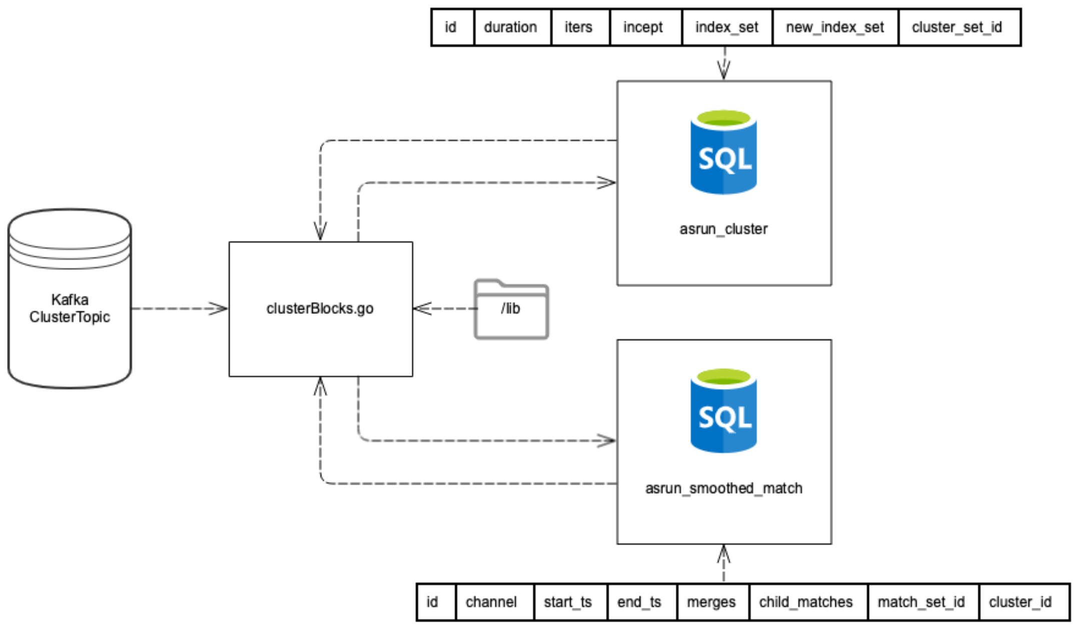


Cluster

Once an instance of clusterBlocks.go starts running, both MySQL tables are scanned and put into a map stored in memory (MatchMap) as a failsafe in case instances go down. The clusterBlocks.go consumer then fetches a smoothed batch of matches and only commits the message once clustering has completed on that batch. In case of an instance crashing, this allows fetched messages to remain in Kafka until they have been fully processed.

Each match from the batch is clustered individually. The first step of clustering is creating an index_set for the match, which is a list of unique MatchSetID's of both the match and its MergedMatches. The duration of the matches associated with each of these indexes is compared to the durations of the matches associated with the same indexes in MatchMap. If the durations are similar and there is a large intersection between the two index_sets, the MatchMap cluster is updated with the incoming match data. Updated clusters' index_set's are also updated with new_index_set. Otherwise, a new cluster is created.

These clusters, along with the smoothed match they are associated with, are then stored in MySQL.



Input/Output

The input to the addb system are matches from cd_match_log in MySQL, with an example shown below:

idx	mci_idx	channel	match_idx	start_ts	end_ts	len	match_start_ts	match_end_ts	match_len	happened	instance
9484207138	112112357	LOGOHD	36028797061021015	2019-08-06 17:18:35	2019-08-06 17:18:42	00:00:07	2019-06-15 11:56:14	2019-06-15 11:56:21	00:00:07	2019-08-06 17:18:42	172.17.15.

The output to the addb system are clusters associated with smoothed matches from asrun_cluster and asrun_smoothed_match respectively in MySQL, with an example shown below:

id	duration	iters	incept	index_set	new_index_set	cluster_set_id
18524	12	1	2019-08-09 16:06:02	[81418 127275 81434 81812 81913 81399 81925 81919 81924 81806 81815 81671 81814 81916 81408 81912 81923 81429]	[81418 127275 81434 81812 81913 81399 81925 81919 81924 81806 81815 81671 81814 81916 81408 81912 81923 81429]	5485

id	channel	start_ts	end_ts	merges	child_matches
23961	LOGOHD	2019-08-02 21:18:49	2019-08-02 21:19:05	19	[{"channel":"LOGOHD","match_idx":"36028797061021015","start_ts":"","end_ts":"","match_start_ts":"","match_end_ts":""," ... ...]

As you can see, the id in asrun_cluster corresponds to the cluster_id in asrun_smoothed_match. The new_index_set in asrun_cluster corresponds to the MatchSetID's and match_set_id in asrun_smoothed_match (and its child_matches). There are 19 total child_matches in the above example, but it's too long to display.

The input example shown above is only part of the input needed to produce the output. Making a single cluster requires over 1,000 matches, which is too long to display.

Here is a deeper look into asrun_cluster and asrun_smoothed_match:

create table

```
CREATE TABLE asrun_cluster (
    id MEDIUMINT NOT NULL AUTO_INCREMENT,
    duration INT,
    iters INT,
    index_set TEXT,
    new_index_set TEXT
 incept TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id)
);

CREATE TABLE asrun_smoothed_match (
    id MEDIUMINT NOT NULL AUTO_INCREMENT,
    channel VARCHAR(10),
    start_ts TIMESTAMP,
    end_ts TIMESTAMP,
    merges INT,
    child_matches TEXT,
    match_set_id MEDIUMINT,
    cluster_id MEDIUMINT,
    PRIMARY KEY (id)
);
```

Another byproduct of the output is the Redis match_idx map, with an example shown below:

match_idx	[]Match
36028797061021015	[{\"channel\":\"WNYWDT\",\"match_idx\":\"36028797061021015\",\"start_ts\":\"\",\"end_ts\":\"\",\"match_start_ts\":\"\",\"mat {\"channel\":\"WITIDT\",\"match_idx\":\"36028797061021015\",\"start_ts\":\"\",\"end_ts\":\"\",\"match_start_ts\":\"\",\"match_ {\"channel\":\"LOGOHD\",\"match_idx\":\"36028797061021015\",\"start_ts\":\"\",\"end_ts\":\"\",\"match_start_ts\":\"\",\"match

Running

The system acts very much like a pipeline, with producers sending data to consumers in realtime. Throughout the pipeline, there are Redis and SQL databases (shown in modules) that are used to store 1) match lists by matchSetID, 2) the cluster matchmap, and 3) the smoothed matches associated with each cluster. This stored information will be used for ad extraction, and is also necessary as a failsafe for this addb architecture. In the case of a crash of any of these components, stored data will be read from either Redis or MySQL to maintain current state.

Inputs for database hosts/ports, etc. are inputted as environment variables into each go application.

Each module should be within its own thread, in a pipeline producer consumer configuration. However, multiple instances of the mySqlToKafka.go and matchPreProcessor.go can be run concurrently on separate EC2 instances. Note that the number of matchPreProcessor.go (Kafka consumer) instances should never be greater than the number of Kafka partitions. In terms of scaling, ideally creating one partition per channel with an equal number of matchPreProcessor.go consumers would optimize batching, id creation, and smoothing. Any number of mySqlToKafka.go (Kafka producer) instances can be run at once; however, each instance transfers SQL records extremely fast (hence the 20 second sleep), so there is no major benefit.

In the case of all the mySqlToKafka.go instances crashing, delete asrun_unloader:highestIdx from Redis to prevent the pipeline from reading old data.

Currently, to test and run these programs, I am using Cris's dev box (varun@172.17.136.96 at ~/go-addb/). The hosts for the databases are listed in each program when setting the environment variable.

Run

```
go run filename.go
```

Validation

To validate smoothing and clustering results, compare the output of this system to the output of Brian Reed's addb system. In order to do so, run matchPreProcessorTest.go with the appropriate query (same start_ts, end_ts, and channels as Brian's ~/home/ubuntu/config/as-run-test.yaml). By default, the query looks like:

Validation Query

```
"SELECT channel, match_idx, start_ts, end_ts, match_start_ts,
match_end_ts FROM cd_match_log WHERE len >= 5.0 AND start_ts >=
'2019-02-05 01:00:00' AND end_ts <= '2019-02-05 06:00:00' AND channel in
('TBSHD','CNNHD', 'WABCDT') ORDER BY start_ts ASC"
```

To run Brian's addb in his dev box, traverse ~/home/ubuntu/addb/py/workshop/ and:

Brian addb

```
python test_as_run.py
```

His output is located in ~/home/ubuntu/tmp/addb/

This test performs the exact same functions in terms of smoothing and clustering, and tests based on a specific subset of Redshift match logs. This means that it is testing historic, not realtime, data.

Comparison in output can be seen through debugging in the logs, more specifically by looking at 1) the number of blocks created after every step (merge, fill, concat, fix boundary jitters) in smoothing and 2) the number of found vs. created clusters in clustering. Note that for clustering, the number of found vs. created clusters can be different than Brian's output due to a differing order of incoming batches and matches, but it should be close. To take a closer look at the specifics of the smoothed matches and clusters, add more debug statements and look at the MySQL tables.

Next Steps

The next step is to delete data from the Redis and MySQL databases (asrun_smoothed_match, asrun_cluster, and MatchSetID map) that is considered to be old.