.. \_real\_time:

Real-Time aggregate features

============================

In addition to computing batch aggregate features, the aggregation framework supports real-time aggregates as well. The framework concepts used here are identical to the batch use case, however, the underlying implementation differs and is provided by summingbird-storm jobs.

RTA Runbook

-----------

For operational details, please visit http://go/tqrealtimeaggregates.

Prerequisites

-------------

In order to start computing real-time aggregate features, the framework requires the following to be provided:

\* A backing memcached store that will hold the computed aggregate features. This is conceptually equivalent to the output HDFS store in the batch compute case.

\* Implementation of `StormAggregateSource <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/heron/StormAggregateSource.scala#n15>`\_ that creates `DataRecords` with the necessary input features. This serves as the input to the aggregation operations.

\* Definition of aggregate features by defining `AggregateGroup` in an implementation of `OnlineAggregationConfigTrait`. This is identical to the batch case.

\* Job config file defining the backing memcached for feature storage and retrieval, and job-related parameters.

We will now go through the details in setting up each required component.

Memcached store

---------------

Real-time aggregates use Memcache as the backing cache to store and update aggregate features keys. Caches can be provisioned on `go/cacheboard <https://cacheboardv2--prod--cache.service.atla.twitter.biz/>`\_.

.. admonition:: Test and prod caches

For development, it is sufficient to setup a test cache that your new job can query and write to. At the same time, a production cache request should also be submitted as these generally have significant lead times for provisioning.

StormAggregateSource

--------------------

To enable aggregation of your features, we need to start with defining a `StormAggregateSource` that builds a `Producer[Storm, DataRecord]`. This summingbird producer generates `DataRecords` that contain the input features and labels that the real-time aggregate job will compute aggregate features on. Conceptually, this is equivalent to the input data set in the offline batch use case.

.. admonition:: Example

If you are planning to aggregate on client engagements, you would need to subscribe to the `ClientEvent` kafka stream and then convert each event to a `DataRecord` that contains the key and the engagement on which to aggregate.

Typically, we would setup a julep filter for the relevant client events that we would like to aggregate on. This gives us a `Producer[Storm, LogEvent]` object which we then convert to `Producer[Storm, DataRecord]` with adapters that we wrote:

.. code-block:: scala

lazy val clientEventProducer: Producer[Storm, LogEvent] =

ClientEventSourceScrooge(

appId = AppId(jobConfig.appId),

topic = "julep\_client\_event\_suggests",

resumeAtLastReadOffset = false

).source.name("timelines\_events")

lazy val clientEventWithCachedFeaturesProducer: Producer[Storm, DataRecord] = clientEventProducer

.flatMap(mkDataRecords)

Note that this way of composing the storm graph gives us flexiblity in how we can hydrate input features. If you would like to join more complex features to `DataRecord`, you can do so here with additional storm components which can implement cache queries.

.. admonition:: Timelines Quality use case

In Timelines Quality, we aggregate client engagements on `userId` or `tweetId` and implement

`TimelinesStormAggregateSource <https://cgit.twitter.biz/source/tree/src/scala/com/twitter/timelines/prediction/common/aggregates/real\_time/TimelinesStormAggregateSource.scala>`\_. We create

`Producer[Storm,LogEvent]` of Timelines engagements to which we apply `ClientLogEventAdapter <https://cgit.twitter.biz/source/tree/src/scala/com/twitter/timelines/prediction/adapters/client\_log\_event/ClientLogEventAdapter.scala>`\_ which converts the event to `DataRecord` containing `userId`, `tweetId`, `timestampFeature` of the engagement and the engagement label itself.

.. admonition:: MagicRecs use case

MagicRecs has a very similar setup for real-time aggregate features. In addition, they also implement a more complex cache query to fetch the user's history in the `StormAggregateSource` for each observed client engagement to hydrate a richer set of input `DataRecords`:

.. code-block:: scala

val userHistoryStoreService: Storm#Service[Long, History] =

Storm.service(UserHistoryReadableStore)

val clientEventDataRecordProducer: Producer[Storm, DataRecord] =

magicRecsClientEventProducer

.flatMap { ...

(userId, logEvent)

}.leftJoin(userHistoryStoreService)

.flatMap {

case (\_, (logEvent, history)) =>

mkDataRecords(LogEventHistoryPair(logEvent, history))

}

.. admonition:: EmailRecs use case

EmailRecs shares the same cache as MagicRecs. They combine notification scribe data with email history data to identify the particular item a user engaged with in an email:

.. code-block:: scala

val emailHistoryStoreService: Storm#Service[Long, History] =

Storm.service(EmailHistoryReadableStore)

val emailEventDataRecordProducer: Producer[Storm, DataRecord] =

emailEventProducer

.flatMap { ...

(userId, logEvent)

}.leftJoin(emailHistoryStoreService)

.flatMap {

case (\_, (scribe, history)) =>

mkDataRecords(ScribeHistoryPair(scribe, history))

}

Aggregation config

------------------

The real-time aggregation config is extended from `OnlineAggregationConfigTrait <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/heron/OnlineAggregationConfigTrait.scala>`\_ and defines the features to aggregate and the backing memcached store to which they will be written.

Setting up real-time aggregates follows the same rules as in the offline batch use case. The major difference here is that `inputSource` should point to the `StormAggregateSource` implementation that provides the `DataRecord` containing the engagements and core features on which to aggregate. In the offline case, this would have been an `OfflineAggregateSource` pointing to an offline source of daily records.

Finally, `RealTimeAggregateStore` defines the backing memcache to be used and should be provided here as the `outputStore`.

.. NOTE::

Please make sure to provide an `AggregateGroup` for both staging and production. The main difference should be the `outputStore` where features in either environment are read from and written to. You want to make sure that a staged real-time aggregates summingbird job is reading/writing only to the test memcache store and does not mutate the production store.

Job config

----------

In addition to the aggregation config that defines the features to aggregate, the final piece we need to provide is a `RealTimeAggregatesJobConfig` that specificies job values such as `appId`, `teamName` and counts for the various topology components that define the capacity of the job (`Timelines example <https://cgit.twitter.biz/source/tree/src/scala/com/twitter/timelines/prediction/common/aggregates/real\_time/TimelinesRealTimeAggregatesJob.scala#n22>`\_).

Once you have the job config, implementing the storm job itself is easy and almost as concise as in the batch use case:

.. code-block:: scala

object TimelinesRealTimeAggregatesJob extends RealTimeAggregatesJobBase {

override lazy val statsReceiver = DefaultStatsReceiver.scope("timelines\_real\_time\_aggregates")

override lazy val jobConfigs = TimelinesRealTimeAggregatesJobConfigs

override lazy val aggregatesToCompute = TimelinesOnlineAggregationConfig.AggregatesToCompute

}

.. NOTE::

There are some topology settings that are currently hard-coded. In particular, we enable `Config.TOPOLOGY\_DROPTUPLES\_UPON\_BACKPRESSURE` to be true for added robustness. This may be made user-definable in the future.

Steps to hydrate RTAs

--------------------

1. Make the changes to RTAs and follow the steps for `Running the topology`.

2. Register the new RTAs to feature store. Sample phab: https://phabricator.twitter.biz/D718120

3. Wire the features from feature store to TLX. This is usually done with the feature switch set to False. So it's just a code change and will not yet start hydrating the features yet. Merge the phab. Sample phab: https://phabricator.twitter.biz/D718424

4. Now we hydrate the features to TLX gradually by doing it shard wise. For this, first create a PCM and then enable the hydration. Sample PCM: https://jira.twitter.biz/browse/PCM-147814

Running the topology

--------------------

0. For phab that makes change to the topology (such as adding new ML features), before landing the phab, please create a PCM (`example <https://jira.twitter.biz/browse/PCM-131614>`\_) and deploy the change to devel topology first and then prod (atla and pdxa). Once it is confirmed that the prod topology can handle the change, the phab can be landed.

1. Go to https://ci.twitter.biz/job/tq-ci/build

2. In `commands` input

.. code-block:: bash

. src/scala/com/twitter/timelines/prediction/common/aggregates/real\_time/deploy\_local.sh [devel|atla|pdxa]

One can only deploy either `devel`, `atla` (prod atla), `pdxa` (prod pdxa) at a time.

For example, to deploy both pdxa and atla prod topologies, one needs to build/run the above steps twice, one with `pdxa` and the other with `atla`.

The status and performance stats of the topology are found at `go/heron-ui <http://heron-ui-new--prod--heron.service.pdxa.twitter.biz/topologies>`\_. Here you can view whether the job is processing tuples, whether it is under any memory or backpressure and provides general observability.

Finally, since we enable `Config.TOPOLOGY\_DROPTUPLES\_UPON\_BACKPRESSURE` by default in the topology, we also need to monitor and alert on the number of dropped tuples. Since this is a job generating features a small fraction of dropped tuples is tolerable if that enables us to avoid backpressure that would hold up global computation in the entire graph.

Hydrating Real-Time Aggregate Features

--------------------------------------

Once the job is up and running, the aggregate features will be accessible in the backing memcached store. To access these features and hydrate to your online pipeline, we need to build a Memcache client with the right query key.

.. admonition:: Example

Some care needs to be taken to define the key injection and codec correctly for the memcached store. These types do not change and you can use the Timelines `memcache client builder <https://cgit.twitter.biz/source/tree/timelinemixer/common/src/main/scala/com/twitter/timelinemixer/clients/real\_time\_aggregates\_cache/RealTimeAggregatesMemcacheBuilder.scala>`\_ as an example.

Aggregate features are written to store with a `(AggregationKey, BatchID)` key.

`AggregationKey <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/AggregationKey.scala#n31>`\_ is an instant of the keys that you previously defined in `AggregateGroup`. If your aggregation key is `USER\_ID`, you would need to instantiate `AggregationKey` with the `USER\_ID` featureId and the userId value.

.. admonition:: Returned features

The `DataRecord` that is returned by the cache now contains all real-time aggregate features for the query `AggregationKey` (similar to the batch use case). If your online hydration flow produces data records, the real-time aggregate features can be joined with your existing records in a straightforward way.

Adding features from Feature Store to RTA

--------------------------------------------

To add features from Feature Store to RTA and create real time aggregated features based on them, one needs to follow these steps:

\*\*Step 1\*\*

Copy Strato column for features that one wants to explore and add a cache if needed. See details at `Customize any Columns for your Team as Needed <https://docbird.twitter.biz/ml\_feature\_store/productionisation-checklist.html?highlight=manhattan#customize-any-columns-for-your-team-as-needed>`\_. As an `example <https://phabricator.twitter.biz/D441050>`\_, we copy Strato column of recommendationsUserFeaturesProd.User.strato and add a cache for timelines team's usage.

\*\*Step 2\*\*

Create a new ReadableStore which uses Feature Store Client to request features from Feature Store. Implement FeaturesAdapter which extends TimelinesAdapterBase and derive new features based on raw features from Feature Store. As an `example <https://phabricator.twitter.biz/D458168>`\_, we create UserFeaturesReadableStore which reads discrete feature user state, and convert it to a list of boolean user state features.

\*\*Step 3\*\*

Join these derived features from Feature Store to timelines storm aggregate source. Depends on the characteristic of these derived features, joined key could be tweet id, user id or others. As an `example <https://phabricator.twitter.biz/D454408>`\_, because user state is per user, the joined key is user id.

\*\*Step 4\*\*

Define `AggregateGroup` based on derived features in RTA

Adding New Aggregate Features from an Existing Dataset

--------------------------------

To add a new aggregate feature group from an existing dataset for use in home models, use the following steps:

1. Identify the hypothesis being tested by the addition of the features, in accordance with `go/tpfeatureguide <http://go/tpfeatureguide>`\_.

2. Modify or add a new AggregateGroup to `TimelinesOnlineAggregationConfigBase.scala <https://sourcegraph.twitter.biz/git.twitter.biz/source/-/blob/src/scala/com/twitter/timelines/prediction/common/aggregates/real\_time/TimelinesOnlineAggregationConfigBase.scala>`\_ to define the aggregation key, set of features, labels and metrics. An example phab to add more halflives can be found at `D204415 <https://phabricator.twitter.biz/D204415>`\_.

3. If the change is expected to be very large, it may be recommended to perform capacity estimation. See :ref:`Capacity Estimation` for more details.

4. Create feature catalog items for the new RTAs. An example phab is `D706348 <https://phabricator.twitter.biz/D706438>`\_. For approval from a featurestore owner ping #help-ml-features on slack.

5. Add new features to the featurestore. An example phab is `D706112 <https://phabricator.twitter.biz/D706112>`\_. This change can be rolled out with feature switches or by canarying TLX, depending on the risk. An example PCM for feature switches is: `PCM-148654 <https://jira.twitter.biz/browse/PCM-148654>`\_. An example PCM for canarying is: `PCM-145753 <https://jira.twitter.biz/browse/PCM-145753>`\_.

6. Wait for redeploy and confirm the new features are available. One way is querying in BigQuery from a table like `twitter-bq-timelines-prod.continuous\_training\_recap\_fav`. Another way is to inspect individual records using pcat. The command to be used is like:

.. code-block:: bash

java -cp pcat-deploy.jar:$(hadoop classpath) com.twitter.ml.tool.pcat.PredictionCatTool

-path /atla/proc2/user/timelines/processed/suggests/recap/continuous\_training\_data\_records/fav/data/YYYY/MM/DD/01/part-00000.lzo

-fc /atla/proc2/user/timelines/processed/suggests/recap/continuous\_training\_data\_records/fav/data\_spec.json

-dates YYYY-MM-DDT01 -record\_limit 100 | grep [feature\_group]

7. Create a phab with the new features and test the performance of a model with them compared to a control model without them. Test offline using `Deepbird for training <https://docbird.twitter.biz/tq\_gcp\_guide/deepbird.html to train>`\_ and `RCE Hypothesis Testing <https://docbird.twitter.biz/Timelines\_Deepbird\_v2/training.html#model-evaluation-rce-hypothesis-testing>`\_ to test. Test online using a DDG. Some helpful instructions are available in `Serving Timelines Models <https://docbird.twitter.biz/timelines\_deepbird\_v2/serving.html>`\_ and the `Experiment Cookbook <https://docs.google.com/document/d/1FTaqd\_XOzdTppzePeipLhAgYA9hercN5a\_SyQXbuGws/edit#>`\_

Capacity Estimation

--------------------------------

This section describes how to approximate the capacity required for a new aggregate group. It is not expected to be exact, but should give a rough estimate.

There are two main components that must be stored for each aggregate group.

Key space: Each AggregationKey struct consists of two maps, one of which is populated with tuples [Long, Long] representing <featureId, value> of discrete features. This takes up 4 x 8 bytes or 32 bytes. The cache team estimates an additional 40 bytes of overhead.

Features: An aggregate feature is represented as a <Long, Double> pair (16 bytes) and is produced for each feature x label x metric x halflife combination.

1. Use bigquery to estimate how many unique values exist for the selected key (key\_count). Also collect the number of features, labels, metrics, and half-lives being used.

2. Compute the number of entries to be created, which is num\_entires = feature\_count \* label\_count \* metric\_count \* halflife\_count

3. Compute the number of bytes per entry, which is num\_entry\_bytes = 16\*num\_entries + 32 bytes (key storage) + 40 bytes (overhead)

4. Compute total space required = num\_entry\_bytes \* key\_count

Debugging New Aggregate Features

--------------------------------

To debug problems in the setup of your job, there are several steps you can take.

First, ensure that data is being received from the input stream and passed through to create data records. This can be achieved by logging results at various places in your code, and especially at the point of data record creation.

For example, suppose you want to ensure that a data record is being created with

the features you expect. With push and email features, we find that data records

are created in the adaptor, using logic like the following:

.. code-block:: scala

val record = new SRichDataRecord(new DataRecord)

...

record.setFeatureValue(feature, value)

To see what these feature values look like, we can have our adaptor class extend

Twitter's `Logging` trait, and write each created record to a log file.

.. code-block:: scala

class MyEventAdaptor extends TimelinesAdapterBase[MyObject] with Logging {

...

...

def mkDataRecord(myFeatures: MyFeatures): DataRecord = {

val record = new SRichDataRecord(new DataRecord)

...

record.setFeatureValue(feature, value)

logger.info("data record xyz: " + record.getRecord.toString)

}

This way, every time a data record is sent to the aggregator, it will also be

logged. To inspect these logs, you can push these changes to a staging instance,

ssh into that aurora instance, and grep the `log-files` directory for `xyz`. The

data record objects you find should resemble a map from feature ids to their

values.

To check that steps in the aggregation are being performed, you can also inspect the job's topology on go/heronui.

Lastly, to verify that values are being written to your cache you can check the `set` chart in your cache's viz.

To check particular feature values for a given key, you can spin up a Scala REPL like so:

.. code-block:: bash

$ ssh -fN -L\*:2181:sdzookeeper-read.atla.twitter.com:2181 -D \*:50001 nest.atlc.twitter.com

$ ./pants repl --jvm-repl-scala-options='-DsocksProxyHost=localhost -DsocksProxyPort=50001 -Dcom.twitter.server.resolverZkHosts=localhost:2181' timelinemixer/common/src/main/scala/com/twitter/timelinemixer/clients/real\_time\_aggregates\_cache

You will then need to create a connection to the cache, and a key with which to query it.

.. code-block:: scala

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.{DefaultStatsReceiver, StatsReceiver}

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.AggregationKey

import com.twitter.summingbird.batch.Batcher

import com.twitter.timelinemixer.clients.real\_time\_aggregates\_cache.RealTimeAggregatesMemcacheBuilder

import com.twitter.timelines.clients.memcache\_common.StorehausMemcacheConfig

val userFeature = -1887718638306251279L // feature id corresponding to User feature

val userId = 12L // replace with a user id logged when creating your data record

val key = (AggregationKey(Map(userFeature -> userId), Map.empty), Batcher.unit.currentBatch)

val dataset = "twemcache\_magicrecs\_real\_time\_aggregates\_cache\_staging" // replace with the appropriate cache name

val dest = s"/srv#/test/local/cache/twemcache\_/$dataset"

val statsReceiver: StatsReceiver = DefaultStatsReceiver

val cache = new RealTimeAggregatesMemcacheBuilder(

config = StorehausMemcacheConfig(

destName = dest,

keyPrefix = "",

requestTimeout = 10.seconds,

numTries = 1,

globalTimeout = 10.seconds,

tcpConnectTimeout = 10.seconds,

connectionAcquisitionTimeout = 10.seconds,

numPendingRequests = 250,

isReadOnly = true

),

statsReceiver.scope(dataset)

).build

val result = cache.get(key)

Another option is to create a debugger which points to the staging cache and creates a cache connection and key similar to the logic above.

Run CQL query to find metrics/counters

--------------------------------

We can also visualize the counters from our job to verify new features. Run CQL query on terminal to find the right path of metrics/counters. For example, in order to check counter mergeNumFeatures, run:

cql -z atla keys heron/summingbird\_timelines\_real\_time\_aggregates Tail-FlatMap | grep mergeNumFeatures

Then use the right path to create the viz, example: https://monitoring.twitter.biz/tiny/2552105