.. \_aggregation:

Core Concepts

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

This page provides an overview of the aggregation framework and goes through examples on how to define aggregate features. In general, we can think of an aggregate feature as a grouped set of records, on which we incrementally update the aggregate feature values, crossed by the provided features and conditional on the provided labels.

AggregateGroup

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

An `AggregateGroup` defines a single unit of aggregate computation, similar to a SQL query. These are executed by the underlying jobs (internally, a `DataRecordAggregationMonoid <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/DataRecordAggregationMonoid.scala#n42>`\_ is applied to `DataRecords` that contain the features to aggregate). Many of these groups can exist to define different types of aggregate features.

Let's start with the following examples of an `AggregateGroup` to discuss the meaning of each of its constructor arguments:

.. code-block:: scala

val UserAggregateStore = "user\_aggregates"

val aggregatesToCompute: Set[TypedAggregateGroup[\_]] = Set(

AggregateGroup(

inputSource = timelinesDailyRecapSource,

aggregatePrefix = "user\_aggregate\_v2",

preTransformOpt = Some(RemoveUserIdZero),

keys = Set(USER\_ID),

features = Set(HAS\_PHOTO),

labels = Set(IS\_FAVORITED),

metrics = Set(CountMetric, SumMetric),

halfLives = Set(50.days),

outputStore = OfflineAggregateStore(

name = UserAggregateStore,

startDate = "2016-07-15 00:00",

commonConfig = timelinesDailyAggregateSink,

batchesToKeep = 5

)

)

.flatMap(\_.buildTypedAggregateGroups)

)

This `AggregateGroup` computes the number of times each user has faved a tweet with a photo. The aggregate count is decayed with a 50 day halflife.

Naming and preprocessing

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

`UserAggregateStore` is a string val that acts as a scope of a "root path" to which this group of aggregate features will be written. The root path is provided separately by the implementing job.

`inputSource` defines the input source of `DataRecords` that we aggregate on. These records contain the relevant features required for aggregation.

`aggregatePrefix` tells the framework what prefix to use for the aggregate features it generates. A descriptive naming scheme with versioning makes it easier to maintain features as you add or remove them over the long-term.

`preTransforms` is a `Seq[com.twitter.ml.api.ITransform] <https://cgit.twitter.biz/source/tree/src/java/com/twitter/ml/api/ITransform.java>`\_ that can be applied to the data records read from the input source before they are fed into the `AggregateGroup` to apply aggregation. These transforms are optional but can be useful for certain preprocessing operations for a group's raw input features.

.. admonition:: Examples

You can downsample input data records by providing `preTransforms`. In addition, you could also join different input labels (e.g. "is\_push\_openend" and "is\_push\_favorited") and transform them into a combined label that is their union ("is\_push\_engaged") on which aggregate counts will be calculated.

Keys

----

`keys` is a crucial field in the config. It defines a `Set[com.twitter.ml.api.Feature]` which specifies a set of grouping keys to use for this `AggregateGroup`.

Keys can only be of 3 supported types currently: `DISCRETE`, `STRING` and `SPARSE\_BINARY`. Using a discrete or a string/text feature as a key specifies the unit to group records by before applying counting/aggregation operators.

.. admonition:: Examples

.. cssclass:: shortlist

#. If the key is `USER\_ID`, this tells the framework to group all records by `USER\_ID`, and then apply aggregations (sum/count/etc) within each user’s data to generate aggregate features for each user.

#. If the key is `(USER\_ID, AUTHOR\_ID)`, then the `AggregateGroup` will output features for each unique user-author pair in the input data.

#. Finally, using a sparse binary feature as key has special "flattening" or "flatMap" like semantics. For example, consider grouping by `(USER\_ID, AUTHOR\_INTEREST\_IDS)` where `AUTHOR\_INTEREST\_IDS` is a sparse binary feature which represents a set of topic IDs the author may be tweeting about. This creates one record for each `(user\_id, interest\_id)` pair - so each record with multiple author interests is flattened before feeding it to the aggregation.

Features

--------

`features` specifies a `Set[com.twitter.ml.api.Feature]` to aggregate within each group (defined by the keys specified earlier).

We support 2 types of `features`: `BINARY` and `CONTINUOUS`.

The semantics of how the aggregation works is slightly different based on the type of “feature”, and based on the “metric” (or aggregation operation):

.. cssclass:: shortlist

#. Binary Feature, Count Metric: Suppose we have a binary feature `HAS\_PHOTO` in this set, and are applying the “Count” metric (see below for more details on the metrics), with key `USER\_ID`. The semantics is that this computes a feature which measures the count of records with `HAS\_PHOTO` set to true for each user.

#. Binary Feature, Sum Metric - Does not apply. No feature will be computed.

#. Continuous Feature, Count Metric - The count metric treats all features as binary features ignoring their value. For example, suppose we have a continuous feature `NUM\_CHARACTERS\_IN\_TWEET`, and key `USER\_ID`. This measures the count of records that have this feature `NUM\_CHARACTERS\_IN\_TWEET` present.

#. Continuous Feature, Sum Metric - In the above example, the features measures the sum of (num\_characters\_in\_tweet) over all a user’s records. Dividing this sum feature by the count feature would give the average number of characters in all tweets.

.. admonition:: Unsupported feature types

`DISCRETE` and `SPARSE` features are not supported by the Sum Metric, because there is no meaning in summing a discrete feature or a sparse feature. You can use them with the CountMetric, but they may not do what you would expect since they will be treated as binary features losing all the information within the feature. The best way to use these is as “keys” and not as “features”.

.. admonition:: Setting includeAnyFeature

If constructor argument `includeAnyFeature` is set, the framework will append a feature with scope `any\_feature` to the set of all features you define. This additional feature simply measures the total count of records. So if you set your features to be equal to Set.empty, this will measure the count of records for a given `USER\_ID`.

Labels

------

`labels` specifies a set of `BINARY` features that you can cross with, prior to applying aggregations on the `features`. This essentially restricts the aggregate computation to a subset of the records within a particular key.

We typically use this to represent engagement labels in an ML model, in this case, `IS\_FAVORITED`.

In this example, we are grouping by `USER\_ID`, the feature is `HAS\_PHOTO`, the label is `IS\_FAVORITED`, and we are computing `CountMetric`. The system will output a feature for each user that represents the number of favorites on tweets having photos by this `userId`.

.. admonition:: Setting includeAnyLabel

If constructor argument `includeAnyLabel` is set (as it is by default), then similar to `any\_feature`, the framework automatically appends a label of type `any\_label` to the set of all labels you define, which represents not applying any filter or cross.

In this example, `any\_label` and `any\_feature` are set by default and the system would actually output 4 features for each `user\_id`:

.. cssclass:: shortlist

#. The number of `IS\_FAVORITED` (favorites) on tweet impressions having `HAS\_PHOTO=true`

#. The number of `IS\_FAVORITED` (favorites) on all tweet impressions (`any\_feature` aggregate)

#. The number of tweet impressions having `HAS\_PHOTO=true` (`any\_label` aggregate)

#. The total number of tweet impressions for this user id (`any\_feature.any\_label` aggregate)

.. admonition:: Disabling includeAnyLabel

To disable this automatically generated feature you can use `includeAnyLabel = false` in your config. This will remove some useful features (particularly for counterfactual signal), but it can greatly save on space since it does not store every possible impressed set of keys in the output store. So use this if you are short on space, but not otherwise.

Metrics

-------

`metrics` specifies the aggregate operators to apply. The most commonly used are `Count`, `Sum` and `SumSq`.

As mentioned before, `Count` can be applied to all types of features, but treats every feature as binary and ignores the value of the feature. `Sum` and `SumSq` can only be applied to Continuous features - they will ignore all other features you specify. By combining sum and sumsq and count, you can produce powerful “z-score” features or other distributional features using a post-transform.

It is also possible to add your own aggregate operators (e.g. `LastResetMetric <https://phabricator.twitter.biz/D228537>`\_) to the framework with some additional work.

HalfLives

---------

`halfLives` specifies how fast aggregate features should be decayed. It is important to note that the framework works on an incremental basis: in the batch implementation, the summingbird-scalding job takes in the most recently computed aggregate features, processed on data until day `N-1`, then reads new data records for day `N` and computes updated values of the aggregate features. Similarly, the decay of real-time aggregate features takes the actual time delta between the current time and the last time the aggregate feature value was updated.

The halflife `H` specifies how fast to decay old sums/counts to simulate a sliding window of counts. The implementation is such that it will take `H` amount of time to decay an aggregate feature to half its initial value. New observed values of sums/counts are added to the aggregate feature value.

.. admonition:: Batch and real-time

In the batch use case where aggregate features are recomputed on a daily basis, we typically take halflives on the order of weeks or longer (in Timelines, 50 days). In the real-time use case, shorter halflives are appropriate (hours) since they are updated as client engagements are received by the summingbird job.

SQL Equivalent

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

Conceptually, you can also think of it as:

.. code-block:: sql

INSERT INTO <outputStore>.<aggregatePrefix>

SELECT AGG(<features>) /\* AGG is <metrics>, which is a exponentially decaying SUM or COUNT etc. based on the halfLifves \*/

FROM (

SELECT preTransformOpt(\*) FROM <inputSource>

)

GROUP BY <keys>

WHERE <labels> = True

any\_features is AGG(\*).

any\_labels removes the WHERE clause.