.. \_joining:

Joining aggregates features to records

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

After setting up either offline batch jobs or online real-time summingbird jobs to produce

aggregate features and querying them, we are left with data records containing aggregate features.

This page will go over how to join them with other data records to produce offline training data.

(To discuss: joining aggregates to records online)

Joining Aggregates on Discrete/String Keys

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

Joining aggregate features keyed on discrete or text features to your training data is very easy -

you can use the built in methods provided by `DataSetPipe`. For example, suppose you have aggregates

keyed by `(USER\_ID, AUTHOR\_ID)`:

.. code-block:: scala

val userAuthorAggregates: DataSetPipe = AggregatesV2FeatureSource(

rootPath = “/path/to/my/aggregates”,

storeName = “user\_author\_aggregates”,

aggregates = MyConfig.aggregatesToCompute,

trimThreshold = 0

)(dateRange).read

Offline, you can then join with your training data set as follows:

.. code-block:: scala

val myTrainingData: DataSetPipe = ...

val joinedData = myTrainingData.joinWithLarger((USER\_ID, AUTHOR\_ID), userAuthorAggregates)

You can read from `AggregatesV2MostRecentFeatureSourceBeforeDate` in order to read the most recent aggregates

before a provided date `beforeDate`. Just note that `beforeDate` must be aligned with the date boundary so if

you’re passing in a `dateRange`, use `dateRange.end`).

Joining Aggregates on Sparse Binary Keys

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

When joining on sparse binary keys, there can be multiple aggregate records to join to each training record in

your training data set. For example, suppose you have setup an aggregate group that is keyed on `(INTEREST\_ID, AUTHOR\_ID)`

capturing engagement counts of users interested in a particular `INTEREST\_ID` for specific authors provided by `AUTHOR\_ID`.

Suppose now that you have a training data record representing a specific user action. This training data record contains

a sparse binary feature `INTEREST\_IDS` representing all the "interests" of that user - e.g. music, sports, and so on. Each `interest\_id`

translates to a different set of counting features found in your aggregates data. Therefore we need a way to merge all of

these different sets of counting features to produce a more compact, fixed-size set of features.

.. admonition:: Merge policies

To do this, the aggregate framework provides a trait `SparseBinaryMergePolicy <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/conversion/SparseBinaryMergePolicy.scala>`\_. Classes overriding this trait define policies

that state how to merge the individual aggregate features from each sparse binary value (in this case, each `INTEREST\_ID` for a user).

Furthermore, we provide `SparseBinaryMultipleAggregateJoin` which executes these policies to merge aggregates.

A simple policy might simply average all the counts from the individual interests, or just take the max, or

a specific quantile. More advanced policies might use custom criteria to decide which interest is most relevant and choose

features from that interest to represent the user, or use some weighted combination of counts.

The framework provides two simple in-built policies (`PickTopCtrPolicy <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/conversion/PickTopCtrPolicy.scala>`\_

and `CombineCountsPolicy <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ml\_util/aggregation\_framework/conversion/CombineCountsPolicy.scala>`\_, which keeps the topK counts per

record) that you can get started with, though you likely want to implement your own policy based on domain knowledge to get

the best results for your specific problem domain.

.. admonition:: Offline Code Example

The scalding job `TrainingDataWithAggV2Generator <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ad\_hoc/recap/training\_data\_generator/TrainingDataWithAggV2Generator.scala>`\_ shows how multiple merge policies are defined and implemented to merge aggregates on sparse binary keys to the TQ's training data records.

.. admonition:: Online Code Example

In our (non-FeatureStore enabled) online code path, we merge aggregates on sparse binary keys using the `CombineCountsPolicy <https://cgit.twitter.biz/source/tree/timelinemixer/server/src/main/scala/com/twitter/timelinemixer/injection/recapbase/aggregates/UserFeaturesHydrator.scala#n201>`\_.