**Map-side Join**

In a map-side (fragment-replicate) join, you hold one dataset in memory (in say a hash table) and join on the other dataset, record-by-record. In Pig, you'd write

edges\_from\_list = JOIN a\_follows\_b BY user\_a\_id, some\_list BY user\_id using 'replicated';

taking care that the smaller dataset is on the *right*. This is extremely efficient, as there is no network overhead and minimal CPU demand.

**Reduce Join**

In a reduce-side join, you group on the join key using hadoop's standard merge sort.

<user\_id {A, B, F, ..., Z}, { A, C, G, ..., Q} >

and emit a record for every pair of an element from the first set with an element from the second set:

[A user\_id A]

[A user\_id C]

...

[A user\_id Q]

...

[Z user\_id Q]

You should design your keys so that the dataset with the fewest records per key comes first -- you need to hold the first group in memory and stream the second one past it. In Pig, for a standard join you accomplish this by putting the *largest* dataset last. (As opposed to the fragment-replicate join, where the in-memory dataset is given last).

Note that for a map-side join the *entirety* of the smaller dataset must fit in memory. In a standard reduce-side join, only each key's groups must fit in memory (actually each key's group except the last one). It's possible to avoid even this restriction, but it requires care; look for example at the [skewed join](http://hadoop.apache.org/pig/docs/r0.7.0/piglatin_users.html#Skewed+Joins) in Pig.

**Merge Join**

Finally, if both datasets are stored in total-sorted order on the join key, you can do a merge join on the map side. Same as the reduce-side join, you do a merge sort to cogroup on the join key, and then project (flatten) back out on the pairs.

Because of this, when generating a frequently-read dataset it's often a good idea to do a total sort in the last pass. Zebra and other databases may also give you total-sorted input for (almost) free.

Both of these joins of Hadoop are merge joins, which require a (explicit) sorting beforehand. Hash join, on the other hand, do not require sorting, but partition data by some hash function. Detailed discussion can be found in section "Relational Joins" in [Data-Intensive Text Processing with MapReduce](http://lintool.github.io/MapReduceAlgorithms/) by Jimmy Lin and Chris Dyer, a well-written book that is free and open source.