Given a social network with tens of millions of users, in this chapter we'll implement a MapReduce program to identify "common friends" among all pairs of users. Let U be a set of all users: {U 1 , U 2 , ..., U n }. Our goal is to find common friends for every pair of (U i , U j) where i ≠ j.
These days most social network sites (such as Facebook, hi5, and LinkedIn) offer services to help you share messages, pictures, and videos among friends. Some sites even offer video chat services to help you connect with friends. By definition, a friend is a person whom one knows, likes, and trusts. For example, Facebook has your list of friends, and friend relationships are bidirectional on the site; if I'm your friend, you're mine too. Typically social networks precompute calculations when they can to reduce the processing time of requests, and one common processing request involves the "You and Mary (your friend) have 185 friends in common" feature. When you visit someone's profile, you see a list of friends that you have in common. This list doesn't change frequently, so it is wasteful for the site to recalculate it every time you visit that person's profile.

Input
We prepare input as a set of records, where each record has the following format:
<person><,><friend 1 ><friend 2 >...<friend N >
where <friend 1 ><friend 2 > ... <friend N > are friends of the <person> . Note that in real projects/applications, each person/friend will be identified as a unique user ID. A very simple and complete example of input is as follows:
100, 200 300 400 500 600
200, 100 300 400
300, 100 200 400 500
400, 100 200 300
500, 100 300
600, 100

In this example, user 500 has two friends identified by the user IDs 100 and 300, and user 600 has only one friend: user 100.

The mapper accepts a (key 1 , value 1) pair, where
key 1 is a person and value 1 is a list of associated friends of that person. The mapper emits a set of new (key 2 , value 2) pairs; key 2 is a Tuple2(key 1 , friend i) , where friend i ϵ value 1 , and value 2 is the same as value 1 (a list of all friends for key 1). The reducer's key is a pair of two users (User j , User k) and value is a list of sets of friends. The reduce() function will intersect all sets of friends to find common and mutual friends for the (User j , User k) pair.

```
1 // key is the person
2 // value is a list of friends for this key=person
3 // value = (<friend_1> <friend_2> ... <friend_N>)
4 map(key, value) {
5    reducerValue = (<friend_1>< friend_2> ...< friend_N>);
6    foreach friend in (<friend_1>< friend_2> ... <friend_N>) {
7       reducerKey = buildSortedKey(person, friend);
```

```
        emit(reducerKey, reducerValue);
    }
}
```

```
Tuple2 buildSortedKey(person1, person2) {
    if (person1 < person2) {
        return new Tuple2(person1, person2);
    }
    else {
      return new Tuple2(person2, person1);
    }
}
```

```
// key = Tuple2(person1, person2)
// value = List {List_1, List_2, ..., List_M}
//   where each List_i = { set of unique user IDs }
reduce(key, value) {
    outputKey = key;
    outputValue = intersection (List_1, List_1, ..., List_M);
    emit (outputKey, outputValue);
}
```

Mapreduce in action for given example

`map(100, (200 300 400 500 600))` will generate:

```
([100, 200], [200 300 400 500 600])
([100, 300], [200 300 400 500 600])
([100, 400], [200 300 400 500 600])
([100, 500], [200 300 400 500 600])
([100, 600], [200 300 400 500 600])
```

`map(200, (100 300 400))` will generate:

```
([100, 200], [100 300 400])
([200, 300], [100 300 400])
([200, 400], [100 300 400])
```

`map(300, (100 200 400 500))` will generate:

```
([100, 300], [100 200 400 500])
([200, 300], [100 200 400 500])
([300, 400], [100 200 400 500])
([300, 500], [100 200 400 500])
```

`map(400, (100 200 300))` will generate:

```
([100, 400], [100 200 300])
([200, 400], [100 200 300])
([300, 400], [100 200 300])
```

`map(500, (100 300))` will generate:

```
([100, 500], [100 300])
([300, 500], [100 300])
```

`map(600, (100))` will generate:

```
([100, 600], [100])
```

So, the mappers generate the following key-value pairs:

```
([100, 200], [200 300 400 500 600])
([100, 300], [200 300 400 500 600])
([100, 400], [200 300 400 500 600])
([100, 500], [200 300 400 500 600])
([100, 600], [200 300 400 500 600])
([100, 200], [100 300 400])
([200, 300], [100 300 400])
([200, 400], [100 300 400])
([100, 300], [100 200 400 500])
([200, 300], [100 200 400 500])
([300, 400], [100 200 400 500])
([300, 500], [100 200 400 500])
([100, 400], [100 200 300])
([200, 400], [100 200 300])
([300, 400], [100 200 300])
([100, 500], [100 300])
([300, 500], [100 300])
([100, 600], [100])
```

Before these key-value pairs are sent to the reducers, they are grouped by keys:

```
([100, 200], [200 300 400 500 600])
([100, 200], [100 300 400])
=> ([100, 200], ([200 300 400 500 600], [100 300 400]))

([100, 300], [200 300 400 500 600])
([100, 300], [100 200 400 500])
=> ([100, 300], ([200 300 400 500 600],[100 200 400 500]))
```

So, the reducers will receive the following set of key-value pairs:

```
([100, 200], ([200 300 400 500 600], [100 300 400]))
([100, 300], ([200 300 400 500 600],[100 200 400 500]))
([100, 400], ([200 300 400 500 600], [100 200 300]))
([100, 500], ([200 300 400 500 600], [100 300]))
([200, 300], ([100 300 400],[100 200 400 500]))
([200, 400], ([100 300 400],[100 200 300]))
([300, 400], ([100 200 400 500][100 200 300]))
([300, 500], ([100 200 400 500],[100 300]))
([100, 600], ([200 300 400 500 600], [100])
```

Finally, the reducers will generate:

```
([100, 200], [300, 400])
([100, 300], [200, 400, 500])
([100, 400], [200, 300])
([100, 500], [300])
([200, 300], [100, 400])
([200, 400], [100, 300])
([300, 400], [100, 200])
([300, 500], [100])
([100, 600], [])
```

Following the generated output, we can see that users 100 and 600 have no common friends. The business case is that when user 100 visits user 200's profile, we can now quickly look up the [100, 200] key and see that they have two friends in common: [300, 400] . Meanwhile, the users identified by 100 and 500 have one friend (identified by user ID 300) in common.