MapReduce Joins

Shalish.V.J

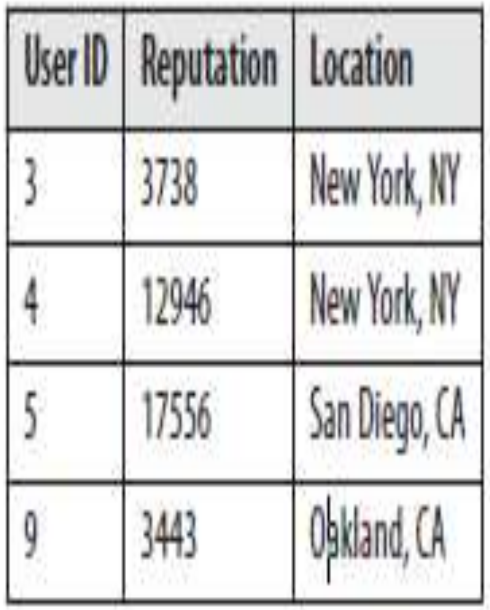
A Refresher on Joins

A join is an operation that combines records from two or more data sets based on a field or set of fields, known as the foreign key

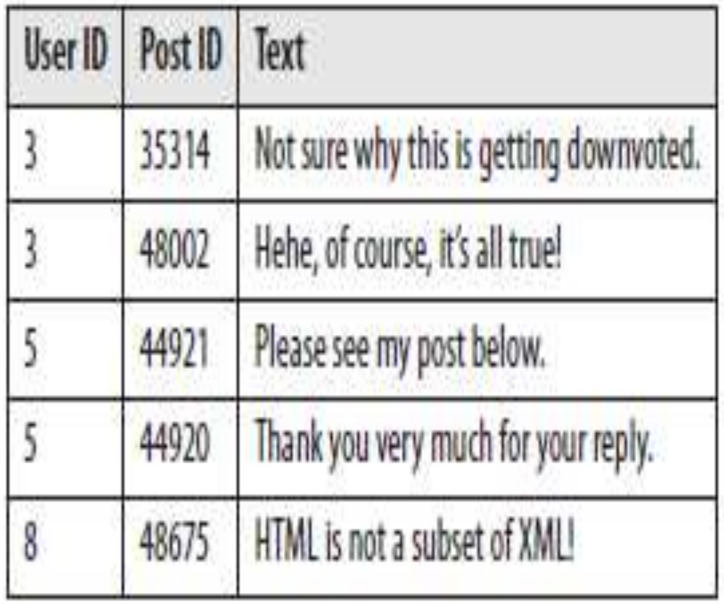
The foreign key is the field in a relational table that matches the column of another table

Sample Data Sets : A & B

Two data sets A and B, with the foreign key defined as f

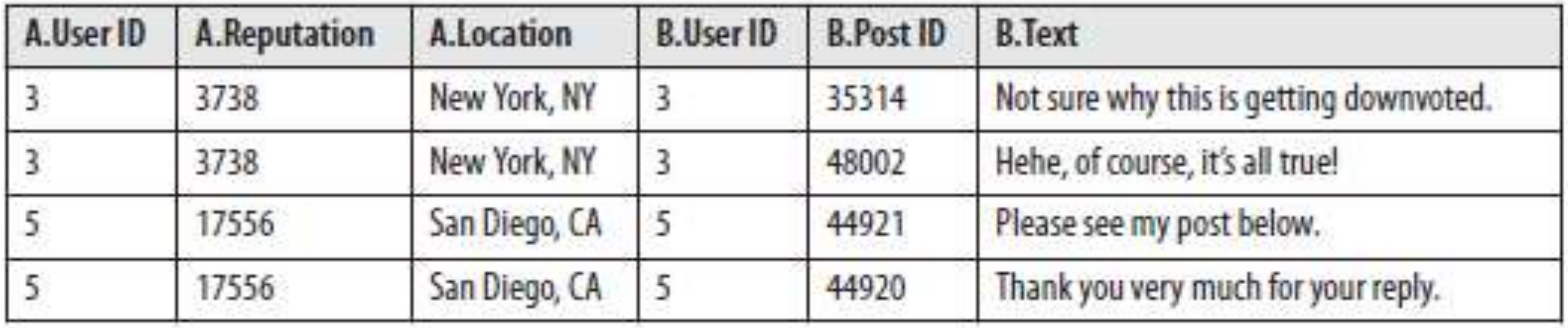


User ID	Reputation	Location
3	3738	New York, NY
4	12946	New York, NY
5	17556	San Diego, CA
9	3443	Oakland, CA



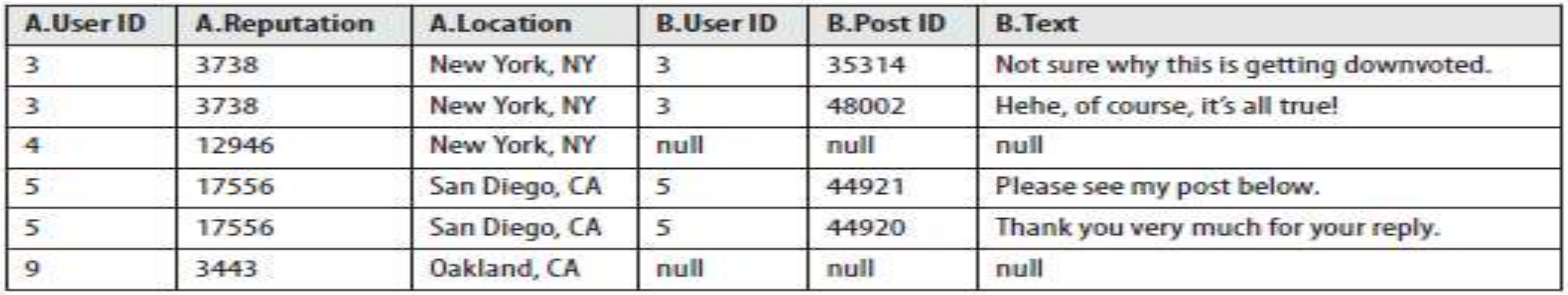
User ID	Post ID	Text
3	35314	Not sure why this is getting downvoted.
3	48002	Hehe, of course, it's all true!
5	44921	Please see my post below.
5	44920	Thank you very much for your reply.
8	48675	HTML is not a subset of XML!

Inner Join

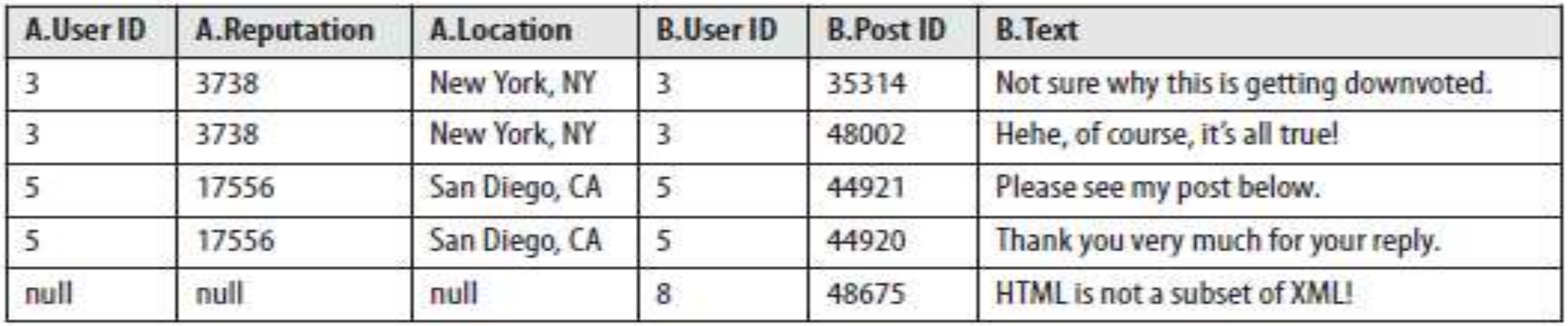


A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.

Outer Joins : Left & Right

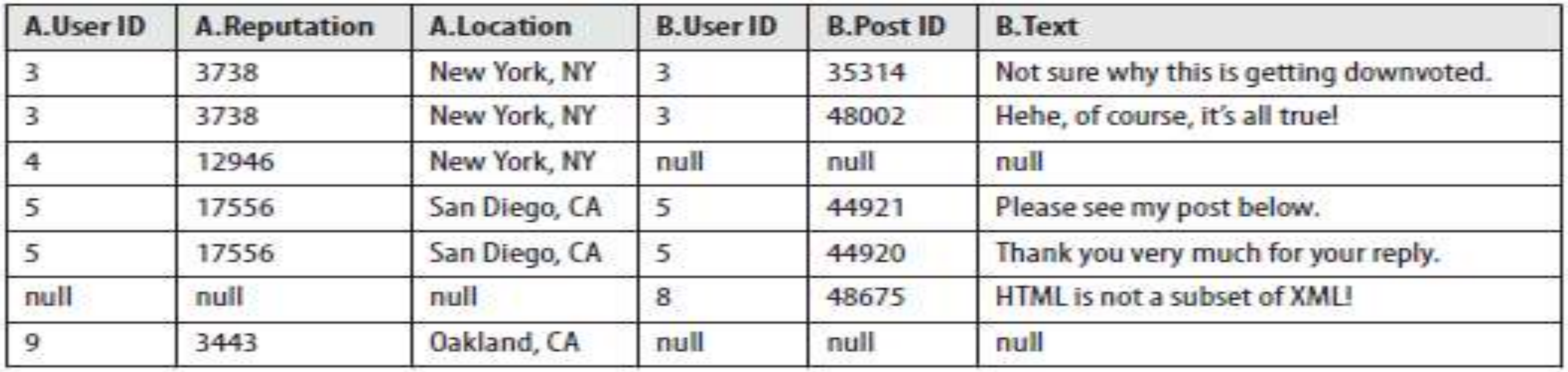


A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
4	12946	New York, NY	null	null	null
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.
9	3443	Oakland, CA	null	null	null

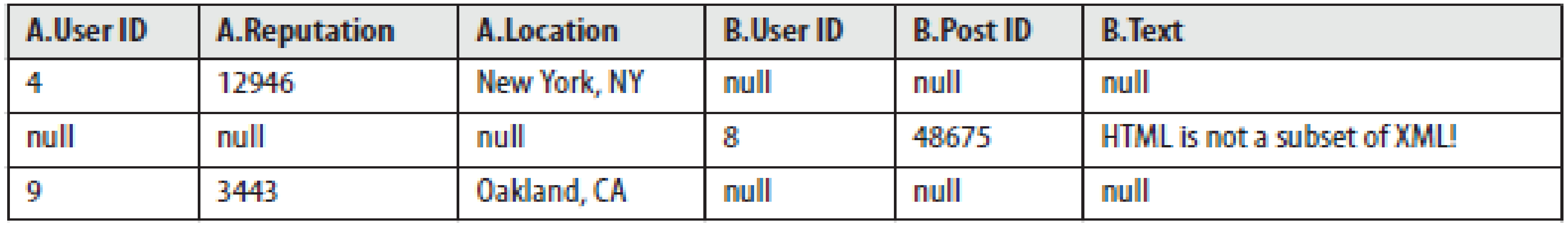


A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.
null	null	null	8	48675	HTML is not a subset of XML!

Full Outer & Anti Joins



A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
4	12946	New York, NY	null	null	null
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.
null	null	null	8	48675	HTML is not a subset of XML!
9	3443	Oakland, CA	null	null	null



A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
4	12946	New York, NY	null	null	null
null	null	null	8	48675	HTML is not a subset of XML!
9	3443	Oakland, CA	null	null	null

Reduce Side Join

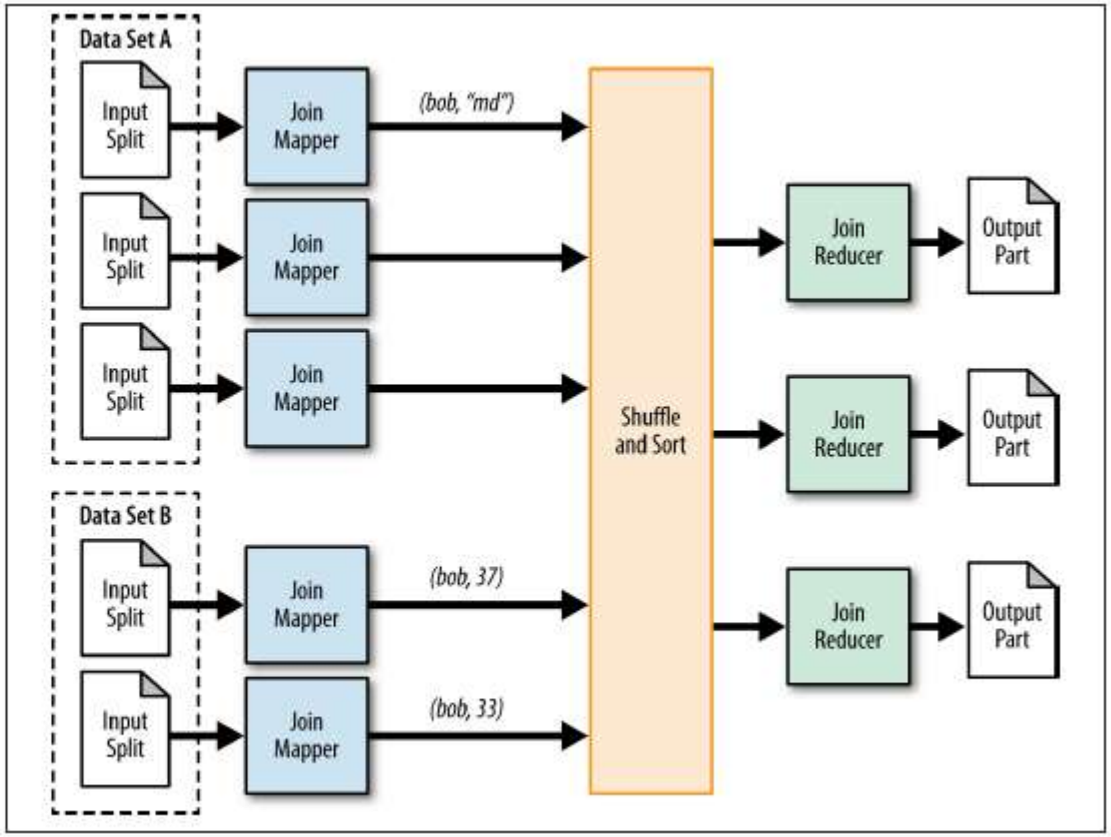
- Join large multiple data sets together by some foreign key
- Can be used to execute any of the types of joins
- No limitation on the size of data sets
- Require a large amount of network bandwidth

Reduce Side Join : Structure

- ❑ Mapper prepares the join operation
  - Takes each input record from each of the data sets
  - Extracts foreign key from each record
  - output key : foreign key
  - output value : entire input record which is flagged by some unique identifier for the data set

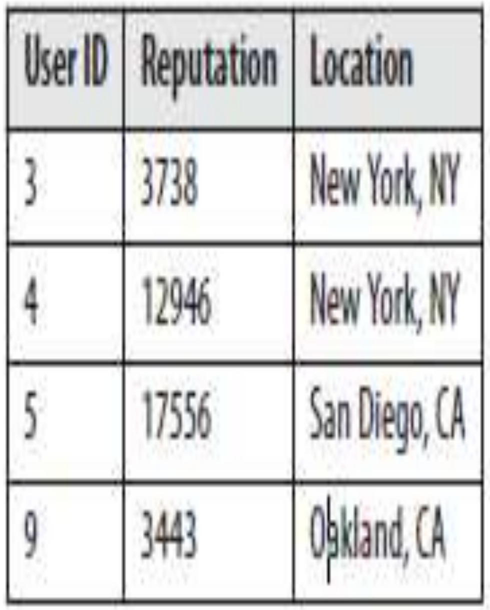
- ❑ Reducer performs the desired join operation by collecting the values of each input group into temporary lists
  - lists are then iterated over and the records from both sets are joined together
  - output is a number of part files equivalent to the number of reduce tasks

Reduce Side Join : Structure

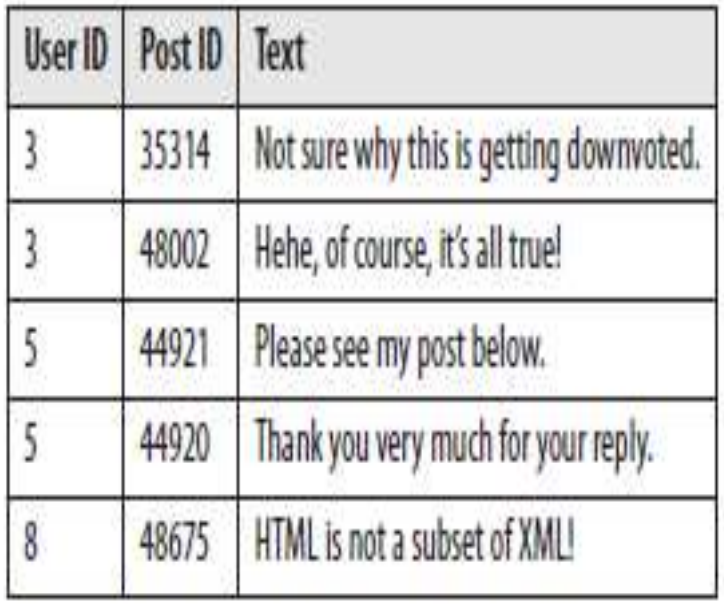


Sample Data Sets : A & B

Two data sets A and B, with the foreign key defined as f



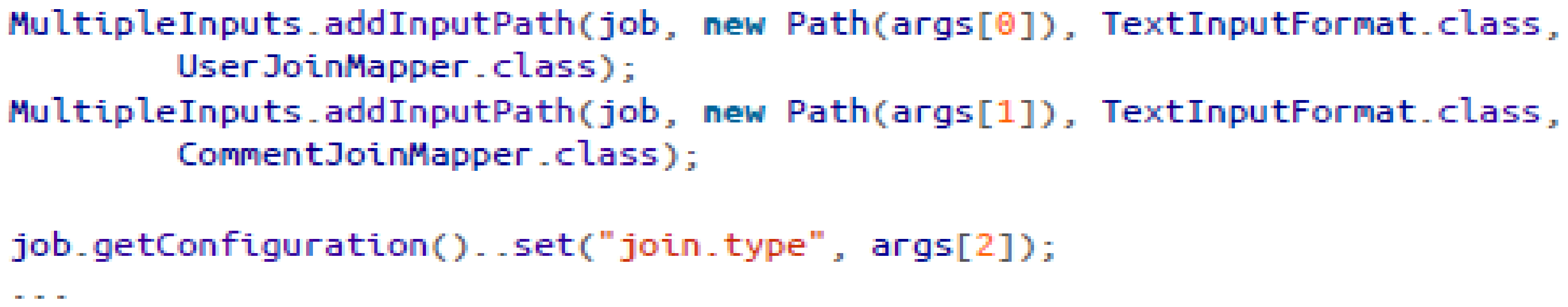
User ID	Reputation	Location
3	3738	New York, NY
4	12946	New York, NY
5	17556	San Diego, CA
9	3443	Oakland, CA



User ID	Post ID	Text
3	35314	Not sure why this is getting downvoted.
3	48002	Hehe, of course, it's all true!
5	44921	Please see my post below.
5	44920	Thank you very much for your reply.
8	48675	HTML is not a subset of XML!

Reduce Side Join : Driver Code

MultipleInputs data types : allows to create a mapper class and input format for different data sources

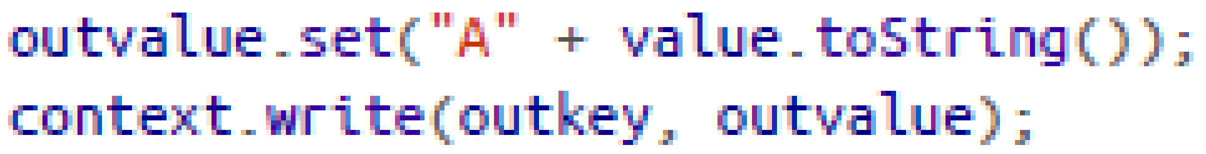


```
MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class,
        UserJoinMapper.class);
MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class,
        CommentJoinMapper.class);

job.getConfiguration()..set("join.type", args[2]);
...
```

Reduce Side Join : Mappers

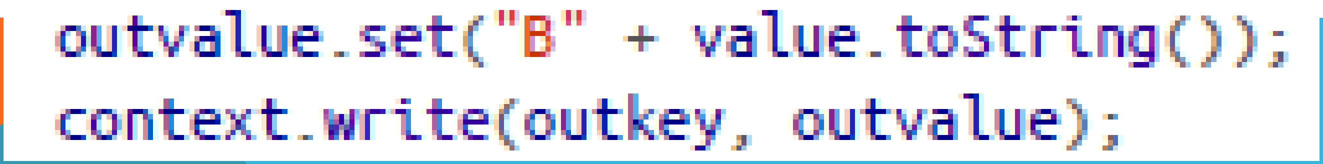
Each mapper class outputs the user ID as the foreign key and entire record as the value along with a single character to flag which record came from what set

UserJoinMapper



```
outvalue.set("A" + value.toString());
context.write(outkey, outvalue);
```

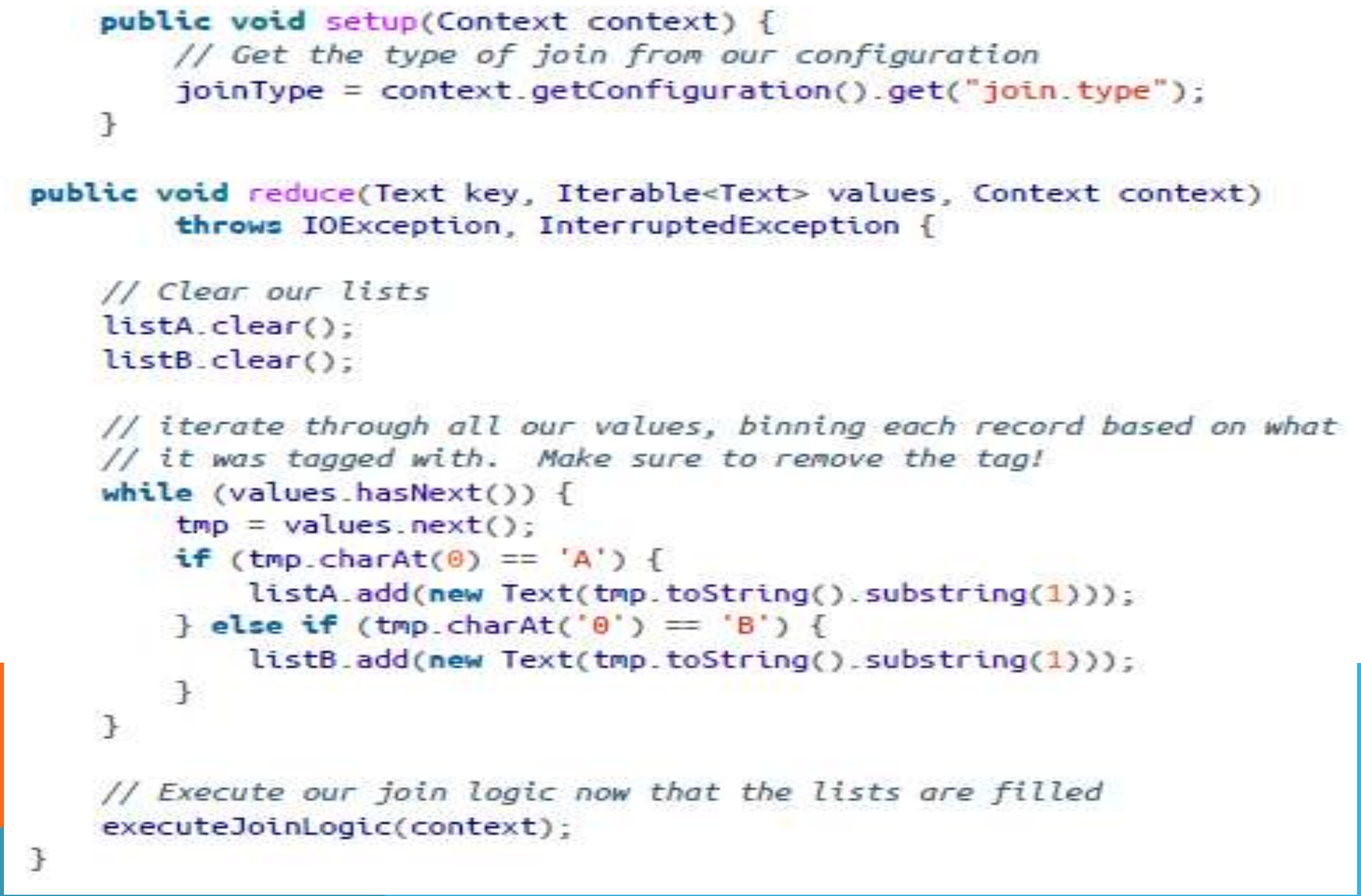
CommentJoinMapper



```
outvalue.set("B" + value.toString());
context.write(outkey, outvalue);
```

Reduce Side Join : Reducer

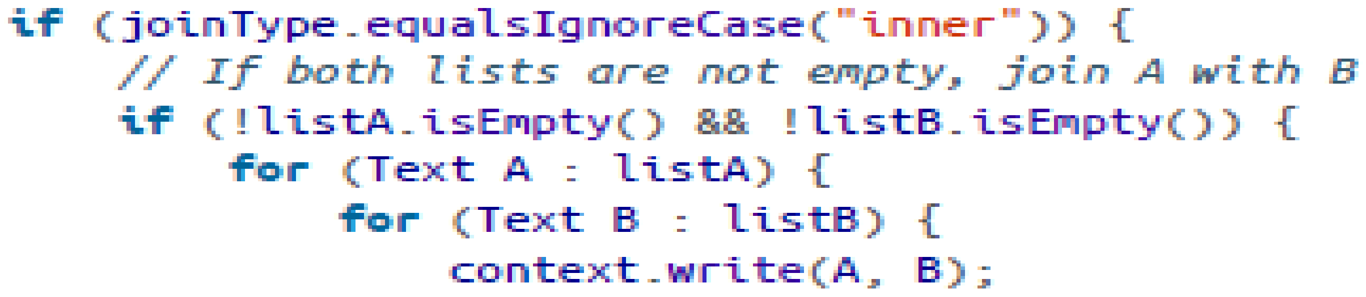
Reducer copies all values for each group in memory, keeping track of which record came from what data set



```
    public void setup(Context context) {
        // Get the type of join from our configuration
        joinType = context.getConfiguration().get("join.type");
    }

public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

    // Clear our lists
    listA.clear();
    listB.clear();

    // iterate through all our values, binning each record based on what
    // it was tagged with.  Make sure to remove the tag!
    while (values.hasNext()) {
        tmp = values.next();
        if (tmp.charAt(0) == 'A') {
            listA.add(new Text(tmp.toString().substring(1)));
        } else if (tmp.charAt('0') == 'B') {
            listB.add(new Text(tmp.toString().substring(1)));
        }
    }

    // Execute our join logic now that the lists are filled
    executeJoinLogic(context);
}
```

Reduce Side Join : Reducer – Inner Join

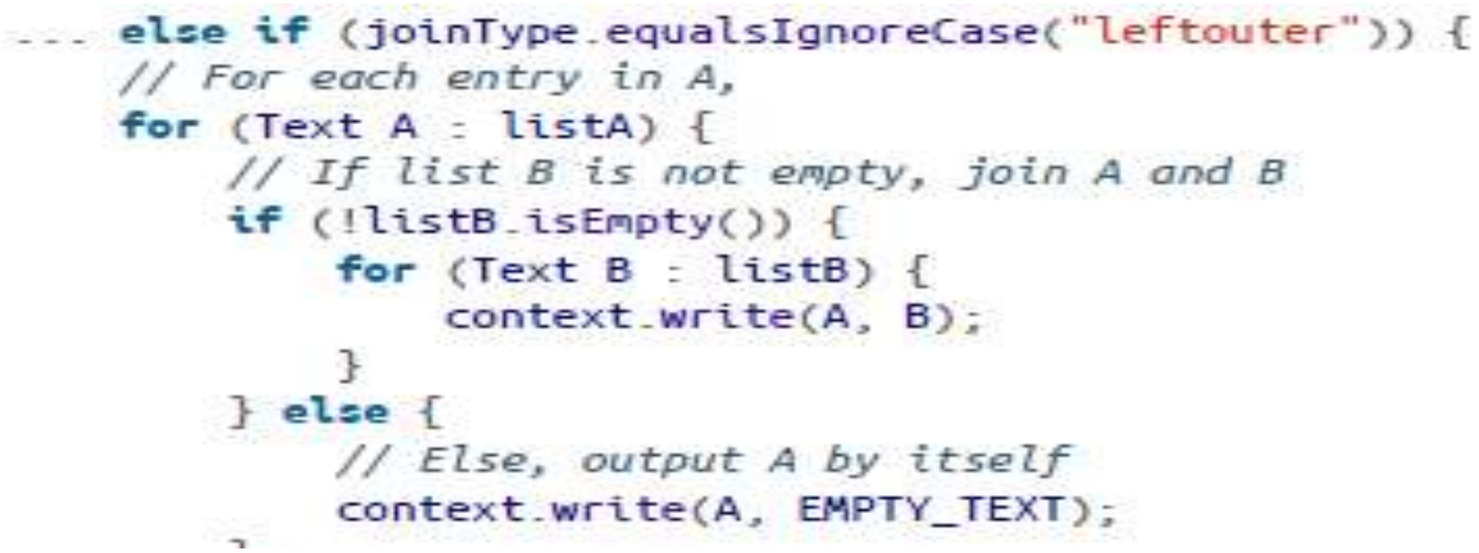
For an inner join, a joined record is output if all the lists are not empty



```
if (joinType.equalsIgnoreCase("inner")) {
    // If both lists are not empty, join A with B
    if (!listA.isEmpty() && !listB.isEmpty()) {
        for (Text A : listA) {
            for (Text B : listB) {
                context.write(A, B);
```

Reduce Side Join : Reducer – Left Outer Join

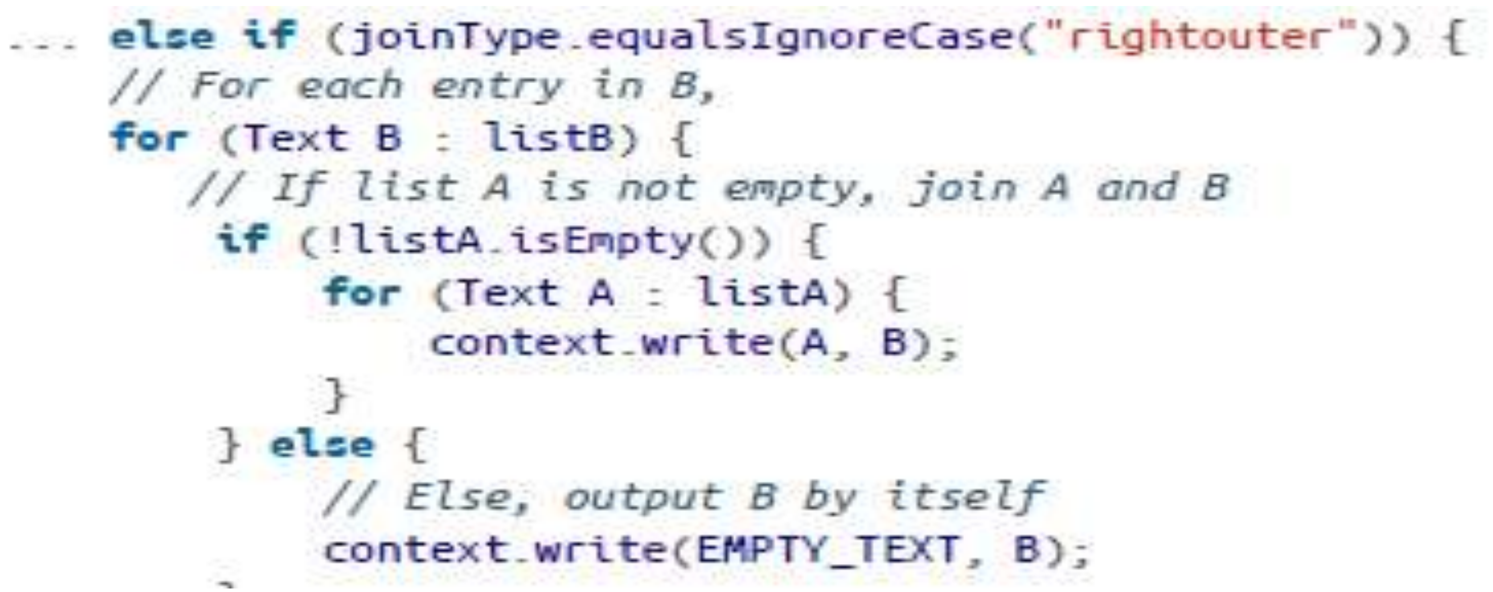
If the right list is not empty, join A with B.
If the right list is empty, output each record of A with an empty string.



```
... else if (joinType.equalsIgnoreCase("leftouter")) {
    // For each entry in A,
    for (Text A : listA) {
        // If list B is not empty, join A and B
        if (!listB.isEmpty()) {
            for (Text B : listB) {
                context.write(A, B);
            }
        } else {
            // Else, output A by itself
            context.write(A, EMPTY_TEXT);
```

Reduce Side Join : Reducer – Right Outer Join

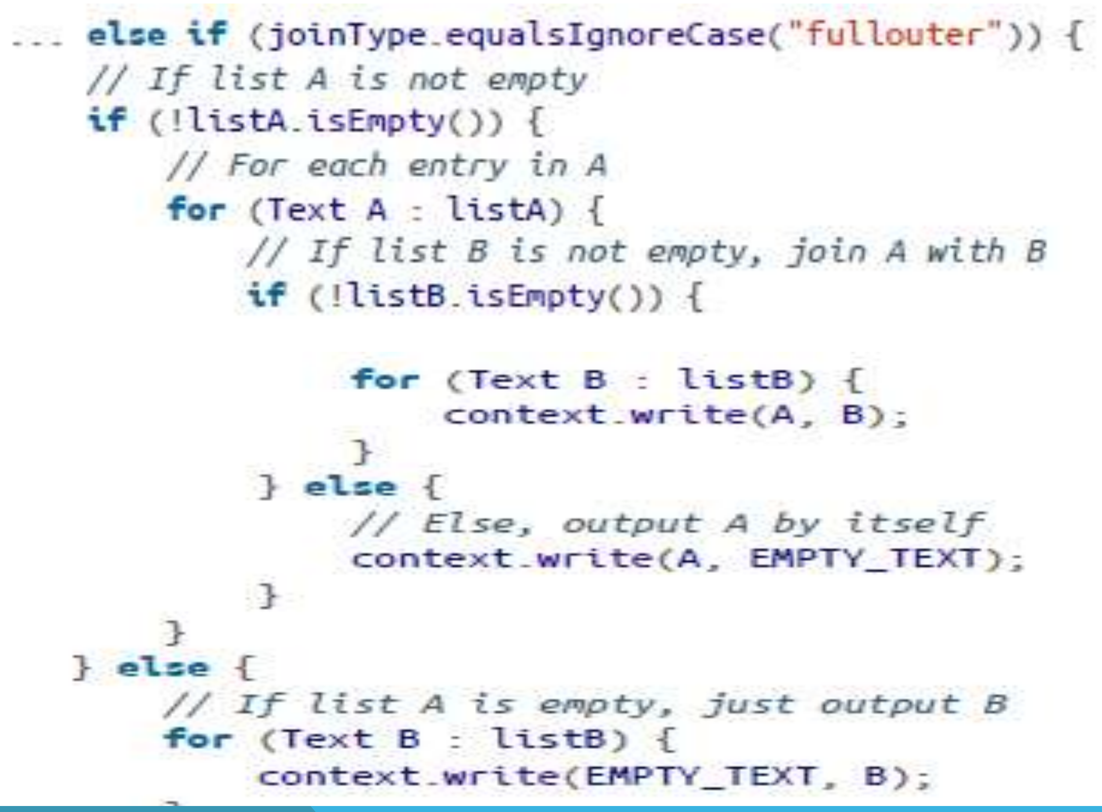
If the left list is not empty, join A with B.
If the left list is empty, output each record of A with an empty string.



```
... else if (joinType.equalsIgnoreCase("rightouter")) {
    // For each entry in B,
    for (Text B : listB) {
        // If list A is not empty, join A and B
        if (!listA.isEmpty()) {
            for (Text A : listA) {
                context.write(A, B);
            }
        } else {
            // Else, output B by itself
            context.write(EMPTY_TEXT, B);
```

Reduce Side Join : Reducer – Full Outer Join

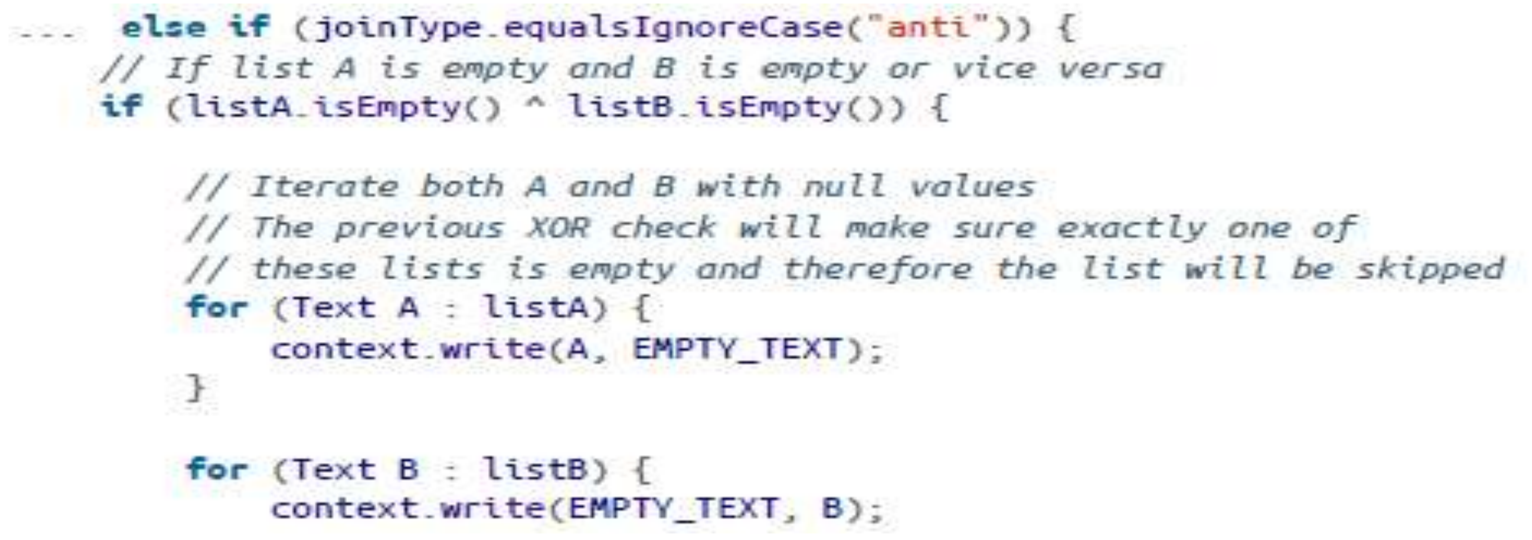
If list A is not empty, then for every element in A, join with B when the B list is not empty,
or output A by itself.
If A is empty, then just output B.



```
... else if (joinType.equalsIgnoreCase("fullouter")) {
    // If list A is not empty
    if (!listA.isEmpty()) {
        // For each entry in A
        for (Text A : listA) {
            // If list B is not empty, join A with B
            if (!listB.isEmpty()) {

                for (Text B : listB) {
                    context.write(A, B);
                }
            } else {
                // Else, output A by itself
                context.write(A, EMPTY_TEXT);
            }
        }
    } else {
        // If list A is empty, just output B
        for (Text B : listB) {
            context.write(EMPTY_TEXT, B);
```

Reduce Side Join : Reducer – AntiJoin

For an antijoin, if at least one of the lists is empty,
output the records from the nonempty list with an empty Text object.



```
...  else if (joinType.equalsIgnoreCase("anti")) {
    // If list A is empty and B is empty or vice versa
    if (listA.isEmpty() ^ listB.isEmpty()) {

        // Iterate both A and B with null values
        // The previous XOR check will make sure exactly one of
        // these lists is empty and therefore the list will be skipped
        for (Text A : listA) {
            context.write(A, EMPTY_TEXT);
        }

        for (Text B : listB) {
            context.write(EMPTY_TEXT, B);
```

Performance Analysis

A plain reduce side join puts a lot of strain on the cluster’s network

Replicated Join

- Join operation between one large and many small data sets that can be performed on the map-side
- Completely eliminates the need to shuffle any data to the reduce phase
- All the data sets except the very large one are essentially read into memory during the setup phase of each map task, which is limited by the JVM heap
- Join is done entirely in the map phase, with the very large data set being the input for the MapReduce job.
- Restriction : a replicated join is really useful only for an inner or a left outer join where the large data set is the “left” data set
- Output is a number of part files equivalent to the number of map tasks

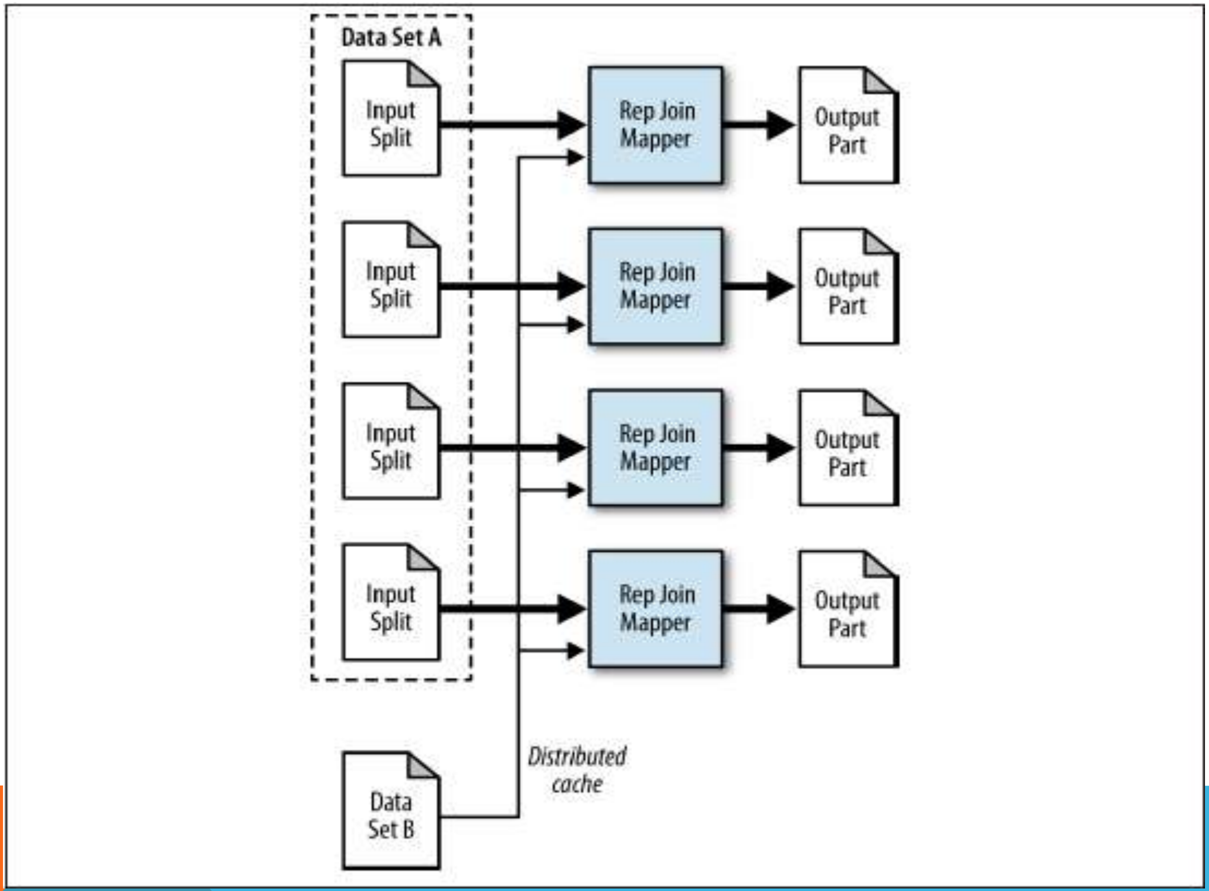
Replicated Join : Applicability

- ❑ The type of join to execute is an inner join or a left outer join, with the large input data set being the “left” part of the operation
- ❑ All of the data sets, except for the large one, can be fit into main memory of each map task

Replicated Join : Structure

- The mapper is responsible for reading all files from the distributed cache during the setup phase and storing them into in-memory lookup tables
- Mapper processes each record and joins it with all the data stored in-memory

Replicated Join : Structure

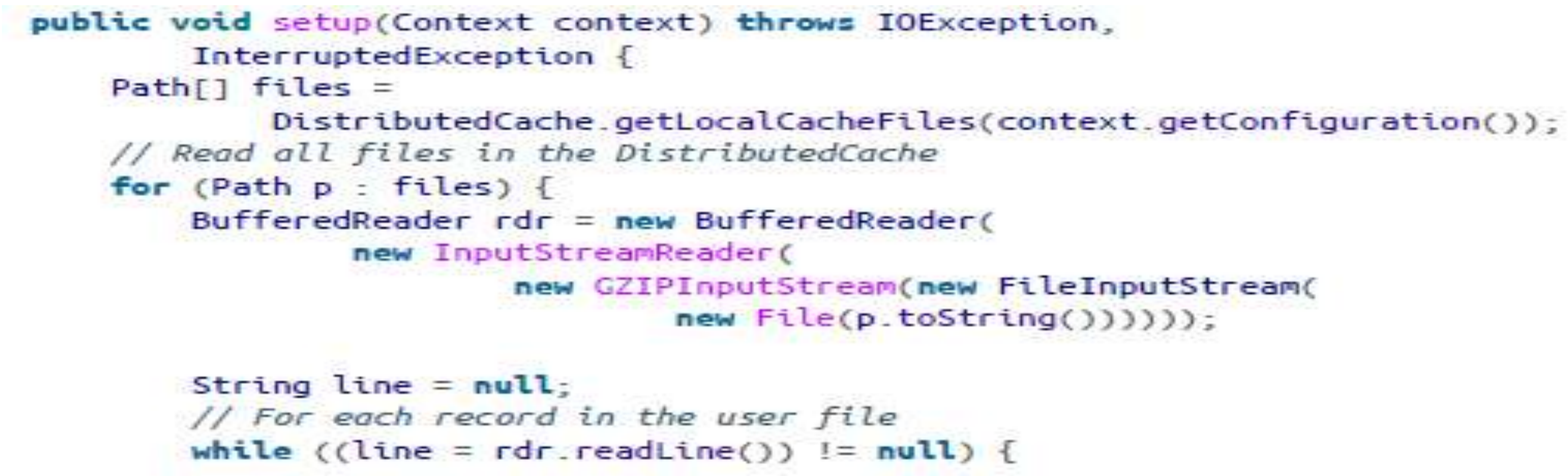


Replicated user-comment example

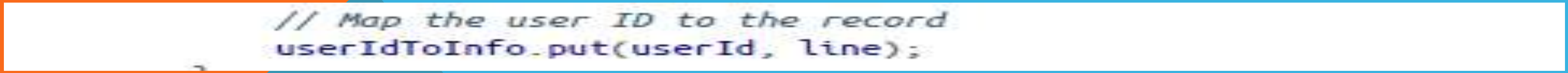
Small set of user information and a large set of comment
join operation between one large and many data sets performed on the map-sides
eliminates the need to shuffle any data to the reduce phase
useful only for an inner or a left outer join where the large data set is the “left” data set

Replicated Join : Mapper : setup

All the data sets except the very large one are read into memory during the setup phase of each map task which is limited by the JVM heap
user ID is pulled out of the record.
user ID and record are added to a HashMap for retrieval
in the map method



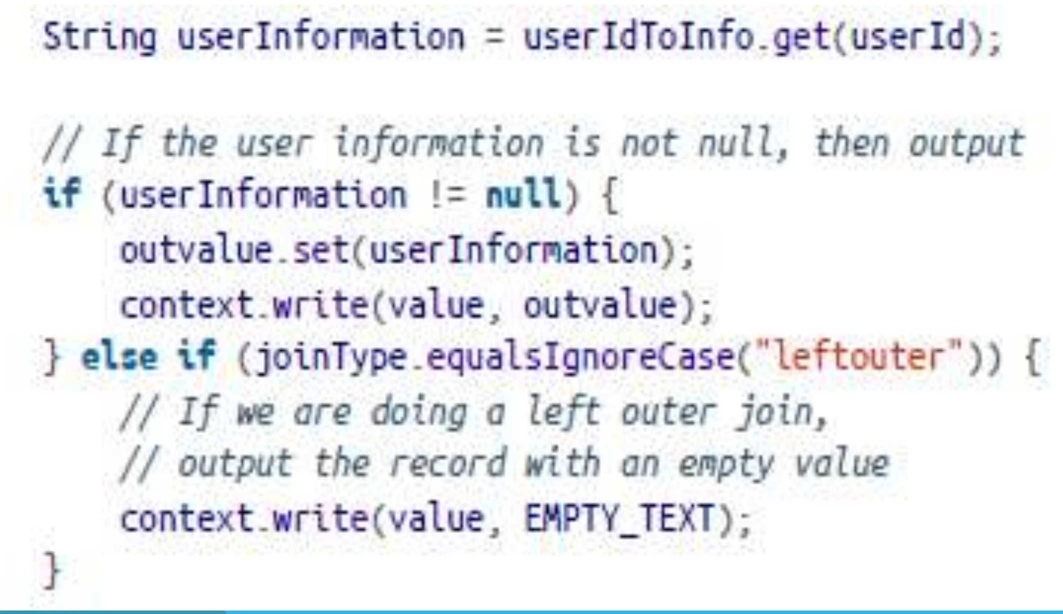
```
public void setup(Context context) throws IOException,
        InterruptedException {
    Path[] files =
            DistributedCache.getLocalCacheFiles(context.getConfiguration());
    // Read all files in the DistributedCache
    for (Path p : files) {
        BufferedReader rdr = new BufferedReader(
                new InputStreamReader(
                        new GZIPInputStream(new FileInputStream(
                                new File(p.toString())))));

        String line = null;
        // For each record in the user file
        while ((line = rdr.readLine()) != null) {
```



```
            // Map the user ID to the record
            userIdToInfo.put(userId, line);
```

Replicated Join : Mapper : map

consecutive calls to the map method are performed. For each input record,the user ID is pulled from the comment.
This user ID is then used to retrieve a value from the HashMap
If a value is found, the input value is output along with the retrieved value.
If a value is not found, but a left outer join is being executed



```
String userInformation = userIdToInfo.get(userId);

// If the user information is not null, then output
if (userInformation != null) {
    outvalue.set(userInformation);
    context.write(value, outvalue);
} else if (joinType.equalsIgnoreCase("leftouter")) {
    // If we are doing a left outer join,
    // output the record with an empty value
    context.write(value, EMPTY_TEXT);
}
```

Composite Join

Join operation that can be performed on the map-side with many very large formatted inputs

Eliminates the need to shuffle and sort all the data to the reduce phase

Data sets must first be sorted by foreign key, partitioned by foreign key

Hadoop has built in support for a composite join using the CompositeInputFormat.

This join utility is restricted to only inner and full outer joins

The inputs for each mapper must be partitioned and sorted in a specific way, and each input dataset must be divided into the same number of partitions.

All the records for a particular foreign key must be in the same partition

Composite Join

Driver code handles most of the work in the job configuration stage

It sets up the type of input format used to parse the data sets, as well as the join type to execute

The framework then handles executing the actual join when the data is read

Mapper is very trivial. The two values are retrieved from the input tuple and simply output to the file system

Output is a number of part files equivalent to the number of map tasks

Performance Analysis

Composite join can be executed relatively quickly over large data sets

Data preparation needs to taken into account in the performance of this analytic

Composite user comment join

user and comment data sets have been preprocessed by MapReduce and output using the TextOutputFormat

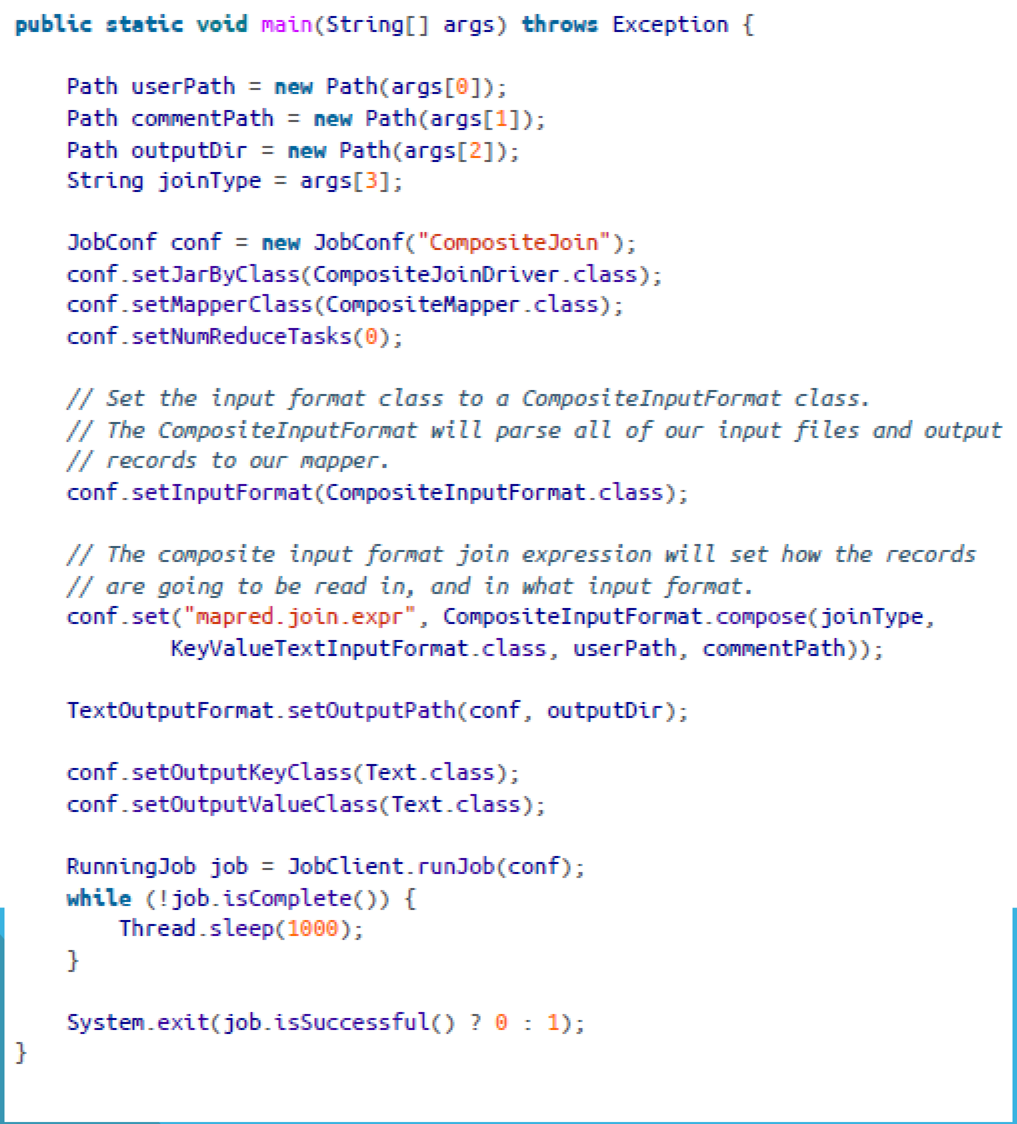
key of each data set is the user ID, and the value is the user comment

Each data set was sorted by the foreign key,
Each data set was then gzipped to prevent it from being split

two large formatted data sets of user information and comments

CompositeInputFormat utilizes the older mapred API

Composite Join



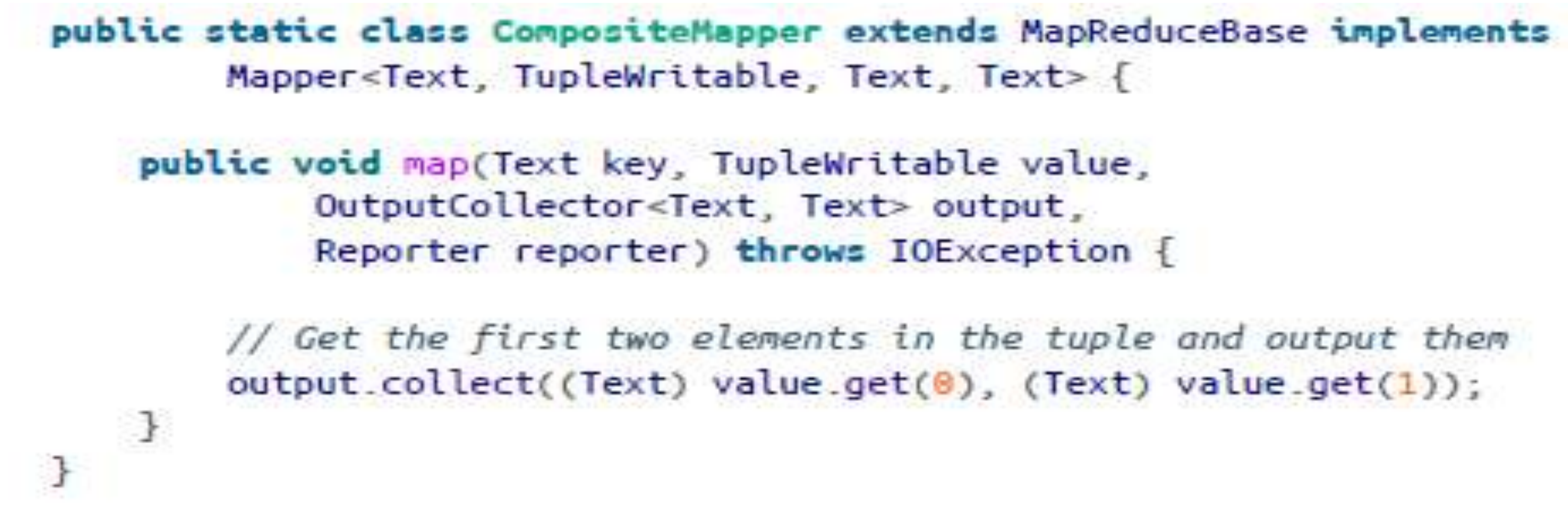
```java
public static void main(String[] args) throws Exception {

    Path userPath = new Path(args[0]);
    Path commentPath = new Path(args[1]);
    Path outputDir = new Path(args[2]);
    String joinType = args[3];

    JobConf conf = new JobConf("CompositeJoin");
    conf.setJarByClass(CompositeJoinDriver.class);
    conf.setMapperClass(CompositeMapper.class);
    conf.setNumReduceTasks(0);

    // Set the input format class to a CompositeInputFormat class.
    // The CompositeInputFormat will parse all of our input files and output
    // records to our mapper.
    conf.setInputFormat(CompositeInputFormat.class);

    // The composite input format join expression will set how the records
    // are going to be read in, and in what input format.
    conf.set("mapred.join.expr", CompositeInputFormat.compose(joinType,
            KeyValueTextInputFormat.class, userPath, commentPath));

    TextOutputFormat.setOutputPath(conf, outputDir);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(Text.class);

    RunningJob job = JobClient.runJob(conf);
    while (!job.isComplete()) {
        Thread.sleep(1000);
    }

    System.exit(job.isSuccessful() ? 0 : 1);
}
```

Composite Join : Mapper

Input to the mapper is the foreign key and a TupleWritable

Tuple contains a number of Text objects equivalent to the number of data sets

First input path is the zeroth index



```
public static class CompositeMapper extends MapReduceBase implements
        Mapper<Text, TupleWritable, Text, Text> {

    public void map(Text key, TupleWritable value,
            OutputCollector<Text, Text> output,
            Reporter reporter) throws IOException {

        // Get the first two elements in the tuple and output them
        output.collect((Text) value.get(0), (Text) value.get(1));
    }
}
```

Thank YOU !!!