MapReduce Join Patterns

- Having all your data in one giant data set is a rarity.
- For example, presume you have user information stored in a SQL database because it is updated frequently.
  - Meanwhile, web logs arrive in a constant stream and are dumped directly into HDFS. Also, daily analytics that make sense of these logs are stored someone where in HDFS and financial records are stored in an encrypted repository.
  - The list goes on.
- Data is all over the place, and while it's very valuable on its own, we can discover interesting relationships when we start analyzing these sets together.
  - This is where join patterns come into play.
  - Joins can be used to enrich data with a smaller reference set or they can be used to filter out or select records that are in some type of special list.
  - The use cases go on and on as well.

Joins in SQL

- In SQL, joins are accomplished using simple commands, and the database engine handles all of the grunt work.
- Sadly for us, joins in MapReduce are not nearly this simple.
- MapReduce operates on a single key/value pair at a time, typically from the same input.
- We are now working with at least two data sets that are probably of different structures, so we need to know what data set a record came from in order to process it correctly.
- Typically, no filtering is done prior to the join operation, so some join operations will require every byte of input to be sent to the reduce phase, which is very taxing on your network.
  - For example, joining 1 terabyte of data onto another 1 terabyte data set could require at least 2 terabytes of network bandwith—that's before any actual join logic can be done.

Cont'd

- On top of all of the complexity so far, one has to determine the best way out of a number of different ways to accomplish the same task.
- Because the framework is broken down into simple map and reduce tasks, there is a lot of hands-on work to do and a lot of things to keep in mind.
- After you learn the possibilities, the question to ask is when to use what pattern.
- As with any MapReduce operation, network bandwith is a very important resource and joins have a tendency to use a lot of it.
- Anything we can do to make the network transfer more efficient is worthwhile, and network optimizations are what differentiates these patterns.

Joins

- Each of the upcoming patterns can be used to perform an inner join or at least one type of outer join.

- As far as what pattern to choose, it depends largely on
  - how large the data sets are
  - how your data is formatted
  - what type of join you want.

- Choosing the right type of join for your situation can be challenging.

A Refresher on Joins

- Joins are possibly one of the most complex operations one can execute in MR.
- By design, MapReduce is very good at processing large data sets by looking at every record or group in isolation, so joining two very large data sets together does not fit into the paradigm gracefully.
- Before we dive into the patterns themselves, let's go over what we mean when we say join and the different types of joins that exist.
- 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, and is used as a means to cross-reference between tables.
- Examples are the simplest way to go about explaining joins, so let's dive right in.
  - To simplify explanations of the join types, two data sets will be used, A and B, with the foreign key defined as f. As the different types of joins are described, keep the 2 tables A and B in mind, as they will be used in the upcoming descriptions.

Table A

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

Table B

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

When people don't specify the type of join when they say "join", usually what they are talking about is an *inner join*. With this type of join, records from both *A* and *B* that contain identical values for a given foreign key *f* are brought together, such that all the columns of both *A* and *B* now make a new table. Records that contain values of *f* that are contained in *A* but not in *B*, and vice versa, are not represented in the result table of the join operation.

Table 5-3 shows the result of an inner join operation between *A* and *B* with User ID as *f*.

Table 5-3. Inner Join of A + B on User ID

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.

Records with a User ID of 3 or 5 are present in both tables, so they will be in the final table. Users 4 and 9 in table *A* and User 8 in table *B* are not represented in the other table, so the records will be omitted. However, these records will be present in a type of outer join, which brings us to our next type of join!

OUTER JOIN

An outer join is similar to an inner join, but records with a foreign key not present in both tables will be in the final table. There are three types of outer joins and each type will directly affect which unmatched records will be in the final table.

In a *left outer join*, the unmatched records in the "left" table will be in the final table, with null values in the columns of the right table that did not match on the foreign key. Unmatched records present in the right table will be discarded. A *right outer join* is the same as a left outer, but the difference is the right table records are kept and the left table values are null where appropriate. A *full outer join* will contain all unmatched records from both tables, sort of like a combination of both a left and right outer join.

Table 5-4 shows the result of a left outer join operation between *A* and *B* on User ID.

Table 5-4. Left Outer Join of A + B on User ID

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

Records with a user ID of 3 or 5 are present in both tables, so they will be in the final table. Users 4 and 9 in table *A* does not have a corresponding value in table *B*, but since this is a left outer join and *A* is on the left, these users will be kept but contain null values in the columns present only in table *B*. User 8 in *B* does not have a match in *A*, so it is omitted.

Table 5-5 shows the result of a right outer join operation between *A* and *B* on User ID.

Table 5-5. Right Outer Join of A + B on User ID

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!

Again, records with a user ID of 3 or 5 are present in both tables, so they will be in the final table. User 8 in *B* does not have a match in *A*, but is kept because *B* is the right table. Users 4 and 9 are omitted as they doesn't have a match in table *B*.

Table 5-6 shows the result of a full outer join operation between *A* and *B* on User ID.

Table 5-6. Full Outer Join of A + B on User ID

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

Once again, records with a user ID of 3 or 5 are present in both tables, so they will be in the final table. Users 4, 8, and 9 are present in the resulting table even though they do not contain matches in their respective opposite table.

ANTIJOIN

An *antijoin* is a full outer join minus the inner join. That is, the resulting table contains only records that did not contain a match on *f*.

Table 5-7 shows the result of an antijoin operation between *A* and *B* on User ID.

Table 5-7. Antijoin of A + B on User ID

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

Users 4, 8, and 9 do not contain a value of *f* in both tables, so they are in the resulting table. Records from user 3 and 5 are not present, as they are in both tables.

CARTESIAN PRODUCT

A *Cartesian product* or *cross product* takes each record from a table and matches it up with every record from another table. If table X contains n records and table Y contains m records, the cross product of X and Y, denoted $X \times Y$, contains $n \times m$ records. Unlike the other join operations, a Cartesian product does not contain a foreign key. As we will see in the upcoming pattern, this operation is extremely expensive to perform no matter where you implement it, and MapReduce is no exception.

Table 5-8 shows the result of a Cartesian product between *A* and *B*.

Table 5-8. Cartesian Product, A × B

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

Reduce Side Join Pattern

- Pattern Description:
- The reduce side join pattern can take the longest time to execute compared to the other join patterns, but it is simple to implement and supports all the different join operations discussed in the previous section.

- Intent:
- Join large multiple data sets together by some foreign key.

Motivation

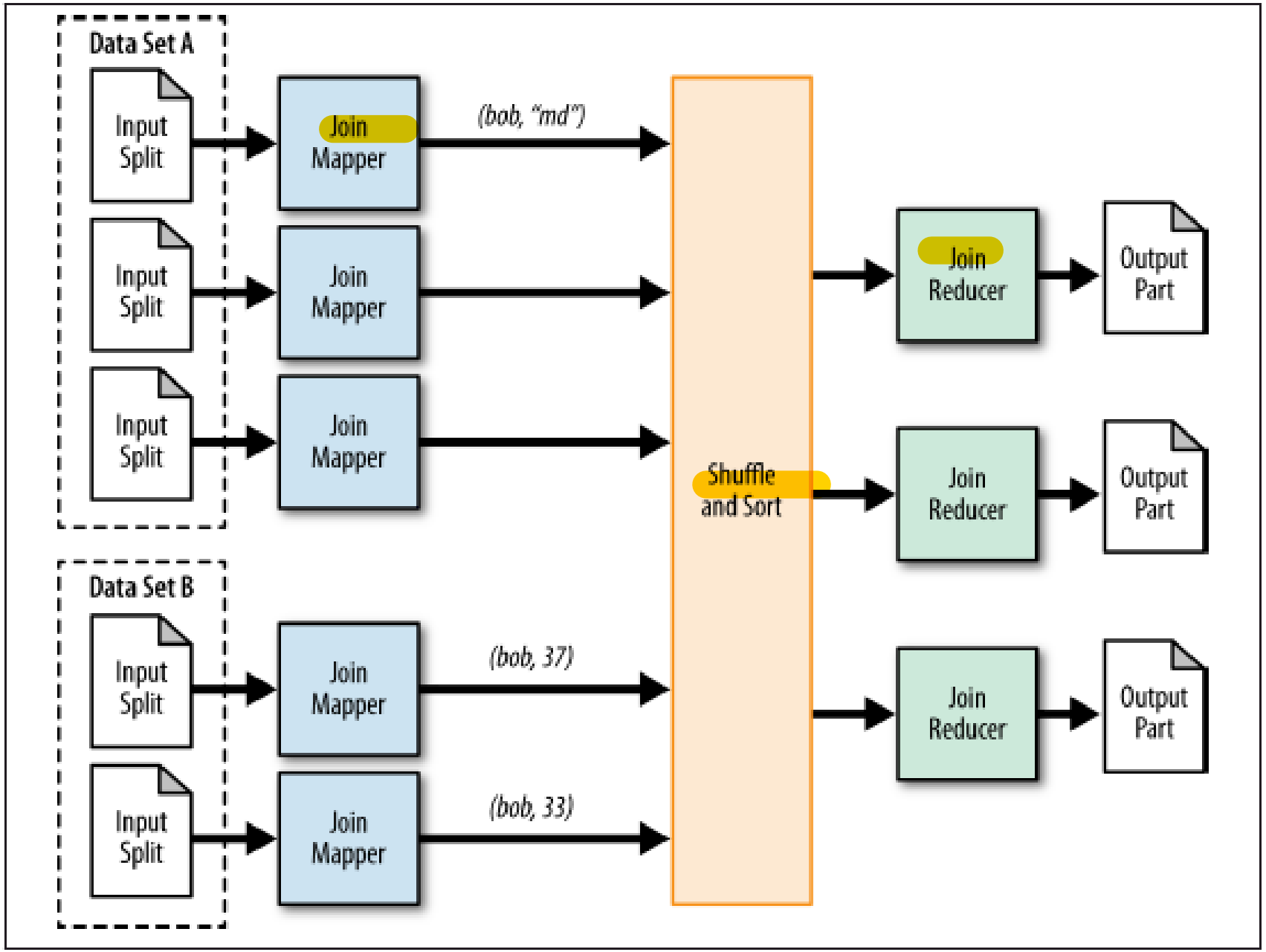
- A reduce side join is arguably one of the easiest implementations of a join in MR, and therefore is a very attractive choice.
- It can be used to execute any of the types of joins described above with relative ease and there is no limitation on the size of your data sets.
- Also, it can join as many data sets together at once as you need.
- All that said, a reduce side join will likely require a large amount of network bandwidth because the bulk of the data is sent to the reduce phase.
  - This can take some time, but if you have resources available and aren't concerned about execution time, by all means use it!
- Unfortunately, if all of the data sets are large, this type of join may be your only choice.

Applicability

- A reduce side join should be used when:
  - Multiple large data sets are being joined by a foreign key.
    - If all but one of the data sets can be fit into memory, try using the replicated join.
  - You want the flexibility of being able to execute any join operation.

Structure

- The mapper prepares the join operation by taking each input record from each of the data sets and extracting the foreign key from the record.
  - The FK is written as the output key, and the entire input record as the output value.
  - This output value is flagged by some unique identifier for the data set, such as A or B if two data sets are used. See Figure in the next slide.
- A hash partitioner can be used, or a customized partitioner can be created to distribute the intermediate key/value pairs more evenly across the reducers.
- The reducer performs the desired join operation by collecting the values of each input group into temporary lists.
  - For example, all records flagged with A are stored in the 'A' list and all records flagged with B are stored in the 'B' list.
  - These lists are then iterated over and the records from both sets are joined together.
  - For an inner join, a joined record is output if all the lists are not empty.
  - For an outer join (left, right, or full), empty lists are still joined with non empty lists.
  - The antijoin is done by examining that exactly one list is empty.
  - The records of the non-empty list are written with an empty writable.



The structure of the reduce side join pattern

Outcomes

- The output is a number of part files equivalent to the number of reduce tasks.
- Each of these part files together contains the portion of the joined records.
- The columns of each record depend on how they were joined in the reducer.
- Some column values will be null if an outer join or antijoin was performed.

Resemblances

SQL

Joins are very common in SQL and easy to execute.

```
SELECT users.ID, users.Location, comments.upVotes
FROM users
[INNER|LEFT|RIGHT] JOIN comments
ON users.ID=comments.UserID
```

Pig

Pig has support for inner joins and left, right, and full outer joins.

```
-- Inner Join
A = JOIN comments BY userID, users BY userID;

-- Outer Join
A = JOIN comments BY userID [LEFT|RIGHT|FULL] OUTER, users BY userID;
```

Reduce Side Join Example

- User and Comment Join
- In this example, we'll be using the users and comments tables from the StackOverflow data set.
- Storing data in this matter makes sense, as storing repetitive user data with each comment is unnecessary.
- This would also make updating user information difficult.
- However, having disjoint data sets poses problems when it comes to associating a comment with the user who wrote it.
- Through the use of a reduce side join, these two data sets can be merged together using the user ID as the foreign key.
- In this example, we'll perform an inner, outer, and antijoin.
- The choice of which join to execute is set during job configuration.

Cont'd

- Hadoop supports the ability to use multiple input data types at once, allowing you to create a mapper class & input format for each input split from different data sources.
- This is extremely helpful, because you don't have to code logic for two different data inputs in the same map implementation.
- In the following example, two mapper classes are created:
  - one for the user data
  - one for the comments.
- Each mapper class outputs the user ID as the foreign key, and the entire record as the value along with a single character to flag which record came from what set.
  - The reducer then copies all values for each group in memory, keeping track of which record came from what data set.
  - The records are then joined together and output.

Problem:

- Given a set of user information and a list of user's comments, enrich each comment with the information about the user who created the comment.

Driver Code

- The job configuration is slightly different from the standard configuration due to the user of the multiple input utility.
- We also set the join type in the job configuration to args[2] so it can be used in the reducer.
- The relevant piece of the driver code to use the MultipleInput follows:

```
...
// Use MultipleInputs to set which input uses what mapper
// This will keep parsing of each data set separate from a logical standpoint
// The first two elements of the args array are the two inputs
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]);
...
```

User Mapper Code

- This mapper parses each input line of user data XML.
- It grabs the user ID associated with each record and outputs it along with the entire input value.
- It prepends the letter A in front of the entire value.
- This allows the reducer to know which values came from what data set.

```java
public static class UserJoinMapper extends Mapper<Object, Text, Text, Text> {

    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        // Parse the input string into a nice map
        Map<String, String> parsed =
                MRDPUtils.transformXmlToMap(value.toString());

        String userId = parsed.get("Id");

        // The foreign join key is the user ID
        outkey.set(userId);

        // Flag this record for the reducer and then output
        outvalue.set("A" + value.toString());
        context.write(outkey, outvalue);
    }
}
```

Remarks

- When you output the value from the map side, the entire record doesn't have to be sent.
- This is an opportunity to optimize the join by keeping only the fields of data you want to join together.
  - It requires more processing on the map side, but is worth it in the long run.
- Also, since the foreign key is in the map output key, you don't need to keep that in the value, either.

Comment Mapper Code

- This mapper parses each input line of comment XML.
- Very similar to the UserJoinMapper, it too grabs the user ID associated with each record and outputs it along with the entire input value.
- The only different here is that the XML attribute UserId represents the user that posted to comment, where as Id in the user data set is the user ID.
- Here, this mapper prepends the letter B in front of the entire value.

```
public static class CommentJoinMapper extends
        Mapperlt;Object, Text, Text, Text> {

    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = transformXmlToMap(value.toString());

        // The foreign join key is the user ID
        outkey.set( parsed.get("UserId"));

        // Flag this record for the reducer and then output
        outvalue.set("B" + value.toString());
        context.write(outkey, outvalue);
    }
}
```

Reducer code

- The reducer code iterates through all the values of each group and looks at what each record is tagged with and then puts the record in one of two lists.
- After all values are binned in either list, the actual join logic is executed using the two lists.
- The join logic differs slightly based on the type of join, but always involves iterating through both lists and writing to the Context object.
- The type of join is pulled from the job configuration in the setup method.
- Let's look at the main reduce method before looking at the join logic.

```
public static class UserJoinReducer extends Reducer<Text, Text, Text, Text> {

  private static final Text EMPTY_TEXT = Text("");
    private Text tmp = new Text();
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();
    private String joinType = null;

    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);
    }

    private void executeJoinLogic(Context context)
            throws IOException, InterruptedException {
        ...
    }
```

Reducer Cont'd

- The input data types to the reducer are two Text objects.
- The input key is the foreign join key, which in this example is the user's ID.
- The input values associated with the foreign key contain one record from the "users" data set tagged with 'B', as well as all the comments the user posted tagged with 'B'.
- Any type of data formatting you would want to perform should be done here prior to outputting.
- For simplicity, the raw XML value from the left data set (users) is output as the key and the raw XML value from the right data set (comments) is output as the value.

Inner Join: if both the lists are no empty, simply perform two nested for loops and join each of the values together.


```
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);
            }
        }
    }
} ...
```

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);
        }
    }
} ...
```

Right Outer Join: if the left list is empty, write records from B with an empty output key.


```
... 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);
        }
    }
} ...
```

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);
        }
    }
} ...
```

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);
        }
    }
} ...
```

Remarks

- Be considerate of follow on data parsing to ensure proper field delimiters.
- Outputting an empty text object is actually unwise.
- A record that contains the proper structure but with null fields should be generated instead of outputting an empty object.
- This will ensure proper parsing for follow-on analytics.

- Combiner Optimization:
  - Because the join logic is performed on the reduce side, a combiner will not provide much optimization in this example.

Example: Reduce Side Join with Bloom Filter

- Reputable user and comment join
- This example is very similar to the previous one, but with the added optimization of using a Bloom filter to filter out some of mapper output.

- This will help reduce the amount of data being sent to the reducers and in effect reduce the runtime of our analytic.
  - Say we are only interested in enriching comments with reputable users, i.e., greater than 1,500 reputation.
  - A standard reduce side join could be used, with the added condition to verify that a user's reputation is greater than 1,500 prior to writing to the context object.
  - This requires all the data to be parsed and forwarded to the reducer for joining.

- If we could stop outputting data from the mappers that we know are not going to be needed in the join, then we can drastically reduce network I/O.

Cont'd

- Using a Bloom filter is particularly useful with an inner join operation, and may not be useful at all with a full outer join operation or an antijoin.
  - The latter two operations require all records to be sent to the reducer, so adding a Bloom filter has no value.
- Filtering out users that do not meet the reputation requirement is simple enough for the UserJoinMapper class, because the user reputation is in the data.
- However, there are a lot more comments than users and the user reputation is not available in each comment record.
- Through the use of a Bloom filter, a small amount of memory can be used to perform the test we desire.
- A preprocess stage is needed to train a Bloom filter with all users that have at least 1,500 reputation.

Cont'd

- In the following example, both mappers are slightly different from the previous.
- The UserJoinMapper adds a test prior to writing key/value pairs to the context to ensure the user has at least 1,500 reputation.
- The CommentJoin Mapper deserializes a Bloom filter from the DistributedCache and then used it as a test case prior to writing any output.
- The reducer remains the same as in the previous reduce side join example.
- The driver code is slightly different in that we use the DistributedCache to store the Bloom filters.
  - This is omitted in the following code, as a Reference Material on how to use a Bloom filter with the DistributedCache can be found on the Course Materials section of BlackBoard.

User mapper code. The user ID is pulled from the XML record along with the reputation. If the reputation is greater than 1,500, the record is output along with the foreign key (user ID).

```
public static class UserJoinMapper extends Mapper<Object, Text, Text, Text> {

    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {

        Map<String, String> parsed = transformXmlToMap(value.toString());

        // If the reputation is greater than 1,500,
        // output the user ID with the value
        if (Integer.parseInt(parsed.get("Reputation")) > 1500) {
            outkey.set(parsed.get("Id"));
            outvalue.set("A" + value.toString());
            context.write(outkey, outvalue);
        }
    }
}
```

Comment mapper code. The Bloom filter is initially deserialized from the `DistributedCache` prior to any calls to the map method. After deserialization, the user ID is pulled from the XML record and used for the membership test of the Bloom filter. If the test passes, the record is output along with the foreign key (user ID).

```
public static class CommentJoinMapperWithBloom extends
        Mapper<Object, Text, Text, Text> {

    private BloomFilter bfilter = new BloomFilter();
    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void setup(Context context) {
        Path[] files =
                DistributedCache.getLocalCacheFiles(context.getConfiguration());
        DataInputStream strm = new DataInputStream(
            new FileInputStream(new File(files[0].toString())));
        bfilter.readFields(strm);
    }

    public void map(Object key, Text value, Context context) {
            throws IOException, InterruptedException {

        Map>String, String> parsed = transformXmlToMap(value.toString());

        String userId = parsed.get("UserId");

        if (bfilter.membershipTest(new Key(userId.getBytes()))) {
            outkey.set(userId);
            outvalue.set("B" + value.toString());
            context.write(outkey, outvalue);
        }
    }
}
```

Using Bloom Filters

- In this algorithm, we don't need to verify the user's reputation in the reducer prior to writing to the file system.
- While false positive records were output from the CommentJoinMapperWithBloom, they won't be joined up with users on the reduce side since there will be nothing to join them with.
- The 100% check was done by only outputting user IDs with a reputation greater than 1,500.
- The main gain we received out of this Bloom filter was vastly reducing the number of comments output to the mapper phase.
- Be conscious of Bloom filter false positives and how they will affect your reduce side join operation.

Replicated Join Pattern

- Pattern Description
- A replicated join is a special type of join operation between one large and many small data sets that can be performed on the map-side.

- Intent
- This pattern completely eliminates the need to shuffle any data to the reducer.

Motivation

- A replicated join is an extremely useful, but has a strict size limit on all but one of the data sets to be joined.
- 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.
- If you can live within this limitation, you get a drastic benefit because there is no reduce phase at all, and therefore no shuffling or sorting.
- The join is done entirely in the map phase, with the very large data set being the input for the MapReduce job.
- There is an additional restriction that 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.
- The other join types require a reduce phase to group the "right" data set with the entirety of the left data set.
- Although there may not be a match for the data stored in memory for a given map task, there could be match in another input split.
  - Because of this, we will restrict this pattern to inner and left outer joins.

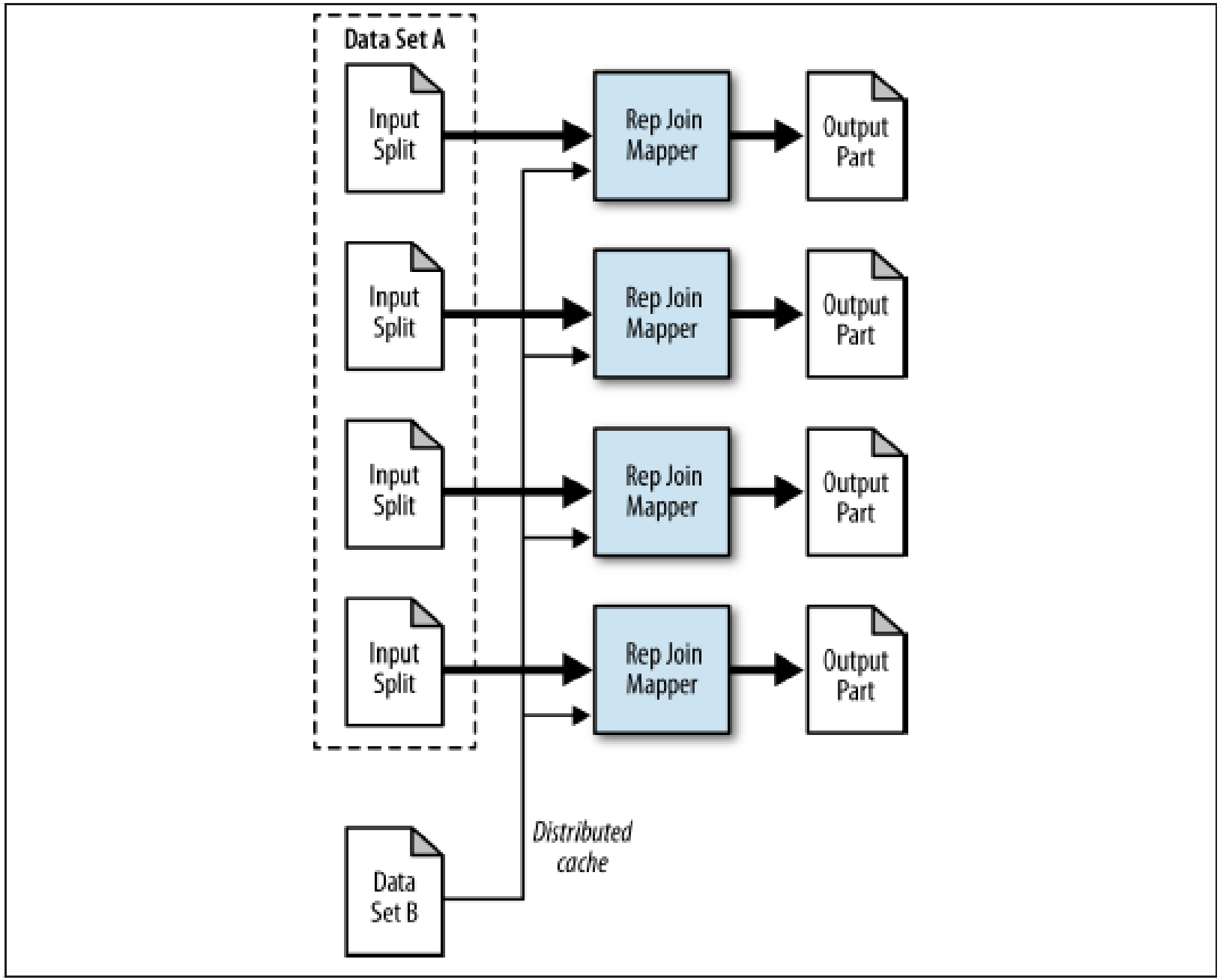
Applicability

- A replicated join should be used when:
  - 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.

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.
  - After this setup phase completes, the mapper processes each record and joins it with all the data stored in-memory.
  - If the foreign key is not found in the in-memory structures, the record is either omitted or output, based on the join type.

- No combiner, partitioner, or reducer is used for this pattern.
  - It is map-only.



The structure of the replicated join pattern

Outcomes

- The output is a number of part files equivalent to the number of map tasks.
- The part files contain the full set of joined records.
- If a left outer join is used, the input to the MapReduce analytic will be output in full, with possible null values.

Resemblances

Pig

Pig has native support for a replicated join through a simple modification to the standard join operation syntax. Only inner and left outer joins are supported for replicated joins, for the same reasons we couldn't do it above. The order of the data sets in the line of code matters because all but the first data sets listed are stored in-memory.

```
huge = LOAD 'huge_data' AS (h1,h2);
smallest = LOAD 'smallest_data' AS (ss1,ss2);
small = LOAD 'small_data' AS (s1,s2);
A = JOIN huge BY h1, small BY s1, smallest BY ss1 USING 'replicated';
```

Performance Analysis

- A replicated join can be the fastest type of join executed because there is no reducer required, but it comes at a cost.
  - There are limitations on the amount of data that can be stored safely inside the JVM, which is largely dependent on how much memory you are willing to give to each map and reduce task.
- Experiment around with your data sets to see how much you can fit into memory prior to fully implementing this pattern.
- Also, be aware that the memory footprint of your data set stored in-memory is not necessarily the number of bytes it takes to store it on disk.
- The data will be inflated due to Java object overhead.
- Thankfully, you can omit any data you know you will not need.

Replicated Join Examples

- Replicated user comment example
- This example is related to the previous replicated join with Bloom filter example.
- The DistributedCache is utilized to push a file around to all map tasks, but instead of a Bloom filter representation of the data, the data itself is read into memory.
- Instead of filtering out data that will never be joined on the reduce side, the data is joined in the map phase.

- Problem:
- Given a small set of user information and a large set of comments, enrich the comments with user information data.

Mapper Code

- During the setup phase of the mapper, the user data is read from the DistributedCache and stored in memory.
- Each record is parsed and the user ID is pulled out of the record.
- Then the user ID and record are added to a HashMap for retrieval in the map method.
  - This is where an out of memory error could occur, as the entire contents of the file is stored, with additional overhead of the data structure itself.
  - If it does, you will either have to increase the JVM size or use a plain reduce side join.

Cont'd

- After setup, consecutive calls to the map method are performed.
- For each input record, the user ID is pulled from the comment.
- Then, this user ID is used to retrieve a value from the HashMap built during the setup phase of the map.
  - 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, the input value is output with an empty Text object.
- That's all there is to it!
  - The input data is enriched with the data stored in memory.

```
public static class ReplicatedJoinMapper extends
        Mapper<Object, Text, Text, Text> {

  private static final Text EMPTY_TEXT = new Text("");
    private HashMap<String, String> userIdToInfo = new HashMap<String, String>();

    private Text outvalue = new Text();
    private String joinType = null;

    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) {

            // Get the user ID for this record
                Map<String, String> parsed = transformXmlToMap(line);
                String userId = parsed.get("Id");

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

        // Get the join type from the configuration
        joinType = context.getConfiguration().get("join.type");
    }

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = transformXmlToMap(value.toString());

        String userId = parsed.get("UserId");
        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 Pattern

- Pattern Description:
- A composite join is a specialized type of join operation that can be performed on the map-side with many very large formatted inputs.

- Intent:
- Using this pattern completely eliminates the need to shuffle and sort all the data to the reduce phase.

- However, it requires the data to be already organized or prepared in a very specific way.

Motivation

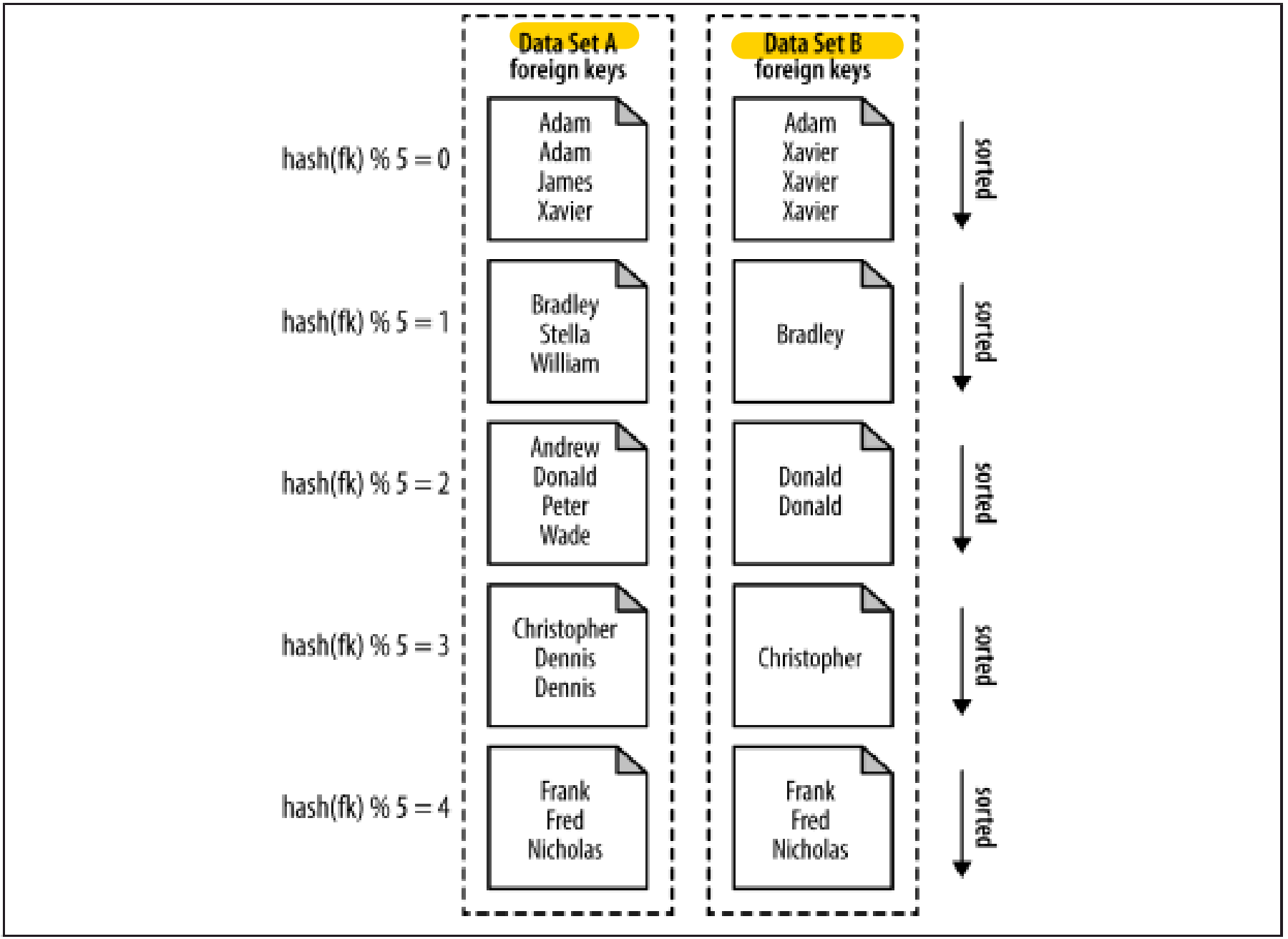
- Composite joins are particularly useful when joining very large data sets together.
- However, the data sets must first be sorted by foreign key, partitioned by foreign key, and read in a very particular manner in order to use this type of join.
- With that said, if your data can be read in such a way or you can prepare your data, a composite join has a huge leg-up over the other types.

Cont'd

- 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.
- In addition to that, all the records for a particular FK must be in the same partition.
- Usually, this occurs only if the output of several jobs has the same number of reducers and the same foreign key, and output files aren't splittable, i.e., not bigger than the HDFS block size or gzipped.
- In many cases, one of the other patterns presented in this lecture is more applicable.
- If you find yourself having to format the data prior to using a composite join, it is probably better to use a reduce side join unless this output is used by many analytics.

Applicability

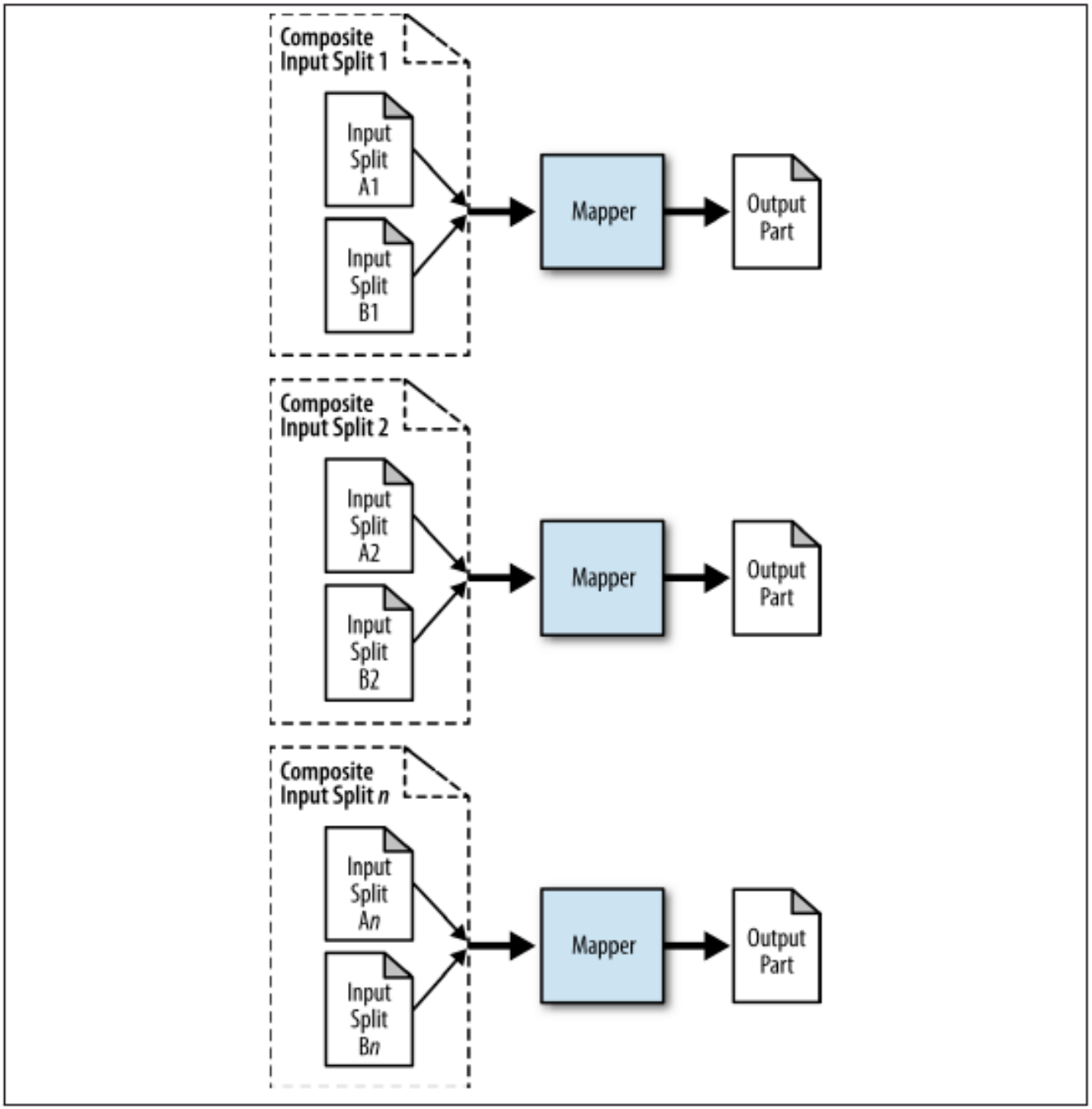
- A composite join should be used when:
  - An inner or full outer join is desired.
  - All the data sets are sufficiently large.
  - All data sets can be read with the foreign key as the input key to the mapper.
  - All data sets have the same number of partitions.
  - Each partition is sorted by foreign key, and all the foreign keys reside in the associated partition of each data set.
    - That is, partition X of data sets A and B contain the same foreign keys and these foreign keys are present only in partition X.
  - The data sets do not change often (if they have to be prepared).



Data sets that are sorted and partitioned on the same key

Structure

- The 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.
- The mapper is very trivial.
  - The two values are retrieved from the input tuple and simply output to the file system.
- No combiner, partitioner, or reducer is used for this pattern.
  - It is map-only.



The structure of the composite join pattern

Outcomes

- The output is a number of part files equivalent to the number of map tasks.
- The part files contain the full set of joined records.
- If configured for an outer join, there may be null values.

Performance Analysis

- A composite join can be executed relatively quickly over large data sets.
- However, the MapReduce framework can only set up the job so that one of the two data sets are data local.
- The respective files that are partitioned by the same key cannot be assumed to be on the same node.
- Any data preparation needs to taken into account in the performance of this analytic.
- The data preparation job is typically a MapReduce job, but if the data sets rarely change, then the sorted and partitioned data sets can be used over and over.
- Thus, the cost of producing these prepared data sets is averaged over all of the runs.

Composite Join Examples

- Composite User Comment Join
- To meet the preconditions of a composite join, both the user and comment data sets have been preprocessed by MapReduce and output using the TextOutputFormat.
- The key of each data set is the user ID, and the value is either the user XML or comment XML, based on the data set.
- Hadoop has a KeyValueTextOutputFormat to parse these formatted data sets as needed.
- The key will be the output key of our format job (user ID) and the value will be the output value (user or comment data).
- Each data set was sorted by the foreign key, the caveat being that they are sorted as Text objects rather than LongWritable objects.
  - That is, user “12345” comes before user “2”. This is because the CompositeInputFormat uses Text objects as the key for comparisons when doing the join.
  - Each data set was then gzipped to prevent it from being split.

- Problem:
- Given two large formatted data sets of user information and comments, enrich the comments with user information data.

Driver Code

- The driver parses the input arguments for the job:
  - the path to the user data
  - the path to the comment data
  - the analytic output directory
  - the type of join (inner or outer).
- The CompositeInputFormat utilizes the older mapred API, but configuration is similar to the mapreduce API.
- The most important piece of configuration is setting the input format and then configuring the join expression.

Cont'd

- The input format has a static helper function to create the join expression itself.
- It takes in the join type (inner or outer), the input format class used to parse all the data sets, and then as many Path or String objects as desired, which represent the data sets to join together.
- That's all there is to it!
- After setting the remaining required parameters, the job is run until completion and the program exits.

```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);
}
```

Mapper Code

- The input to the mapper is the foreign key and a TupleWritable.
- This tuple contains a number of Text objects equivalent to the number of data sets.
- As far as position is concerned, the ordering of the Text objects maps directly to how it was configured.
- The first input path is the 0^{th} index, the 2nd input path is the first index, and so on.
- The mapper simply grabs the objects from the tuple and outputs them.
- There are only two data sets to be joined in this example, so they are output as the key and value.
- If more were used, the strings would need be concatenated in some manner prior to being output.

```
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));
    }
}
```

Reducer and combiner

- This pattern has no reducer or combiner implementation because it is map only.

Cartesian Product Pattern

- Pattern Description:
- The Cartesian product pattern is an effective way to pair every record of multiple inputs with every other record.

- This functionality comes at a cost though, as a job using this pattern can take an extremely long time to complete.

- Intent:
- Pair up and compare every single record with every other record in a data set.

Motivation

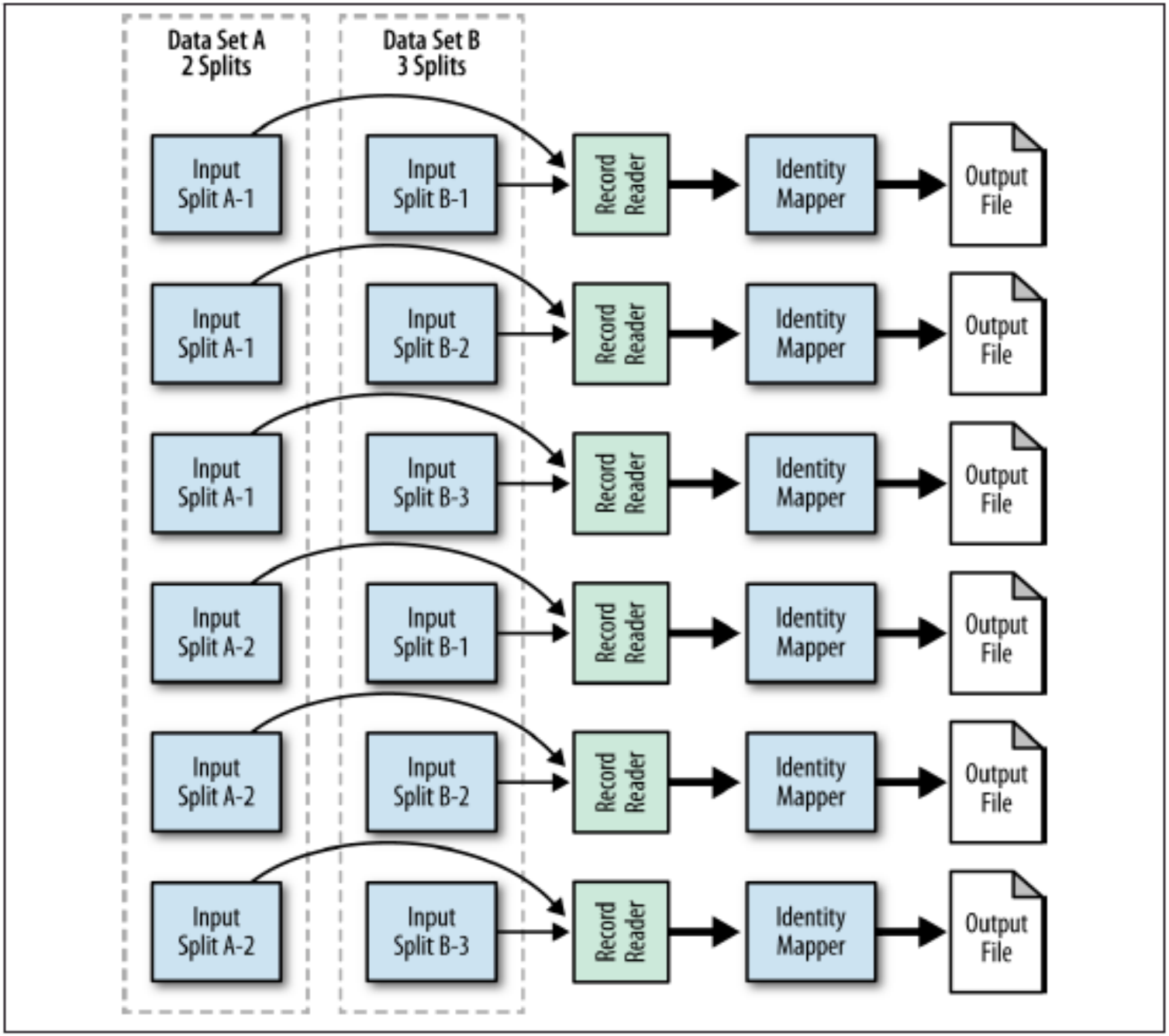
- A Cartesian product allows relationships between every pair of records possible between one or more data sets to be analyzed.
- Rather than pairing data sets together by a foreign key, a Cartesian product simply pairs every record of a data set with every record of all the other data sets.
- With that in mind, a Cartesian product does not fit into the MapReduce paradigm very well because the operation is not intuitively splittable, cannot be parallelized very well, and thus requires a lot of computation time and a lot of network traffic.
- Any preprocessing of that data that can be done to improve execution time and reduce the byte count should be done to improve runtimes.
- It is very rare that you would need to do a Cartesian product, but sometimes there is simply no FK to join on and the comparison is too complex to group by ahead of time.
- Most use cases for using a Cartesian product are some sort of similarity analysis on documents or media.

Applicability

- Use a Cartesian product when:
  - You want to analyze relationships between all pairs of individual records.
  - You've exhausted all other means to solve this problem.
  - You have no time constraints on execution time.

Structure

- The cross product of the input splits is determined during job setup and configuration.
  - After these are calculated, each record reader is responsible for generating the cross product from both of the splits it is given.
  - The record reader gives a pair of records to a mapper class, which simply writes them both out to the file system.
- No reducer, combiner, or partitioner is needed. This is a map-only job.



The structure of the Cartesian product pattern

Outcomes

- The final data set is made up of tuples equivalent to the number of input data sets.
- Every possible tuple combination from the input records is represented in the final output.

Resemblances

SQL

Although very rarely seen, the Cartesian product is the syntactically simplest of all joins in SQL. Just select from multiple tables without a `where` clause.

```
SELECT * FROM tablea, tableb;
```

Pig

Pig can perform a Cartesian product using the CROSS statement. It also comes along with a warning that it is an expensive operation and should be used sparingly.

```
A = LOAD 'data1' AS (a1, a2, a3);
DUMP A;
(1,2,3)
(4,5,6)

B = LOAD 'data2' AS (b1, b2);
DUMP B;
(1,2)
(3,4)
(5,6)

C = CROSS A, B;

DUMP C;
(1,2,3,1,2)
(1,2,3,3,4)
(1,2,3,5,6)
(4,5,6,1,2)
(4,5,6,3,4)
(4,5,6,5,6)
```

Performance Analysis

- The Cartesian product produces a massive explosion in data size, as even a self-join of a measly million records produces a trillion records.
- To be used very sparingly because it will use up many map slots for a very long time.
- This will dramatically increase the run time of other analytics, as any map slots taken by a Cartesian product are unusable by other jobs until completion.
- If the number of tasks is greater than or equal to the total number of map slots in the cluster, all other work won't get done for quite some time.
- Each input split is paired up with every other input split—effectively creating a data set of $O(n^2)$, n being the number of bytes.
- A single record is read from the left input split, and then the entire right input split is read and reset before the second record from the left input split is read.
- If a single input split contains a thousand records, this means the right input split needs to be read a thousand times before the task can finish.
  - This is a massive amount of processing time!
- If a single task fails for an odd reason, the whole thing needs to be restarted!
- You can see why a Cartesian product is a terrible, terrible thing to do in MapReduce.

Cartesian Product Examples

- Comment Comparison
- This example demonstrates how to perform a self-join using the StackOverflow comments.
- This self-join inspects a pair of comments and determines how similar they are to one another based on common words used between the two.
- If they are similar enough, the pair is output to the file system.
- Common words are removed from each comment along with other extra data in a preprocessing stage.
- This example is different than all other examples in that it pays special attention to how the data is read.
- Here, we create a custom input format to generate the Cartesian product of the input splits for the Job.
- If the data set to be processed contains 11 input splits, the job would contain 121 input splits, because 121 pairs are generated from the cross product.
- The record reader of each map task performs the actual Cartesian product and presents each pair to the mapper for processing.
- It accomplishes this by reading a single record from the “left” data set, then pairing it with all the records from the “right” data set.
- The next record is read from the left data set, the reader of the right data set is reset, and it is used to pair up again.
- This process continues until there are no more records in the left set.

Problem:

- Given a groomed data set of StackOverflow comments, find pairs of comments that are similar based on the number of like words between each pair.

Input Format Code

- The CartesianImportFormat piggybacks on a large portion of the CompositeInputFormat seen in the previous example of a composite join.
- It is implemented to support a Cartesian product for just two data sets for demonstration purposes in order to keep the code more simple.
- Single data set can be used as both left and right data sets, as we do for this example.
- During job setup, getInputSplits creates the cross product of the input splits of both data sets into a list of CompositeInputSplits.
- This is done by creating the underlying input format for each data set to get the splits, and then calculating the cross product.
- These input splits are then assigned to map task across the cluster for processing.

Remarks

- This is a homegrown Hadoop implementation of this problem for version to demonstrate the idea behind how a cross product can be executed using MR.
- Future versions of Hadoop MapReduce may have this functionality packaged with the distribution!

```java
public static class CartesianInputFormat extends FileInputFormat {

    public static final String LEFT_INPUT_FORMAT = "cart.left.inputformat";
    public static final String LEFT_INPUT_PATH = "cart.left.path";
    public static final String RIGHT_INPUT_FORMAT = "cart.right.inputformat";
    public static final String RIGHT_INPUT_PATH = "cart.right.path";

    public static void setLeftInputInfo(JobConf job,
            Class<? extends FileInputFormat> inputFormat, String inputPath) {
        job.set(LEFT_INPUT_FORMAT, inputFormat.getCanonicalName());
        job.set(LEFT_INPUT_PATH, inputPath);
    }

    public static void setRightInputInfo(JobConf job,
            Class<? extends FileInputFormat> inputFormat, String inputPath) {
        job.set(RIGHT_INPUT_FORMAT, inputFormat.getCanonicalName());
        job.set(RIGHT_INPUT_PATH, inputPath);
    }

    public InputSplit[] getSplits(JobConf conf, int numSplits)
            throws IOException {
        // Get the input splits from both the left and right data sets
        InputSplit[] leftSplits = getInputSplits(conf,
                conf.get(LEFT_INPUT_FORMAT), conf.get(LEFT_INPUT_PATH),
                    numSplits);
        InputSplit[] rightSplits = getInputSplits(conf,
                conf.get(RIGHT_INPUT_FORMAT), conf.get(RIGHT_INPUT_PATH),
                    numSplits);

        // Create our CompositeInputSplits, size equal to
        // left.length * right.length
        CompositeInputSplit[] returnSplits =
                new CompositeInputSplit[leftSplits.length *
                    rightSplits.length];

        int i = 0;
        // For each of the left input splits
        for (InputSplit left : leftSplits) {
                // For each of the right input splits
            for (InputSplit right : rightSplits) {
                // Create a new composite input split composing of the two
                returnSplits[i] = new CompositeInputSplit(2);
                returnSplits[i].add(left);
                returnSplits[i].add(right);
                ++i;
            }
        }

        // Return the composite splits
        LOG.info("Total splits to process: " + returnSplits.length);
        return returnSplits;
      }

    public RecordReader getRecordReader(InputSplit split, JobConf conf,
            Reporter reporter) throws IOException {
            // Create a new instance of the Cartesian record reader
            return new CartesianRecordReader((CompositeInputSplit) split,
                    conf, reporter);
    }

    private InputSplit[] getInputSplits(JobConf conf,
            String inputFormatClass, String inputPath, int numSplits)
            throws ClassNotFoundException, IOException {
        // Create a new instance of the input format
        FileInputFormat inputFormat = (FileInputFormat) ReflectionUtils
                .newInstance(Class.forName(inputFormatClass), conf);

        // Set the input path for the left data set
        inputFormat.setInputPaths(conf, inputPath);

        // Get the left input splits
        return inputFormat.getSplits(conf, numSplits);
    }
}
```

Driver code. The driver sets the necessary parameters for using the `CartesianInputFormat`. The same input path is used as both data sets for the input format, as we are performing a comparison between pairs of comments.

```
public static void main(String[] args) throws IOException,
        InterruptedException, ClassNotFoundException {

    // Configure the join type
    JobConf conf = new JobConf("Cartesian Product");
    conf.setJarByClass(CartesianProduct.class);

    conf.setMapperClass(CartesianMapper.class);
    conf.setNumReduceTasks(0);

    conf.setInputFormat(CartesianInputFormat.class);

    // Configure the input format
    CartesianInputFormat.setLeftInputInfo(conf, TextInputFormat.class, args[0]);
    CartesianInputFormat.setRightInputInfo(conf, TextInputFormat.class, args[0]);

    TextOutputFormat.setOutputPath(conf, new Path(args[1]));

    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);
}
```

Record Reader Code

- The record reader is where the magic happens of performing the cross product.
- During task setup, getRecordReader is called by the framework to return the CartesianRecordReader.
- The constructor of this class creates two separate record reader objects, one for each split.
- The first call to next reads the first record from the left data set for the mapper input key, and the first record from the right data set as the mapper input value.
- This key/value pair is then given to the mapper for processing by the framework.
- Subsequent calls to next then continue to read all the records from the right record reader, allowing the mapper to process them, until it says it has no more.
- In this case, a flag is set and the do-while will loop backwards, reading the second record from the left data set.
- The right record reader is reset, and the process continues.
- This process completes until the left record reader returns false, stating there are no more key/value pairs.
- At this point, the record reader has given the Cartesian product of both input splits to the map task.

Remarks

- Some of the more simple methods to adhere to the RecordReader interface are missing for brevity, such as close() and getPos().
- There are also some optimization opportunities that could be implemented, such as forcing the record reader to the next left record if you know it is not going to be useful.
- In this example, if the left record contains only one word in it and we are looking for pairs of comments that have a minimum of 3 common words, it doesn't make much sense to read the entire right input split because no output is going to be made.

```java
public static class CartesianRecordReader<K1, V1, K2, V2> implements
        RecordReader<Text, Text> {

    // Record readers to get key value pairs
    private RecordReader leftRR = null, rightRR = null;

    // Store configuration to re-create the right record reader
    private FileInputFormat rightFIF;
    private JobConf rightConf;
    private InputSplit rightIS;
    private Reporter rightReporter;

    // Helper variables
    private K1 lkey;
    private V1 lvalue;
    private K2 rkey;
    private V2 rvalue;
    private boolean goToNextLeft = true, alldone = false;

    public CartesianRecordReader(CompositeInputSplit split, JobConf conf,
            Reporter reporter) throws IOException {
        this.rightConf = conf;
        this.rightIS = split.get(1);
        this.rightReporter = reporter;

        // Create left record reader
        FileInputFormat leftFIF = (FileInputFormat) ReflectionUtils
                .newInstance(Class.forName(conf
                        .get(CartesianInputFormat.LEFT_INPUT_FORMAT)), conf);

        leftRR = leftFIF.getRecordReader(split.get(0), conf, reporter);

        // Create right record reader
        rightFIF = (FileInputFormat) ReflectionUtils.newInstance(Class
                .forName(conf
                        .get(CartesianInputFormat.RIGHT_INPUT_FORMAT)), conf);

        rightRR = rightFIF.getRecordReader(rightIS, rightConf, rightReporter);

        // Create key value pairs for parsing
        lkey = (K1) this.leftRR.createKey();
        lvalue = (V1) this.leftRR.createValue();

        rkey = (K2) this.rightRR.createKey();
        rvalue = (V2) this.rightRR.createValue();
    }

    public boolean next(Text key, Text value) throws IOException {
        do {
            // If we are to go to the next left key/value pair
            if (goToNextLeft) {
                // Read the next key value pair, false means no more pairs
                if (!leftRR.next(lkey, lvalue)) {
                    // If no more, then this task is nearly finished
                    alldone = true;
                    break;
                } else {
                    // If we aren't done, set the value to the key and set
                    // our flags
                    key.set(lvalue.toString());
                    goToNextLeft = alldone = false;

                    // Reset the right record reader
                    this.rightRR = this.rightFIF.getRecordReader(
                            this.rightIS, this.rightConf,
                            this.rightReporter);
                }
            }

            // Read the next key value pair from the right data set
            if (rightRR.next(rkey, rvalue)) {
                // If success, set the value
                value.set(rvalue.toString());
            } else {
                // Otherwise, this right data set is complete
                // and we should go to the next left pair
                goToNextLeft = true;
            }

            // This loop will continue if we finished reading key/value
            // pairs from the right data set
        } while (goToNextLeft);

        // Return true if a key/value pair was read, false otherwise
        return !alldone;
    }
}
```

Mapper code. The mapper is presented with a cross product pair. For each `Text` object, it reads the word tokens into a set. The sets are then iterated to determine how many common words there are between the two. If there are more then ten words, the pair is output to the file system.

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

    private Text outkey = new Text();

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

        // If the two comments are not equal
        if (!key.toString().equals(value.toString())) {
            String[] leftTokens = key.toString().split("\\s");
            String[] rightTokens = value.toString().split("\\s");

            HashSet<String> leftSet = new HashSet<String>(
                    Arrays.asList(leftTokens));
            HashSet<String> rightSet = new HashSet<String>(
                    Arrays.asList(rightTokens));

            int sameWordCount = 0;
            StringBuilder words = new StringBuilder();
            for (String s : leftSet) {
                if (rightSet.contains(s)) {
                    words.append(s + ",");
                    ++sameWordCount;
                }
            }

            // If there are at least three words, output
            if (sameWordCount > 2) {
                outkey.set(words + "\t" + key);
                output.collect(outkey, value);
            }
        }
    }
}
```