CS 378 – Big Data Programming

Lecture 19

Join Patterns

Review

- Assignment 8 – User Session
  - Replicated Join in mappers
  - MultipleOutputs from reducer

- Review the details of the assignment

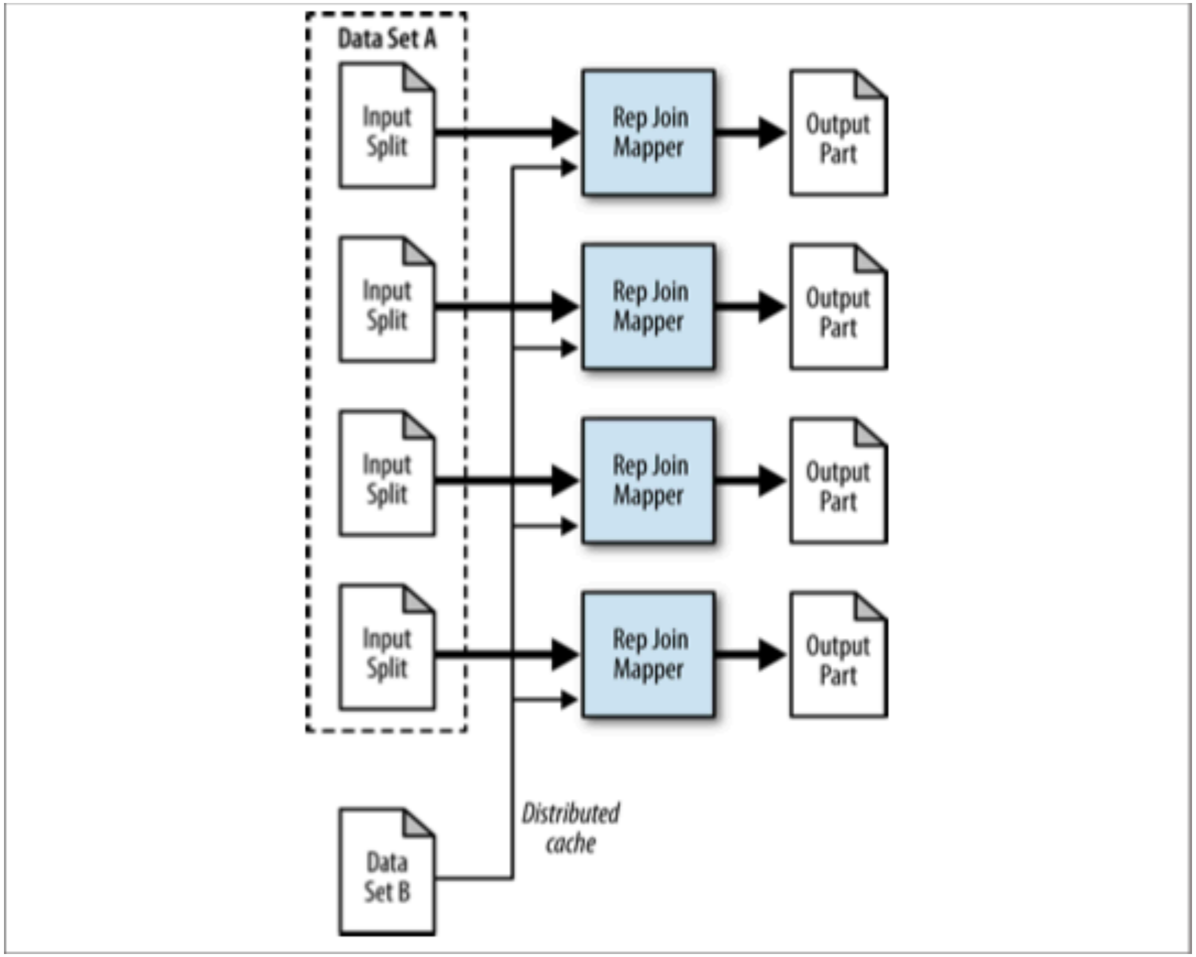
- Questions/issues:
  - DistributedCache setup (in run(), and in map class)

Replicated Join

- Suppose we want to join many sources, only one of which is large
  - User sessions (large)
  - Map from ZIP codes to DMA (demographic marketing area)
- This is called a *replicated* join
  - All the small files will be replicated to all machines
  - All small files must fit in memory
  - Files are replicated with `DistributedCache`

Replicated Join - Data Flow

Figure 5-2 from MapReduce Design Patterns



DistributedCache

- In the driver code (`run()` method)
  - Get the file name from the command line
  - Tell Hadoop about this file
  - File(s) conveyed in the configuration object

```
Path cacheFilePath = new Path(args[3]);
DistributedCache.addCacheFile(
   cacheFilePath.toUri(), conf);
```

DistributedCache

- In the mapper code (`setup()` method)
  - Get the file names from the configuration object
  - Load the data

```
Path[] paths = DistributedCache.getLocalCacheFiles(
      context.getConfiguration());
```

For each entry in `paths`, input the data:

```
Scanner scanner = new Scanner(
   new File(path[i].toString()));
```

MultipleOutputs Setup

- In the `run()` method, specify the named output
  - "Named output": label for specific output format
  - We can write different files using one "named output"

```
MultipleOutputs.addNamedOutput(job, “userType”,
   TextOutputFormat.class, Text.class, Text.class);
```

- Enable counters for the multiple outputs
  - By default they are off, as there may be many counters

```
MultipleOutputs.setCountersEnabled(job, true);
```

MultipleOutputs Setup

- In the reduce class, define an instance variable
  - Why an instance variable?

```
private MultipleOutputs multipleOutputs;
```

- In the `setup()` method of reducer
  - Create the MultipelOutputs object

```
public void setup(Context context) {
   multipleOutputs = new MultipleOutputs(context);
}
```

MultipleOutputs Setup

- In `reduce()` method:
  - Here’s where we write to different files (*category* argument)

```
multipleOutputs.write(“userType”,  key, value, category);
```

- In the `cleanup()` method of reducer

```
public void cleanup(Context context) {
   multipleOutputs.close();
}
```

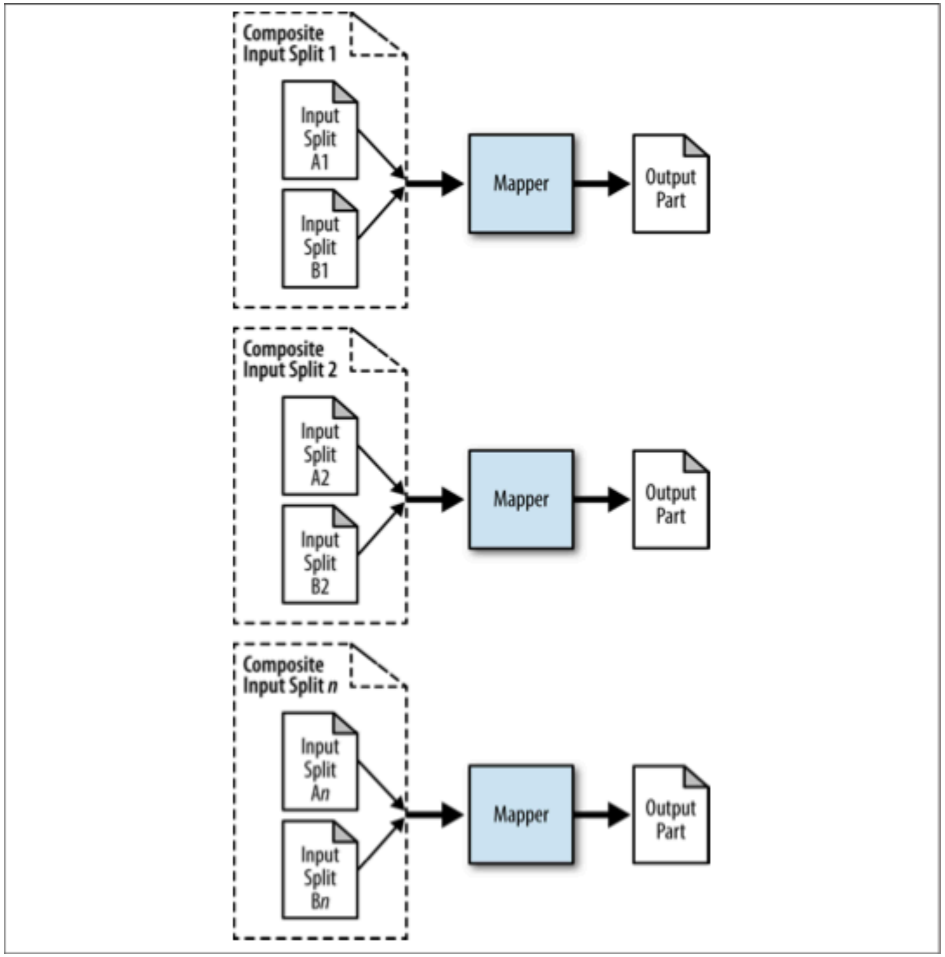
One More Join Pattern

- Suppose we wanted to compare all cars currently available (for sale) to all other cars
  - To identify "similar" cars
  - Usage: "I like this car, show me others like it"
- This join is called "Cartesian Product"
  - Compare N items to M items requires NxM comparisons
  - Not straightforward to do with map-reduce

Cartesian Product

- To accomplish this join, we'll need to pair every record with every other record
- We can start with the approach for composite join
- For composite join, each mapper read two files
  - They had the same key set
  - The data was sorted by key
  - We don't care about the keys, just the 'two file input '

Composite Join – Data Flow



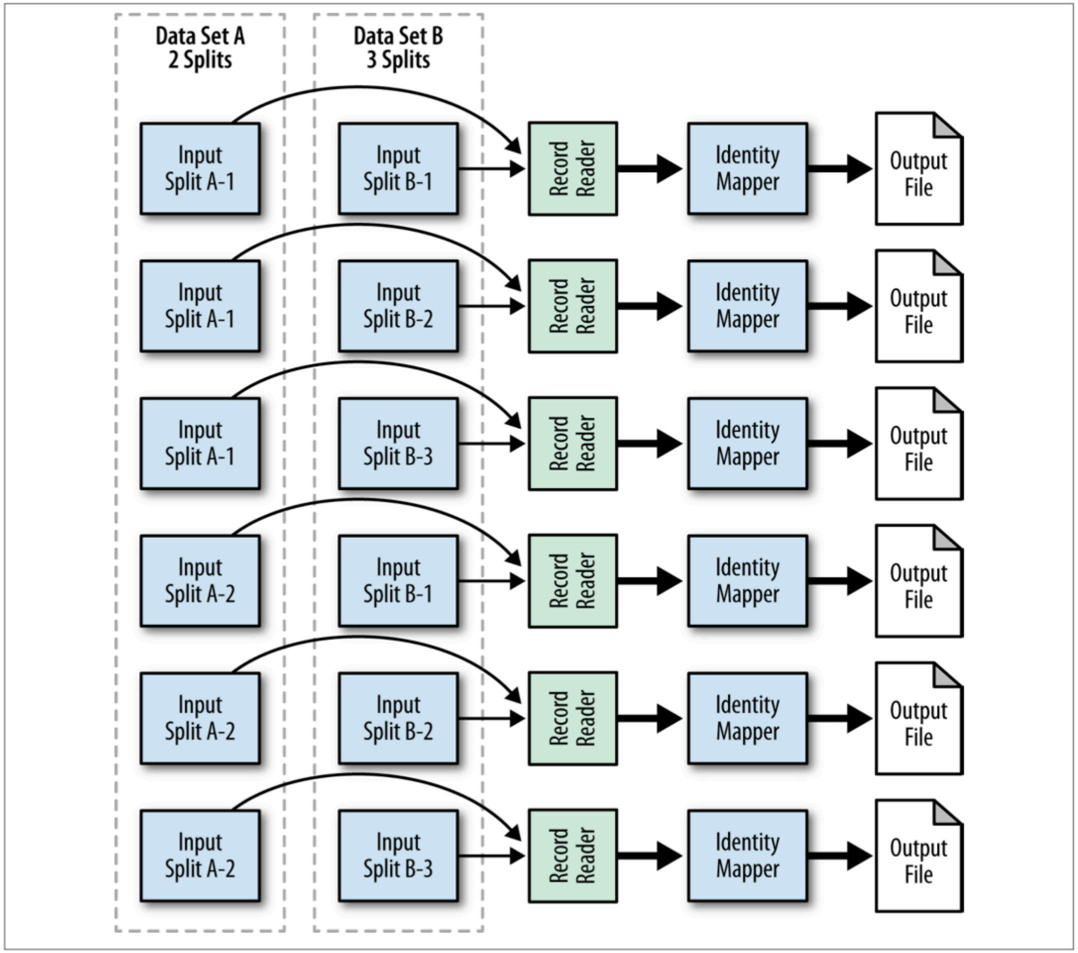
One Mapper, Two Inputs

- For composite join, the key order allowed us to:
  - Read each of the two files only once
  - Worked very much like merge sort

- For Cartesian product
  - For each record in data set 1
  - We'll read every record in data set 2
  - This pair of records is passed to the mapper

- We'd accomplish this with a custom input format
  - RecordReader resets data set 2 for each input of data set 1

Cartesian Product – Data Flow



Cartesian Product

- Pairs every record with every other record
  - No keys needed
  - N x M results, for datasets of size N, M

- Map-only job
- But still expensive to compute

Cartesian Product

- What do we want to output?
  - Inverted index?
    - To all other vehicles?
    - Only some subset?
  - Similarity distance/score?

- Are there some ways to filter this data
  - To limit the processing time
  - To limit the amount of data written out