last.fm A Case Study

- Internet radio and community-driven music discovery service.
- Users transmit information to Last.fm servers indicating which songs they are listening to.
- The received data is processed and stored so the user can access it in the form of charts and so Last.fm can make intelligent taste and compatibility decisions for generating recommendations and radio stations.
- The track listening data is obtained from one of two sources:
  - The listen is a scrobble when a user plays a track of his or her own and sends the information to Last.fm through a client application.
  - The listen is a radio listen when the user tunes into a Last.fm radio station and streams a song.
  - Last.fm applications allow users to love, skip or ban each track they listen to. This track listening data is also transmitted to the server.

Big Data at Last.fm

Within the first four years after launch:

- Over 40M unique visitors and 500M pageviews each month
- **Scrobble stats:**
  - Up to 800 scrobbles per second
  - More than 40 million scrobbles per day
  - Over 75 billion scrobbles so far
- **Radio stats:**
  - Over 10 million streaming hours per month
  - Over 400 thousand unique stations per day
- Each scrobble and radio listen generates at least one log line.
- In other words...**lots of data!!**

- Last.FM's *"Herd of Elephants"*
  - 100 Nodes
  - 8 cores per node (dual quad-core)
  - 24GB memory per node
  - 8TB (4 disks of 2TB each)
- Hive integration to run optimized SQL queries for analysis.

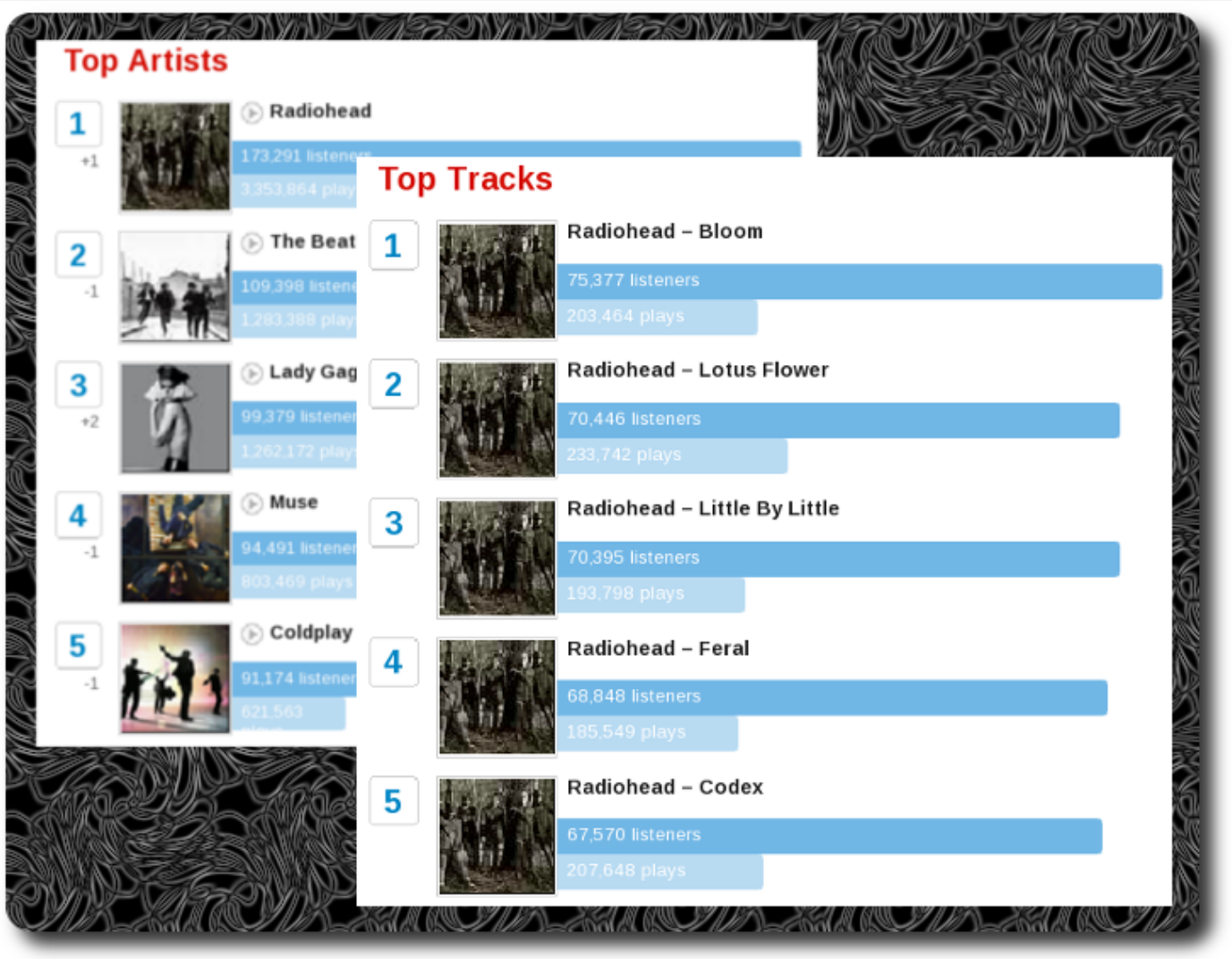
- Started using Hadoop as users grew from thousands to millions.
- Hundreds of daily, monthly, and weekly jobs including,
  - Site stats and metrics
  - Chart generation (track statistics)
  - Metadata corrections (e.g. misspellings of artists)
  - Indexing for search and combining/formatting data for recommendations
  - Data insights, evaluations, reporting
- This case study will focus on the chart generation (Track Statistics) job, which was the first Hadoop implementation at Last.fm.

Features of the case study

- Shows how to handle k-tuples, where $k > 2$ using the *Writable* and *WritableComparable* interfaces from the Hadoop library.
- Shows how to have multiple MapReduce phases using the *ToolRunner*, *Tool* and *Configuration* classes/interfaces.
- Shows how to merge data from two different streams using MapReduce.

Last.fm Chart Generation (Track Statistics)

The goal of the Track Statistics program is to take incoming listening data and summarize it into a format that can be used to display on the website or used as input to other Hadoop programs.



Track Statistics Jobs

- **Input:** Gigabytes of space-delimited text files of the form (userID trackID scrobble radioPlay skip), where the last three fields are 0 or 1.

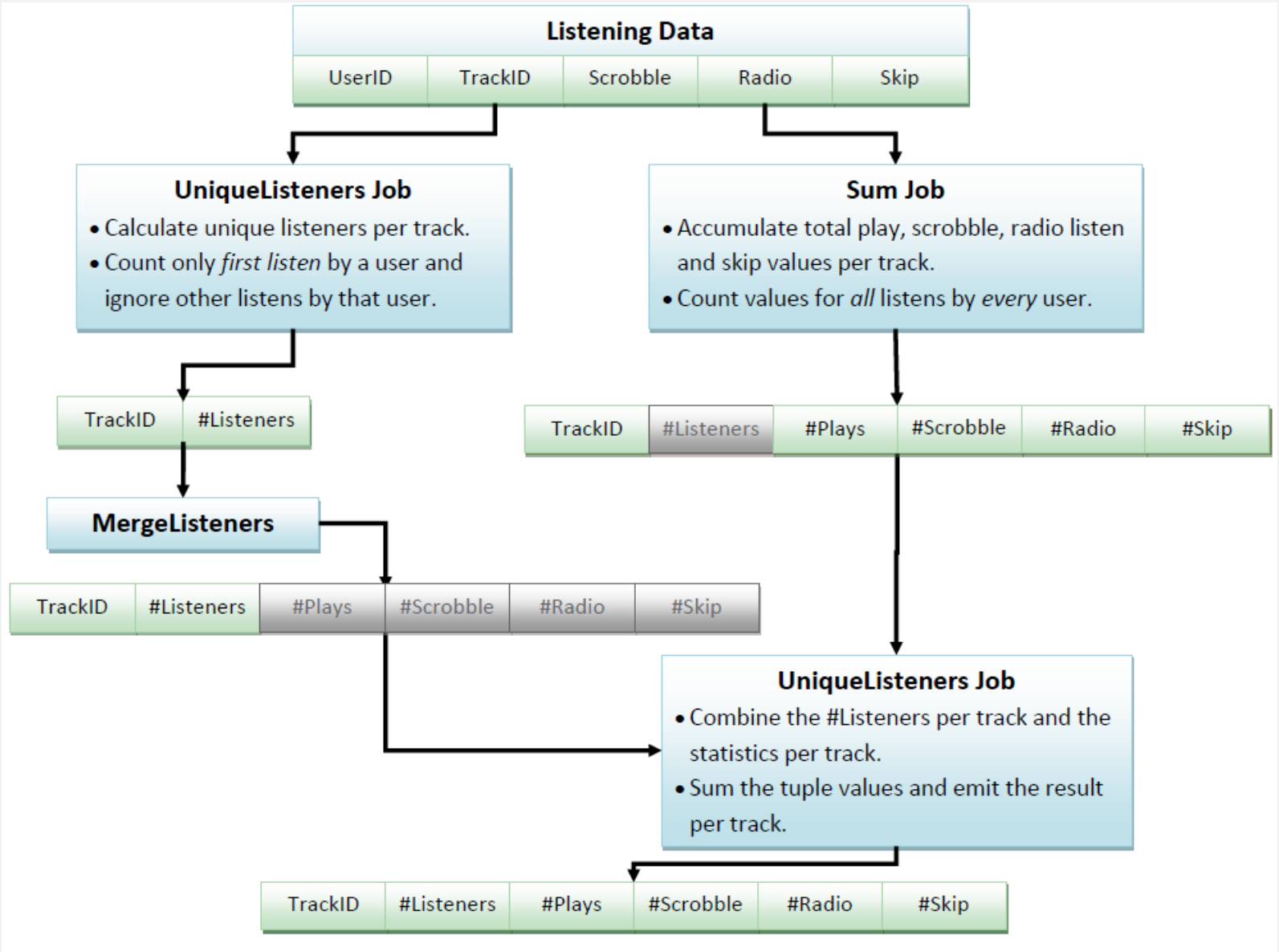
UserId	TrackId	Scrobble	Radio	Skip
111115	222	0	1	0
111113	225	1	0	0
111117	223	0	1	1
111115	225	1	0	0

- **Output:** Charts require the following statistics per track:
  - Number of unique listeners
  - Number of scrobbles
  - Number of radio listens
  - Total number of listens
  - Number of radio skips

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`
222	1	1	0	1	0
223	1	1	0	1	1
225	2	2	2	0	0

Track Statistics Program

Two jobs to calculate values from the data, and a third job to merge the results.



UniqueListeners Job

Calculates the number of *unique* listeners per track.

- `UniqueListenersMapper` **input**:
  - `key` is the line number of the current log entry.
  - `value` is the space-delimited log entry.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- `UniqueListenersMapper` **function**:
  - `if(scrobbles <= 0 && radioListens <=0) output nothing; else output(trackId, userId)`
  - `map(0, '111115 222 0 1 0')` $\rightarrow$ `<222, 111115>`
- `UniqueListenersMapper` **output**:

TrackId	UserId
IntWritable	IntWritable
222	111115
225	111113
223	111117
225	111115

UniqueListenersMapper Class

```
/**
 * Processes space-delimited raw listening data and emits the
 * user ID associated with each track ID.
 */
public static class UniqueListenersMapper extends
    Mapper<LongWritable, Text, IntWritable, IntWritable> {

  public void map(LongWritable offset, Text line, Context context)
      throws IOException, InterruptedException
  {
    String[] parts = (line.toString()).split(" ");

    int scrobbles = Integer.parseInt(parts[TrackStatistics.COL_SCROBBLE]);
    int radioListens = Integer.parseInt(parts[TrackStatistics.COL_RADIO]);

    /* Ignore track if marked with zero plays */
    if (scrobbles <= 0 && radioListens <= 0)
      return;

    /* Output user id against track id */
    IntWritable trackId = new IntWritable(
        Integer.parseInt(parts[TrackStatistics.COL_TRACKID]));
    IntWritable userId = new IntWritable(
        Integer.parseInt(parts[TrackStatistics.COL_USERID]));
    context.write(trackId, userId);
  }
}
```

UniqueListenersReducer

- `UniqueListenersReducer` **input**:
  - `key` is the TrackID output by `UniqueListenersMapper`.
  - `value` is the iterator over the list of all UserIDs who listened to the track.

TrackId	Iterable<UserIds>
IntWritable	Iterable<IntWritable>
222	111115
225	111113 111115
223	111117

- `UniqueListenersReducer` **function**:
  - Add `userIds` to a `HashSet` as you iterate the list. Since a `HashSet` doesn't store duplicates, the size of the set will be the number of unique listeners.
  - `reduce(225, '111115 111113')` → `<225, 2>`
- `UniqueListenersReducer` **output**:

TrackId	numListeners
IntWritable	IntWritable
222	1
223	1
225	2

UniqueListenersReducer Class

```
/**
 * Receives a list of user IDs per track ID and puts them into a
 * set to remove duplicates. The size of this set (the number of
 * unique listeners) is emitted for each track.
 */
public static class UniqueListenersReducer extends
  Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {

  public void reduce(IntWritable trackId, Iteratble<IntWritable> values,
                              Context context)
    throws IOException, InterruptedException
  {

    Set<Integer> userIds = new HashSet<Integer>();

    /* Add all users to set, duplicates automatically removed */
    while (values.hasNext()) {
      IntWritable userId = values.next();
      userIds.add(Integer.valueOf(userId.get()));
    }
    /* Output trackId -> number of unique listeners per track */
    context.write(trackId, new IntWritable(userIds.size()));
  }
}
```

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- SumTrackStatsMapper **input**: The same as UniqueListeners.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- SumTrackStatsMapper **function**:
  - Simply parse input and output the values as a new TrackStats object (see next slide).
  - `map(0, '111115 222 0 1 0') →`
    `<222, new TrackStats(0, 1, 0, 1, 0)>`
- SumTrackStatsMapper **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
225	0	1	1	0	0
223	0	1	0	1	1
225	0	1	1	0	0

TrackStats object

```
public class TrackStats implements WritableComparable<TrackStats> {

   private IntWritable listeners;
   private IntWritable plays;
   private IntWritable scrobbles;
   private IntWritable radioPlays;
   private IntWritable skips;

   public TrackStats(int numListeners, int numPlays, int numScrobbles,
                     int numRadio, int numSkips) {
        this.listeners = new IntWritable(numListeners);
        this.plays = new IntWritable(numPlays);
        this.scrobbles = new IntWritable(numScrobbles);
        this.radioPlays = new IntWritable(numRadio);
        this.skips = new IntWritable(numSkips);
   }

   public TrackStats() {
        this(0, 0, 0, 0, 0);
   }

   ...

}
```

SumTrackStatsMapper Class

```
public static class SumTrackStatsMapper extends
    Mapper<LongWritable, Text, IntWritable, TrackStats> {

public void map(LongWritable offset, Text line, Context context)
    throws IOException, InterruptedException
{
  String[] parts = (line.toString()).split(" ");
  int trackId = Integer.parseInt(parts[TrackStatistics.COL_TRACKID]);
  int scrobbles = Integer.parseInt(parts[TrackStatistics.COL_SCROBBLE]);
  int radio = Integer.parseInt(parts[TrackStatistics.COL_RADIO]);
  int skip = Integer.parseInt(parts[TrackStatistics.COL_SKIP]);

  TrackStats trackstat = new TrackStats(0, scrobbles + radio,
          scrobbles, radio, skip);
  context.write(new IntWritable(trackId), trackstat);
  }
}
```

SumTrackStatsReducer

- `SumTrackStatsReducer` **input**:
  - `key` is the TrackID output by the mapper.
  - `value` is the iterator over the list of TrackStats associated with the track.

TrackId	Iterable<TrackStats>
IntWritable	Iterable<TrackStats>
222	(0,1,0,1,0)
225	(0,1,1,0,0) (0,1,1,0,0)
223	(0,1,0,1,1)

- `SumTrackStatsReducer` **function**:
  - Create new `TrackStats` object to hold totals for current track.
  - Iterate through values and add the stats of each value to the stats of the object we created.
  - `reduce(225,'(0,1,1,0,0) (0,1,1,0,0)')` → <225, (0,2,2,0,0)>
- `SumTrackStatsReducer` **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
223	0	1	0	1	1
225	0	2	2	0	0

SumTrackStatsReducer

```
public static class SumTrackStatsReducer extends
    Reducer<IntWritable, TrackStats, IntWritable, TrackStats> {

  public void reduce(IntWritable trackId, Iterable<TrackStats> values,
                                  Context context)
    throws IOException, InterruptedException
  {
    /* Hold totals for this track */
    TrackStats sum = new TrackStats();
    while (values.hasNext()) {
      TrackStats trackStats = (TrackStats) values.next();
      sum.setListeners(sum.getListeners() + trackStats.getListeners());
      sum.setPlays(sum.getPlays() + trackStats.getPlays());
      sum.setSkips(sum.getSkips() + trackStats.getSkips());
      sum.setScrobbles(sum.getScrobbles() + trackStats.getScrobbles());
      sum.setRadioPlays(sum.getRadioPlays() + trackStats.getRadioPlays());
    }
    context.write(trackId, sum);
  }
}
```

Merging the Results: `MergeResults` Job

Merges the output from the `UniqueListeners` and `SumTrackStats` jobs.

- Specify a mapper for each input type:

```
MultipleInputs.addInputPath(conf, sumInputDir,
    SequenceFileInputFormat.class, IdentityMapper.class);

MultipleInputs.addInputPath(conf, listenersInputDir,
    SequenceFileInputFormat.class, MergeListenersMapper.class);
```

- `IdentityMapper` simply emits the `trackId` and `TrackStats` object output by the `SumTrackStats` job.
- `IdentityMapper` **input** and **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
223	0	1	0	1	1
225	0	2	2	0	0

MergeListenersMapper

Prepares the data for input to the final reducer function by mapping the `TrackId` to a `TrackStats` object with the number of unique listeners set.

- `MergeListenersMapper` **input**: The `UniqueListeners` job output.

TrackId	numListeners
IntWritable	IntWritable
222	1
223	1
225	2

- `MergeListenersMapper` **function**:
  - Create a new `TrackStats` object per track and set the `numListeners` attribute.
  - `map(225, 2)` $\rightarrow$ `<225, new TrackStats(2, 0, 0, 0, 0)>`
- `MergeListenersMapper` **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
222	1	0	0	0	0
223	1	0	0	0	0
225	2	0	0	0	0

MergeListenersMapper

```
public static class MergeListenersMapper extends
    Mapper<IntWritable, IntWritable, IntWritable, TrackStats> {

  public void map(IntWritable trackId, IntWritable uniqueListenerCount,
                                  Context context)
    throws IOException, InterruptedException
  {
    TrackStats trackStats = new TrackStats();
    trackStats.setListeners(uniqueListenerCount.get());
    context.write(trackId, trackStats);
  }
}
```

Final Reduce Stage: `SumTrackStatsReducer`

Finally, we have two partially defined `TrackStats` objects for each track. We can reuse the `SumTrackStatsReducer` to combine them and emit the final result.

- `SumTrackStatsReducer` **input**:
  - `key` is the TrackID output by the two mappers.
  - `value` is the iterator over the list of TrackStats associated with the track (in this case, one contains the unique listener count and the other contains the play, scrobble, radio listen, and skip counts).

TrackId	Iterable<TrackStats>
IntWritable	Iterable<TrackStats>
222	(1,0,0,0,0) (0,1,0,1,0)
223	(1,0,0,0,0) (0,1,0,1,1)
225	(2,0,0,0,0) (0,2,2,0,0)

- `SumTrackStatsReducer` **function**:
  - `reduce(225,'(2,0,0,0,0) (0,2,2,0,0)')` → `<225, (2,2,2,0,0)>`
- `SumTrackStatsReducer` **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	1	1	0	1	0
223	1	1	0	1	1
225	2	2	2	0	0

Final SumTrackStatsReducer

From the SumTrackStats job.

```
public static class SumTrackStatsReducer extends
    Reducer<IntWritable, TrackStats, IntWritable, TrackStats> {

  public void reduce(IntWritable trackId, Iterable<TrackStats> values,
                              Context context)
    throws IOException, InterruptedException
  {
    /* Hold totals for this track */
    TrackStats sum = new TrackStats();
    while (values.hasNext()) {
      TrackStats trackStats = (TrackStats) values.next();
      sum.setListeners(sum.getListeners() + trackStats.getListeners());
      sum.setPlays(sum.getPlays() + trackStats.getPlays());
      sum.setSkips(sum.getSkips() + trackStats.getSkips());
      sum.setScrobbles(sum.getScrobbles() + trackStats.getScrobbles());
      sum.setRadioPlays(sum.getRadioPlays() + trackStats.getRadioPlays());
    }
    context.write(trackId, sum);
  }
}
```

Putting it all together

- All job classes should extend `Configured` and implement `Tool`. `public class MergeResults extends Configured implements Tool`
- Then override the `run` method with specific job configuration.

```
public class MergeResults extends Configured implements Tool
{
  /* mapper and reducer classes */
  public int run(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Path sumInputDir = new Path(args[1] + Path.SEPARATOR_CHAR + "sumInputDir");
    Path listenersInputDir = new Path(args[1] + Path.SEPARATOR_CHAR + "listenersInputDir");
    Path output = new Path(args[1] + Path.SEPARATOR_CHAR + "finalSumOutput");

    Job job = new Job(conf, "merge-results");
    job.setJarByClass(UniqueListeners.class);

    MultipleInputs.addInputPath(job, sumInputDir,
            SequenceFileInputFormat.class, Mapper.class);
    MultipleInputs.addInputPath(job, listenersInputDir,
            SequenceFileInputFormat.class, MergeListenersMapper.class);

    job.setReducerClass(SumTrackStats.SumTrackStatsReducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(TrackStats.class);

    FileOutputFormat.setOutputPath(job, output);
    if (job.waitForCompletion(true))
      return 0;
    else
      return 1;
  }
}
```

Putting it all together

Hadoop provides a `ToolRunner` to simplify "job chaining". The driver class simply runs each job in the correct order using `ToolRunner`.

```
public class GenerateTrackStats {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.println("Usage: GenerateTrackStats <input path> <output path>");
      System.exit(0);
    }
    int exitCode = ToolRunner.run(new UniqueListeners(), args);
    if (exitCode == 0)
      exitCode = ToolRunner.run(new SumTrackStats(), args);
    else {
      System.err.println("GenerateTrackStats: UniqueListeners MapReduce phase failed!!");
      System.exit(1);
    }
    if (exitCode == 0)
      exitCode = ToolRunner.run(new MergeResults(), args);
    else {
      System.err.println("GenerateTrackStats: SumTrackStats MapReduce phase failed!!");
      System.exit(1);
    }
    System.exit(exitCode);
  }
}
```

References

- Thanks to *Marissa Hollingsworth* for the full last.fm code example and slides. Converted to new API by Amit Jain.
- Last.fm. The main website: http://www.last.fm/.
- *Hadoop: The Definitive Guide (3rd ed.)*. Tom White. O'Reilly.