Hadoop. MapReduce. Примеры Join

Возможности hadoop.mapreduce в реальных задачах

Join

- Задача состоит в том чтобы связать два набора данных по foreign ключу

id	systema	systemb
1	system1	system2
2	system1	system3
3	system3	system1

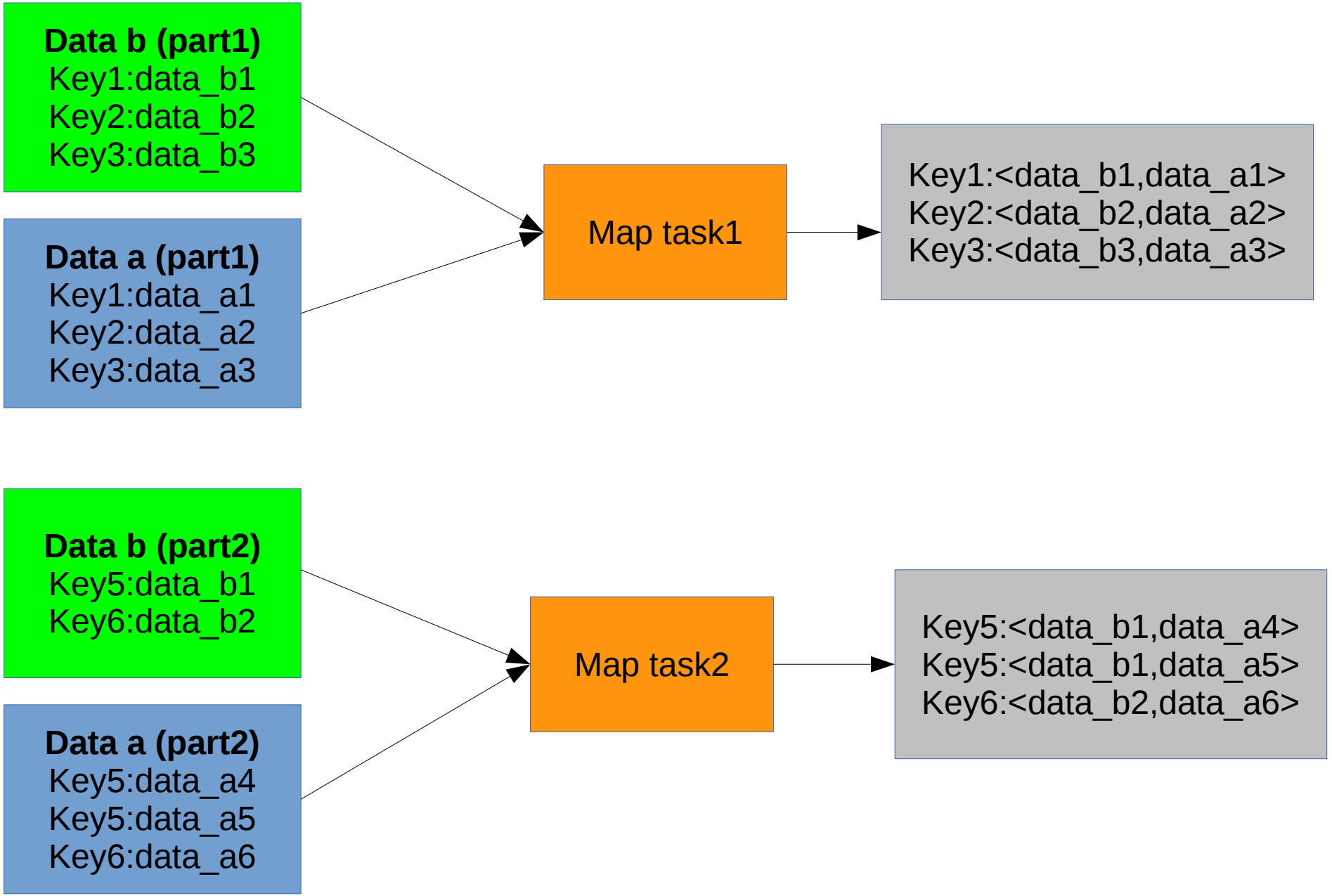
code	name
system1	МВД
system2	ФМС
system3	РосРеестр

id	systema	System a name	systemb
1	system1	МВД	system2
2	system2	ФМС	system3
3	system3	ВС	system1

Map side join

- Join осуществляется на шаге MAP
- На исходные данные накладываются требования :
  - Каждый набор данных должен быть разбит на одинаковое чилсло partitions
  - Данные в каждом наборе должны быть отсортированы по одному и тому же ключу – join key
  - Все данные с одним и тем же ключом должы быть в одном и том же partition

Map side join



Реализация map side join

- Используем класс CompositeInputFormat
- Составляется выражение, описывающее наборы данных и вид join

```
CompositeInputFormat.compose("inner",
                KeyValueTextInputFormat.class,
                file1,
                file2
        ));
```

- На вход mapper поступает TupleWritable в который входят данные из обоих наборов данных

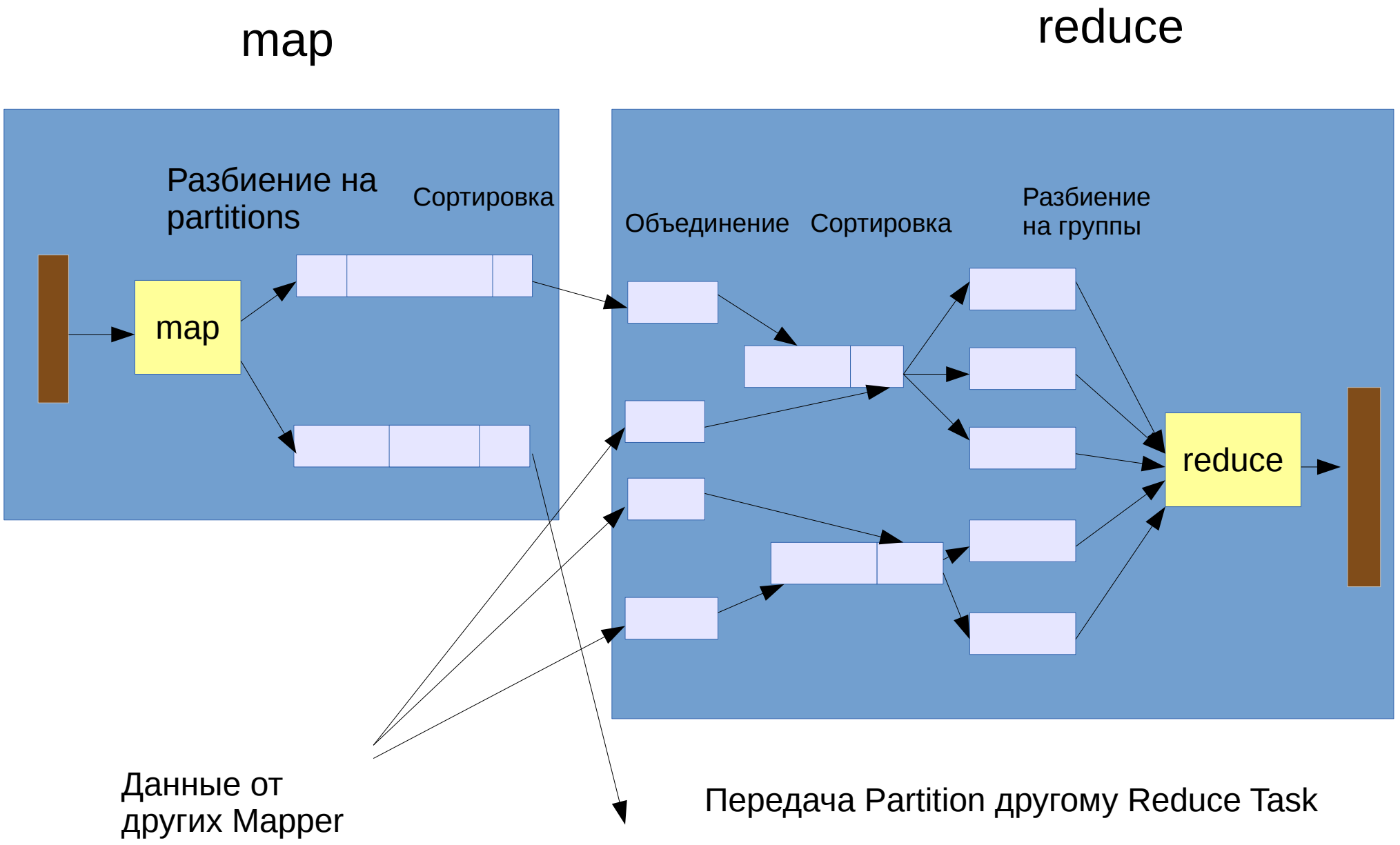
Пример Map Side join

```
JobConf conf = new JobConf(JoinJob.class);
conf.setJobName("map join");
conf.setInputFormat(CompositeInputFormat.class);
FileOutputFormat.setOutputPath(conf, new Path(args[2]));
conf.set("mapred.join.expr", CompositeInputFormat.compose("inner",
            KeyValueTextInputFormat.class,
            args[0],
            args[1]
      ));
conf.setMapperClass(MapJoinMapper.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
JobClient.runJob(conf);
```

Map side join mapper

```
public class MapJoinMapper extends MapReduceBase implements
Mapper<Text, TupleWritable, Text, Text> {
    @Override
    public void map(Text key, TupleWritable value,
                        OutputCollector<Text, Text> output,
                        Reporter reporter) throws IOException {
        Text call = (Text) value.get(0);
        Text system = (Text) value.get(1);
        output.collect(call, system);
    }
}
```

Детали MapReduce



Ход выполнения MAP

- Исходные данные для каждого Map task извлекаются с помощью InputFormat
- К исходным данным применяется функция MAP
- Результат на основании ключа и функции партиционирования разбивается на partitions (1 partition для каждого reduce task)
- Внутри partition производится сортировка по ключу
- Если задана функция combiner, то она применяется для каждого ключа и набора его значений
- Результат сохраняется на локальный жесткий диск компьютера где выполняется Map task

Partitioner

- Позволяет вручную определить partition для записи сформированной функцией map

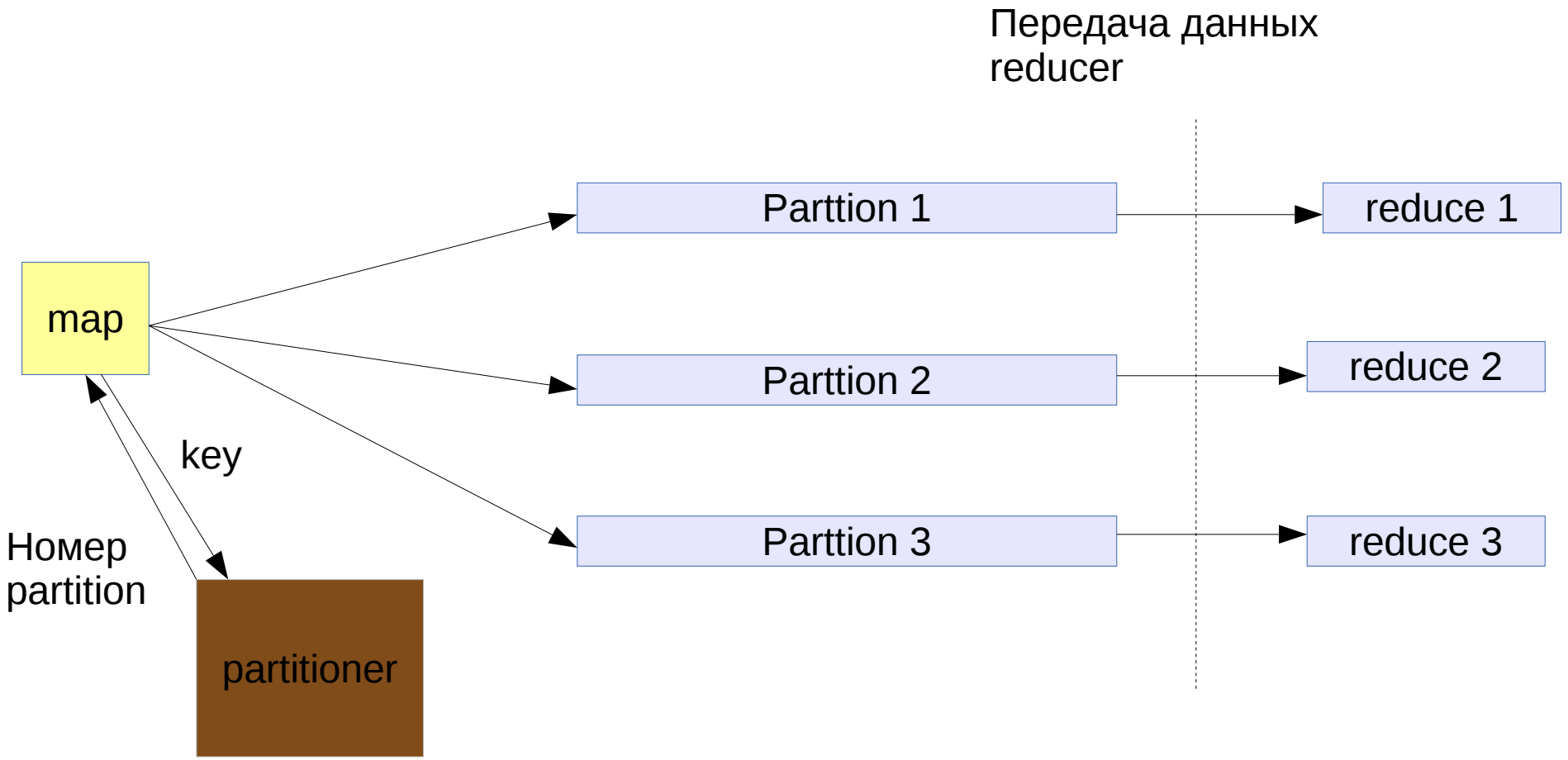
```
package org.apache.hadoop.mapreduce;
public abstract class Partitioner<KEY, VALUE> {
  public abstract int getPartition(KEY key, VALUE value, int numPartitions);
}
```

- Базовый partitioner работающий по умолчанию :

```
public class  HashPartitioner<K, V> extends Partitioner<K, V> {
  public int getPartition(K key, V value,  int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
}
```

-

Partitioner



Ход выполнения Reduce

- Task содеиняется со всеми Map Task и скачивает себе свои partitions
- Выполняется процедура merge:

  -Все partitions объединяются.

  -Происходит разбиение на группы. Либо по ключу, либо используя заданную функцию группировки

  -Группы сортируются
- Для каждой группы применяется функция reduce
- Результат сохраняется в HADOOP с помощью OutputFormat

Grouping Comparator

- Предназначен для разбития исходных данных reduce на группы
- Кажда группа подается на вход функции reduce независимо от других групп.
- Настраивается при создании job

  job.setGroupingComparatorClass(MyComparator.class);

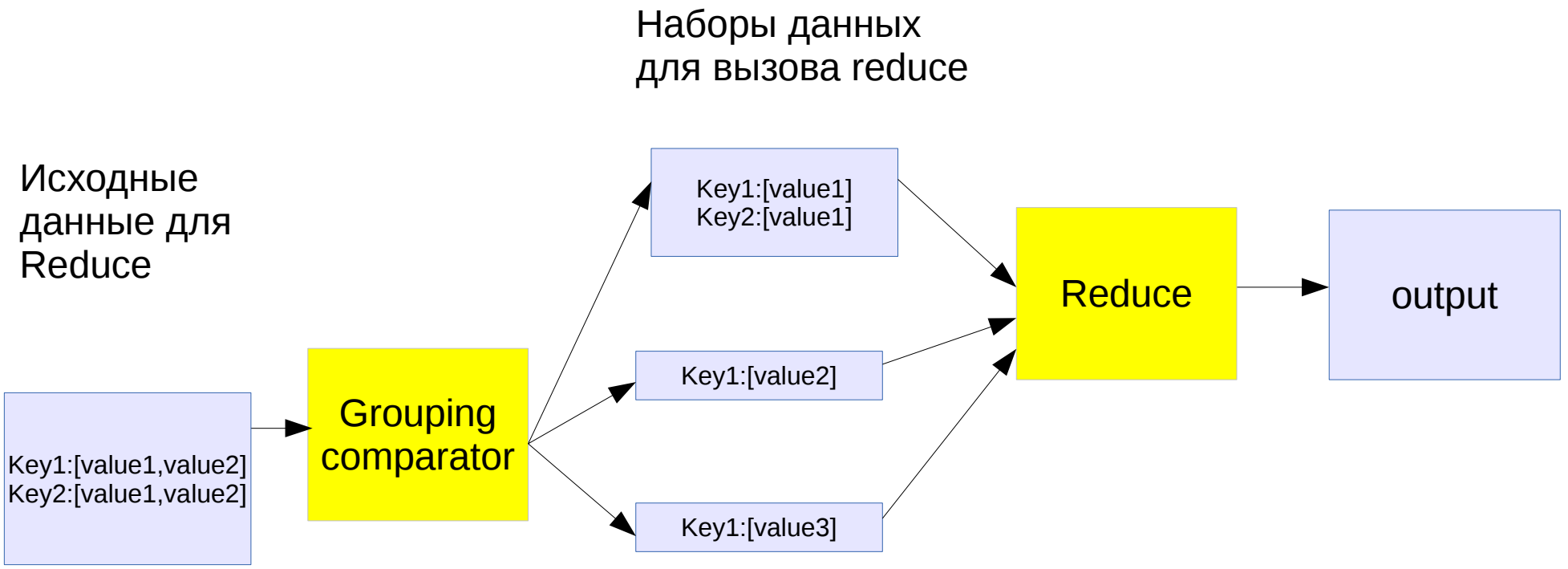
  –

  По умолчанию разбиение на группы не осуществляется – все входные данные reduce приходят в одном вызове
- Базовый класс

```
public class WritableComparator implements RawComparator {
...
 public int compare(WritableComparable a, WritableComparable b) {
    return a.compareTo(b);
 }}
```

Grouping comparator

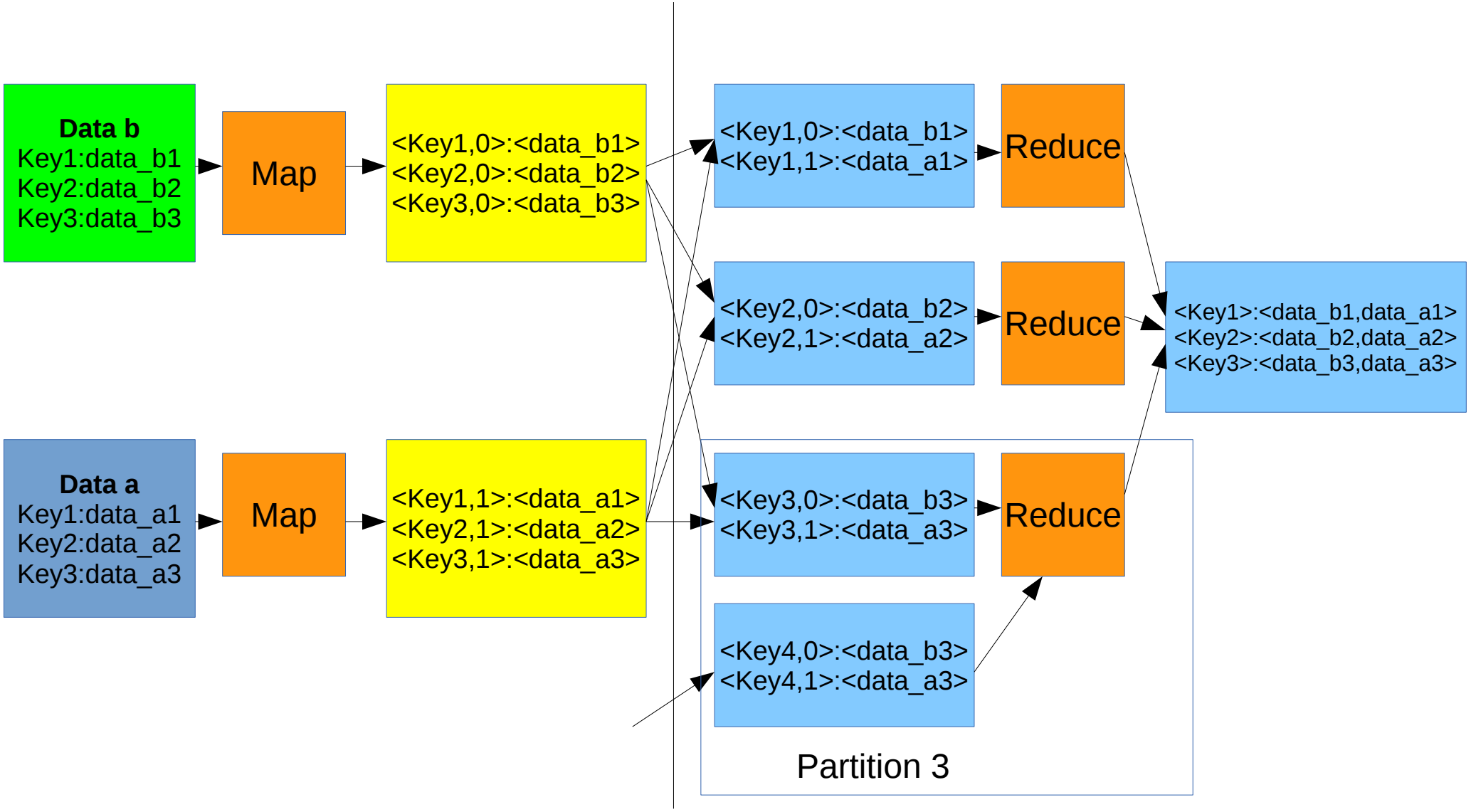
позволяет изменить выборку для Redece



Reduce side join

- Менее эффективен чем map side join
- Не накладывает жестких требований на входные данные
- Использует MultipleInputs
- MultipleInputs позволяет добавить несколько наборов входных данных и каждому набору указать свой mapper
- В качестве ключа используется составной ключ :
  - поле по которому происходит связывание (join key) + флаг вида данных (используем 0 для набора данных справочника и 1 для набора данных основной таблицы)
- Партиционирование делается аналогично полной сортировке но только по первой части составного ключа (join key) – используется ручной partitioner
- Группировка делается по первой части ключа ручным GroupingComparator
- На вход reducer подаются наборы данных в которых в первой строчке будет строка справочника, а в последующих строки основной таблицы

Reduce side join



Пример reduce side join

```
public class CallsJoinMapper extends Mapper<LongWritable, Text, TextPair, Text> {
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        ServiceCall call = new ServiceCall(value);
        context.write(new TextPair(call.getSystemA().toString(),"1"),
                        new Text(call.toString()));
    }
}
public class SystemsJoinMapper extends Mapper<LongWritable, Text, TextPair, Text> {
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        SystemInfo system = new SystemInfo(value);
        context.write(new TextPair(system.getSystemCode().toString(),"0"), new Text(system.toString()));
    }
}
```

Reducer

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {
    @Override
    protected void reduce(TextPair key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        Iterator<Text> iter = values.iterator();
        Text systemInfo = new Text(iter.next());
        while (iter.hasNext()) {
            Text call = iter.next();
            Text outValue = new Text(call.toString() + "\t" + systemInfo.toString());
            context.write(key.getFirst(), outValue);
        }
    }
}
```

job

```
Job job = Job.getInstance();
job.setJarByClass(JoinJob.class);
job.setJobName("JoinJob sort");
MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class, CallsJoinMapper.class);
MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class, SystemsJoinMapper.class);

FileOutputFormat.setOutputPath(job, new Path(args[2]));
job.setPartitionerClass(TextPair.FirstPartitioner.class);
job.setGroupingComparatorClass(TextPair.FirstComparator.class);
job.setReducerClass(JoinReducer.class);
job.setMapOutputKeyClass(TextPair.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setNumReduceTasks(2);
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Distributed cache

- Позволяет добавлять в ресурсы job файлы с данными, архивами и классами.
- Перед запуском task, Hadoop скачивает из хранилища все файлы добавленные в кэш.
- Во время работы mapper и reducer файлы кеша доступны через api
- Эффективный способ join, когда один из справочников гарантировано поместится в памяти Task.

Distributed cache