Pós-Graduação em Computação Distribuída e Ubíqua

INF628 - Engenharia de Software para Sistemas Distribuídos
Design Patterns para MapReduce (parte I)

Sandro S. Andrade
sandroandrade@ifba.edu.br

Objetivos

- Apresentar o MapReduce enquanto modelo computacional / estilo arquitetural para computação distribuída
- Apresentar e discutir os principais design patterns utilizados em aplicações MapReduce
- Proporcionar uma vivência prática acerca desta tecnologia utilizando o framework Hadoop

Introdução

- O que é o MapReduce ?
  - É um paradigma de computação para processamento de dados que residem em uma grande quantidade de computadores
- Criado pelo Google e popularizado em 2004, no artigo "MapReduce: Simplified Data Processing on Large Clusters"
- Logo depois o projeto Hadoop foi criado por Doug Cutting, financiado pelo Yahoo! e passou a integrar a Apache Foundation

Introdução

- Outros projetos opensource funcionam com base no Hadoop:
  - Pig, Hive, HBase, Mahout e ZooKeeper
  - Hadoop como kernel de um novo SO distribuído
- Visão geral do MapReduce/Hadoop:
  - Jobs são implementados como tarefas de map e tarefas de reduce que executam em um cluster
  - Cada tarefa processa um pequeno subconjunto dos dados, de modo que a carga é dividida no cluster

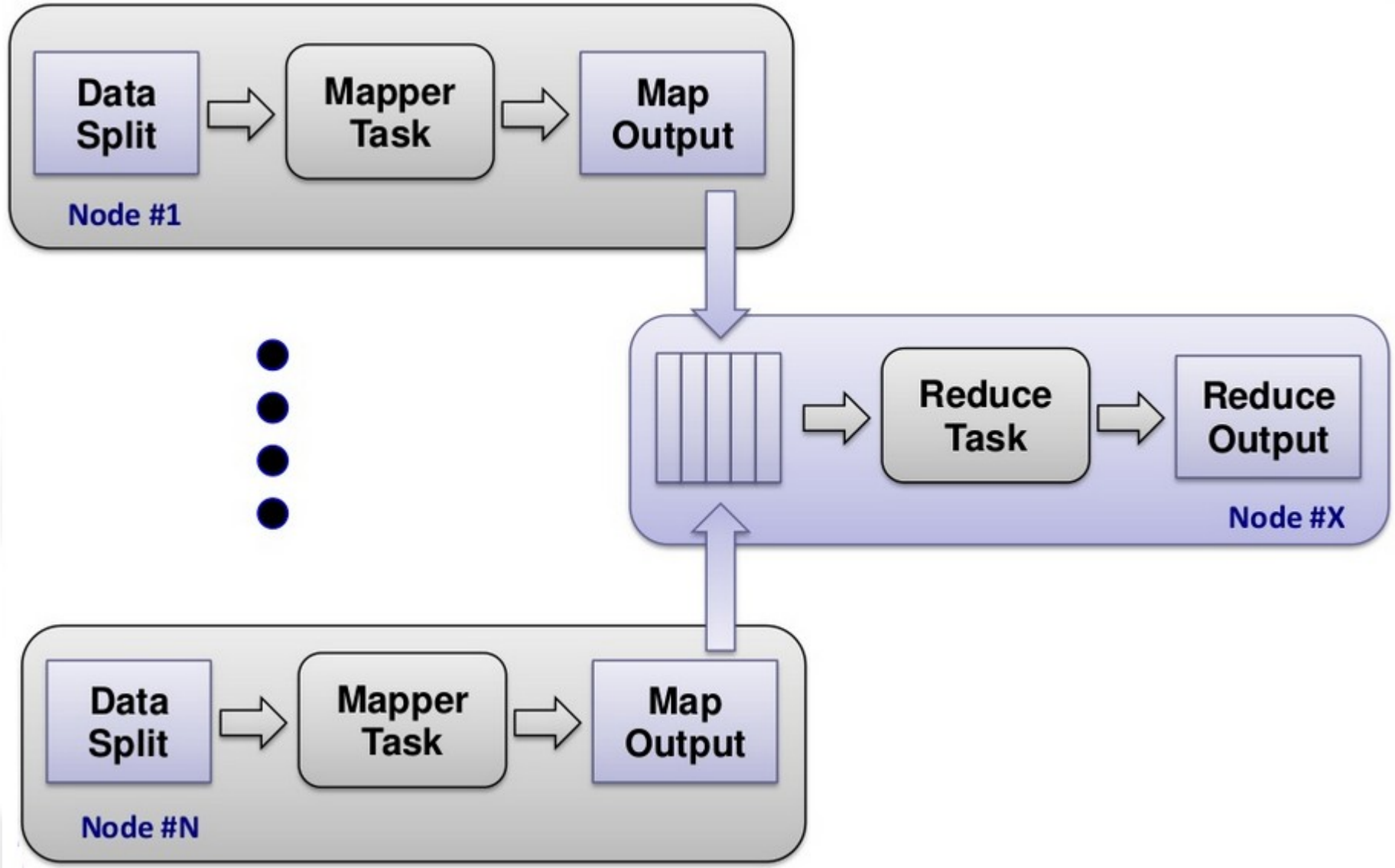
Introdução

- Visão geral do MapReduce/Hadoop:
  - A tarefa de map geralmente faz carga, parse, transformação e filtragem de dados
  - A tarefa de reduce é responsável por tratar um subconjunto dos dados de saída das tarefas de map
  - Dados intermediários são copiados das tarefas map para as tarefas reduce, de modo a agregar e agrupar os dados de saída
  - Um amplo conjunto de problemas pode ser resolvido com este paradigma, desde agregações numéricas simples até joins e produtos cartesianos

Introdução

- Visão geral do MapReduce/Hadoop:
  - A entrada para um job MapReduce é um conjunto de arquivos, armazenados no Hadoop Distributed File System (HDFS)
  - Estes arquivos são particionados de acordo com um input format, que define como separar um arquivo em input splits
  - Um input split é um conjunto de bytes que representa um bloco do arquivo, a ser carregado por uma tarefa de map

Introdução


Data Split
Mapper Task
Map Output
Node #1
Reduce Task
Reduce Output
Node #X
Data Split
Mapper Task
Map Output
Node #N

Introdução

- Visão geral do MapReduce/Hadoop:
  - Cada tarefa de map é formada pelas seguintes fases:
    - Record Reader, Mapper, Combiner e Partitioner
  - A saída das tarefas de map - denominadas keys/values intermediários - é enviada para as tarefas de reduce
  - Cada tarefa de reduce é formada pelas seguintes fases:
    - Shuffle and Sort, Reducer e Output Format
  - Os nós que executam as tarefas de map são, preferencialmente, os nós que contém os dados a serem processados

Introdução

- Fases de uma tarefa de map:
  - Record Reader: traduz um input split, gerado por um input format, em um conjunto de registros. Transforma dados em registros, mas não processa estes registros. Envia os dados para o mapper, na forma key/values
  - Mapper: executa código fornecido pelo desenvolvedor para transformar os pares key/value de entrada em novos pares (intermediários) key/value. A chave dos novos pares é a informação em torno da qual os dados serão agrupados e o valor é a informação pertinente à análise da tarefa de reduce. Diferentes patterns definem diferentes semânticas para key/values

Introdução

- Fases de uma tarefa de map:
  - Combiner: é um reducer opcional, instalado localmente na mesma máquina do mapper. Realiza agrupamento de dados, já na fase de map. Reduz significativamente a quantidade de dados a ser trafegada na rede
    - Ex: envia (hello world, 3) em vez de (hello world, 1) três vezes
  - Partitioner: divide os key/values intermediários em fragmentos (shards), um para cada tarefa de reduce
    - key.hashCode() % (#reducers)
    - Cada fragment é escrito no sistema de arquivos local e espera ser lido pela tarefa de reduce respectiva

Introdução

- Fases de uma tarefa de reduce:
  - Shuffle and Sort:
    - Realiza o download dos arquivos gerados por todos os partitioners para a máquina onde a tarefa de reduce está sendo executada
    - Os dados são então ordenados pela chave em uma única lista de dados
    - A operação é realizada automaticamente pelo framework, mas pode-se definir como as chaves são ordenadas e agrupadas, através da definição de um objeto Comparator

Introdução

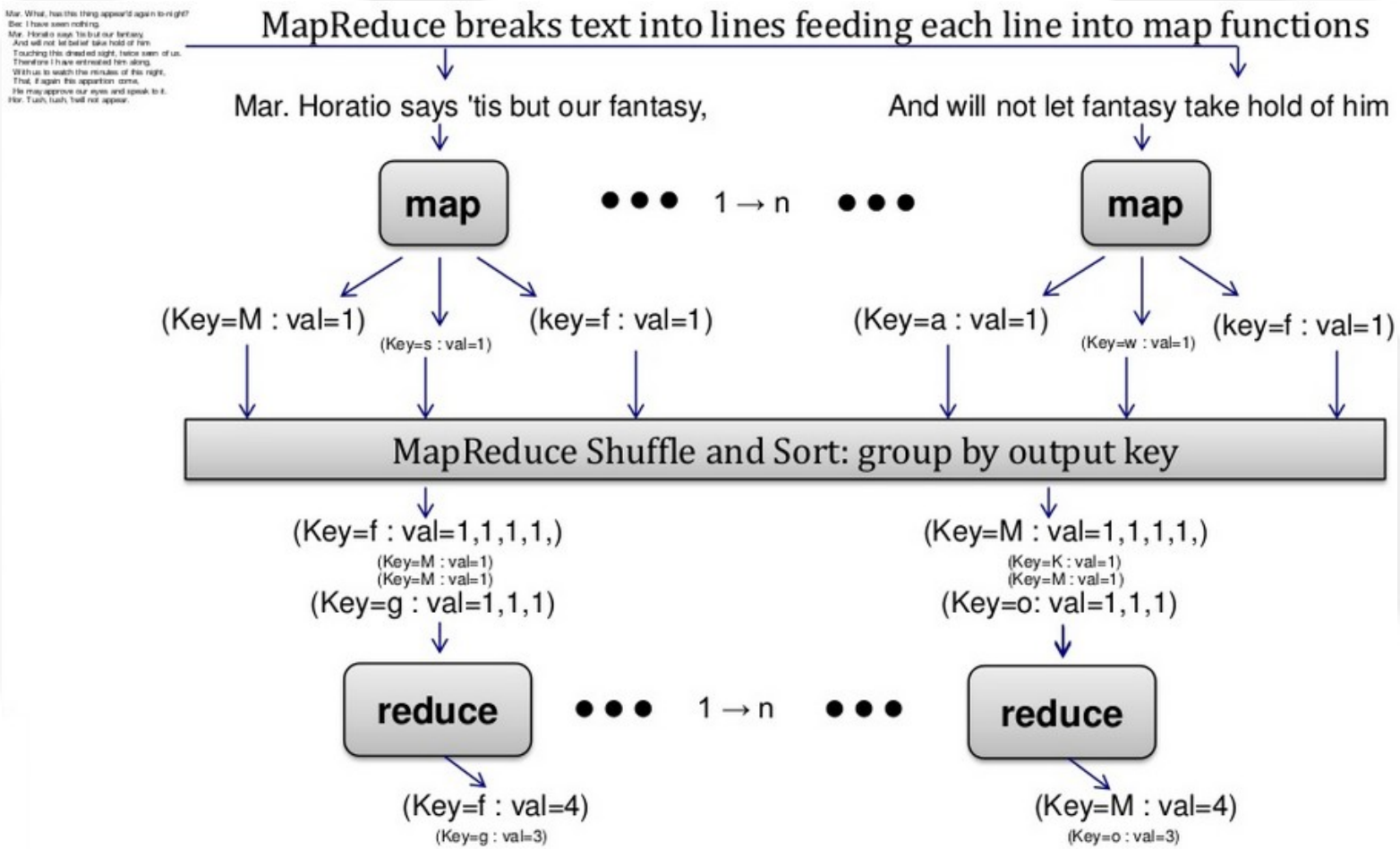
- Fases de uma tarefa de reduce:
  - Reduce:
    - Recebe os dados agrupados como entrada e executa a função de redução uma vez para cada agrupamento de chave
    - A função de redução recebe, como parâmetros, a chave e um iterator para todos os valores daquela chave
    - A função de redução pode realizar qualquer processamento, como agregação, filtragem e combinação
    - A função de redução gera zero ou mais pares de key/values

Introdução

- Fases de uma tarefa de reduce:
  - Output Format:
    - Traduz os key/values gerados pela redução em registros a serem escritos no sistema de arquivos
    - O comportamento default é separar chaves e valores com um tab e separar registros com uma quebra de linha

Introdução

- Exemplo – job StartsWith Count:

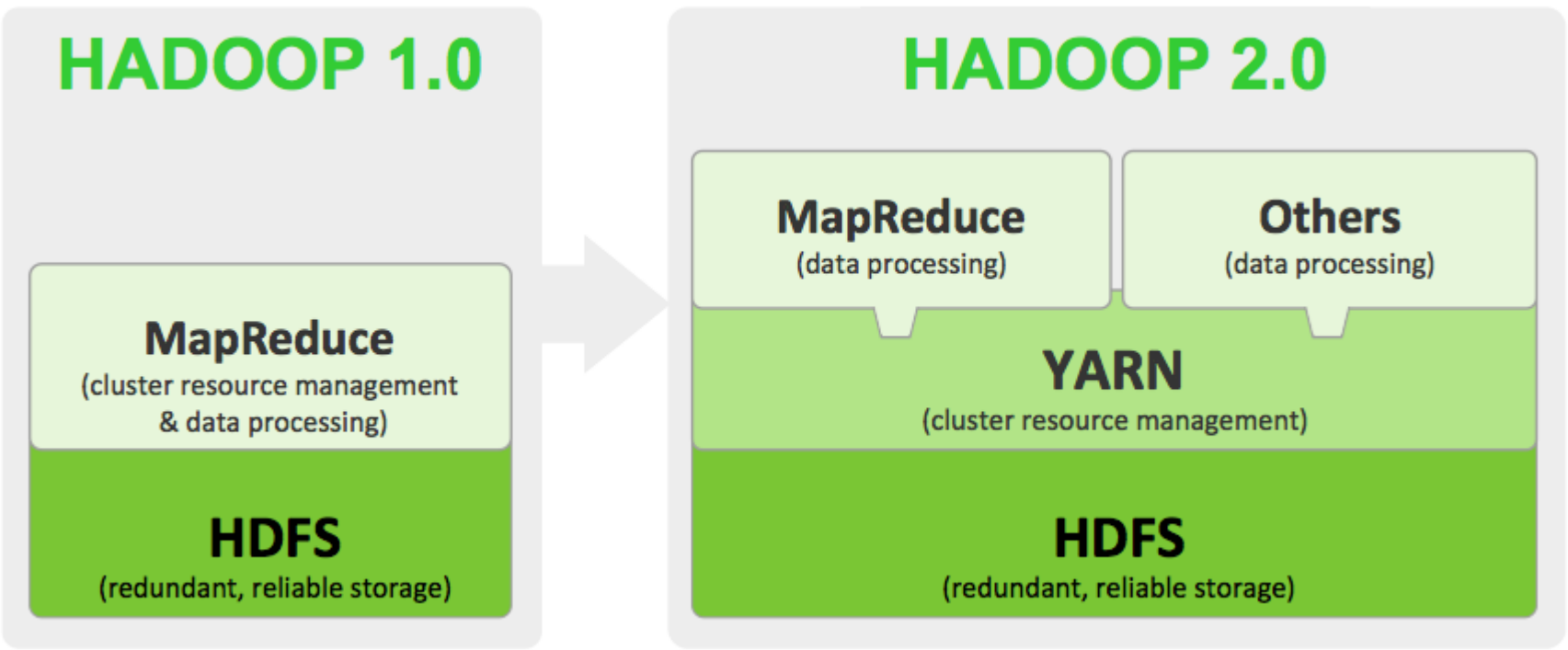


Introdução

- Hadoop:
  - Sistema de arquivos distribuído (HDFS)
  - Framework para execução de aplicações paralelas (YARN)
    - MapReduce como um dos modelos de computação distribuída suportados no YARN
  - OpenSource
  - Mantido pela Apache Foundation

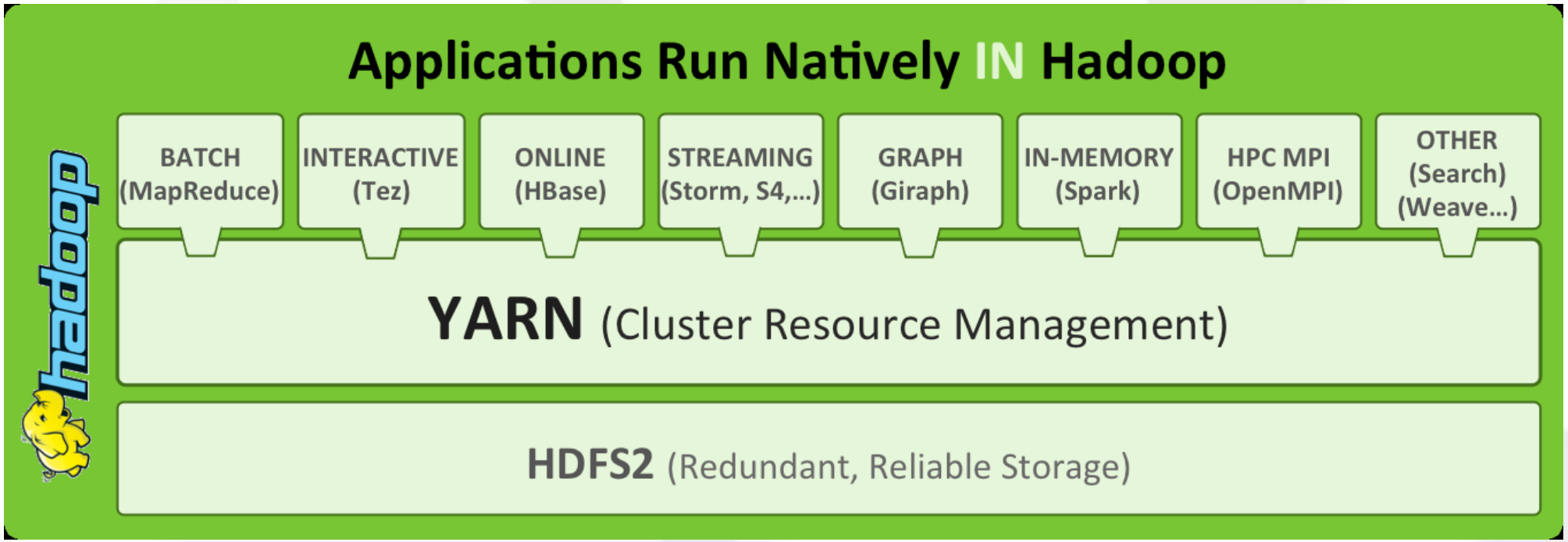
Introdução

- Hadoop 1 x Hadoop 2:



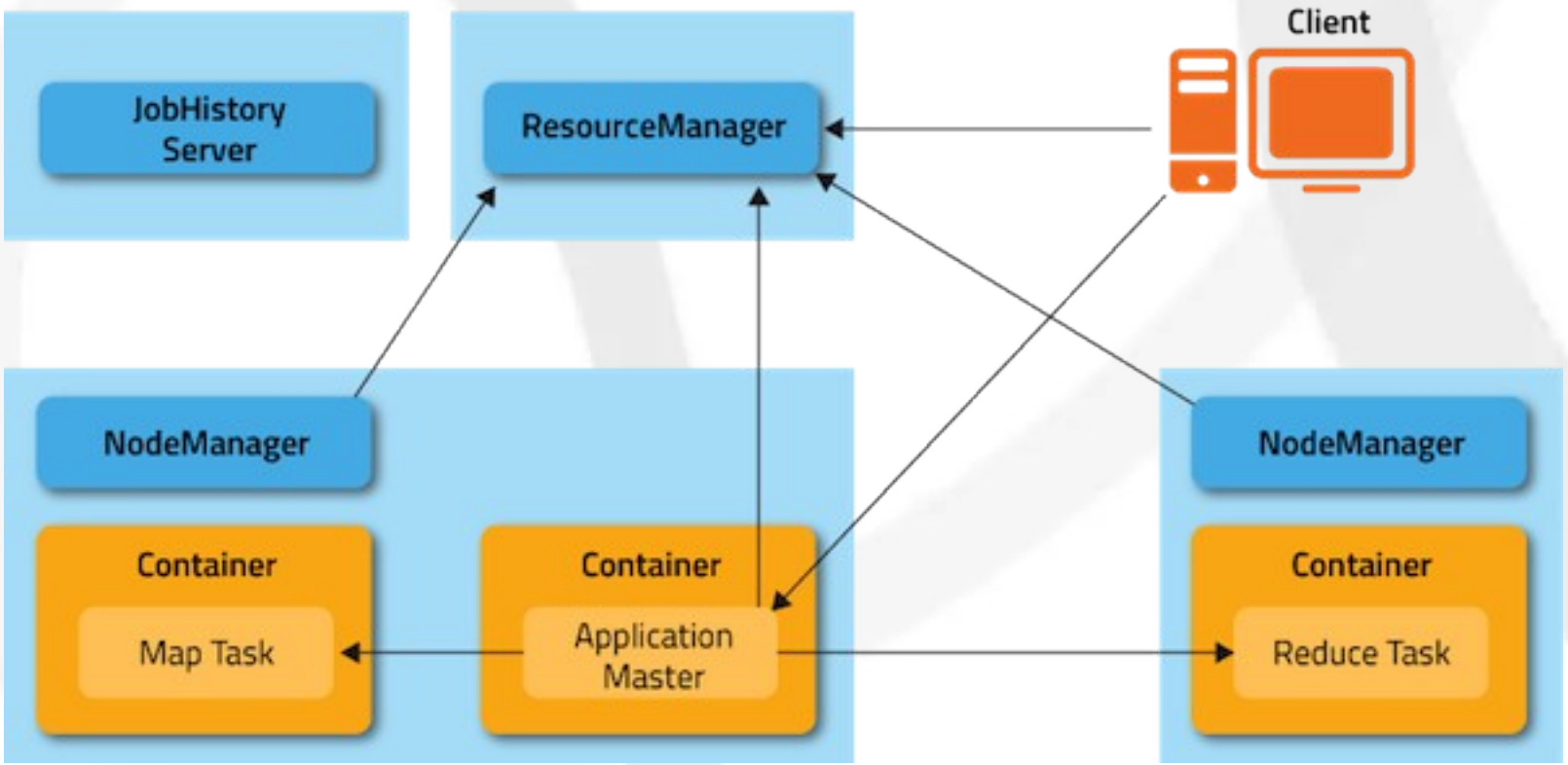
Introdução

- Tecnologias viabilizadas pelo Hadoop:



Introdução

- MapReduce no Hadoop 2:



Introdução

- Demonstração – job WordCount no Hadoop

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount <in> <out>");
    System.exit(2);
  }
  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Introdução

- Demonstração – job WordCount no Hadoop

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

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Introdução

- Demonstração – job WordCount no Hadoop

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

Design Patterns para MapReduce

Design Patterns para MR

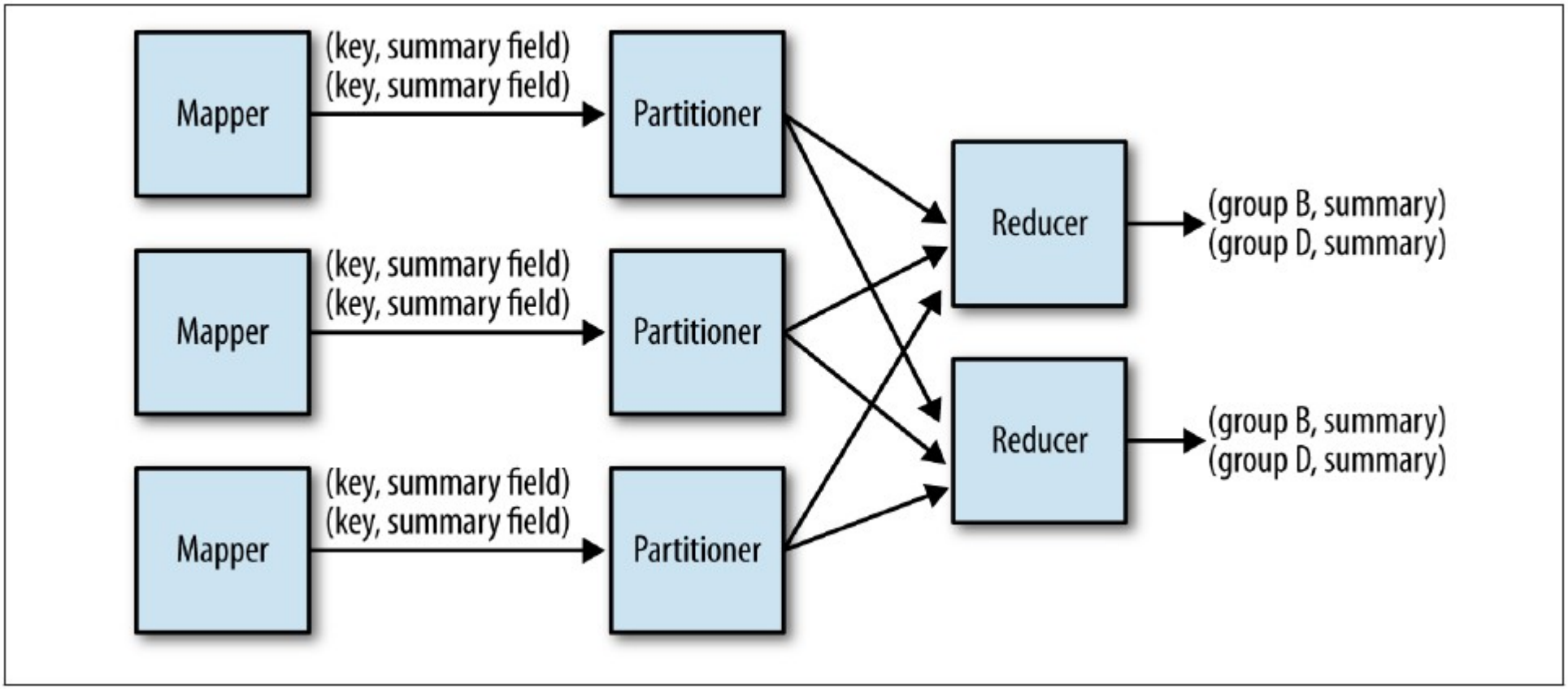
- NUMERICAL SUMMARIZATIONS
  - Intenção:
    - Agrupar registros através de um campo-chave e calcular um agregado numérico (ex: mínimo, máximo, média, mediana, etc) por grupo: $\lambda=\theta(v_1, v_2, ..., v_3)$
  - Motivação:
    - Muitos datasets atuais são grandes demais para se obter alguma informação importante através da investigação individual de registros
    - Exs: obtenção de padrões de uso de servidores web a partir dos seus arquivos de log, número de logins por hora do dia, views de propagandas por tipo, etc

Design Patterns para MR

- NUMERICAL SUMMARIZATIONS
  - Aplicabilidade:
    - Quando se está manipulando dados numericos ou realizando contagem e os dados podem ser agrupados por campos específicos

Design Patterns para MR

- NUMERICAL SUMMARIZATIONS
  - Estrutura:



Design Patterns para MR

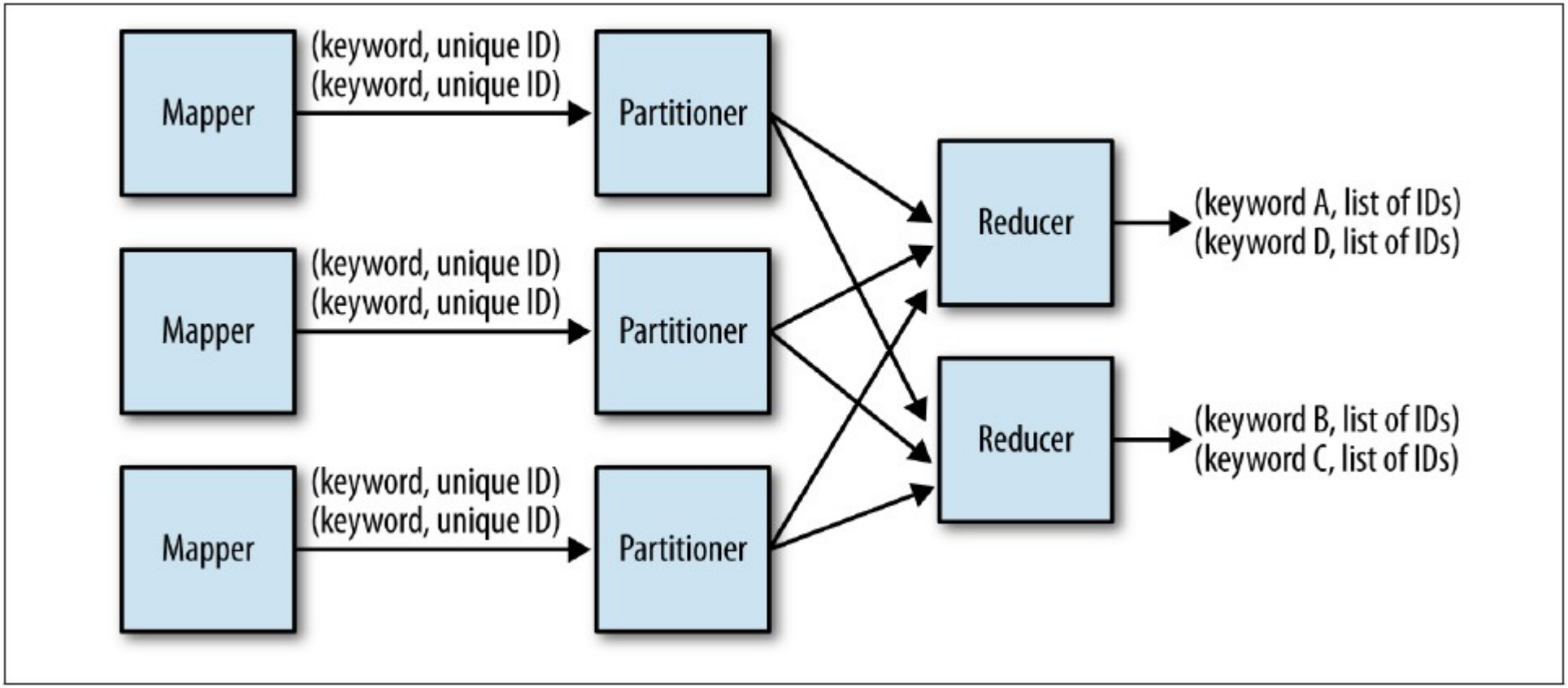
- NUMERICAL SUMMARIZATIONS
  - Consequências:
    - A saída do job será um conjunto de partes de arquivos contendo um único registro por grupo de entrada da redução. Cada registro irá conter a chave e todos os valores agregados
  - Usos conhecidos:
    - Word Count, Record Count, Min/Max/Count e Average/Median/Standard Deviation
  - Semelhanças: GROUP BY no SQL:
    - select min(numericalcol1), max(numericalcol1), count(*) from table group by groupcol2;
  - OBS: partitioners and data skew

Design Patterns para MR

- INVERTED INDEX SUMMARIZATIONS
  - Intenção:
    - Gerar um index a partir de um dataset para permitir buscas mais rápidas ou suportar enriquecimento de dados
  - Motivação:
    - É conveniente indexar grande bases de dados a partir de keywords (ex: páginas web com determinada palavra)
    - Construir o index leva tempo, mas após isso as buscas de tornam mais rápidas
  - Aplicabilidade:
    - Índices invertidos devem ser usados quando respostas rápidas a operações de busca são necessárias

Design Patterns para MR

- INVERTED INDEX SUMMARIZATIONS
  - Estrutura:



Design Patterns para MR

- INVERTED INDEX SUMMARIZATIONS
  - Consequências:
    - A saída do job será um conjunto de partes de arquivos que contêm o mapeamento de um valor de keywork para todos os IDs de registros que contêm o valor de keyword
  - Exemplo: índice invertido para referências a Wikipedia em posts do StackOverflow

Design Patterns para MR

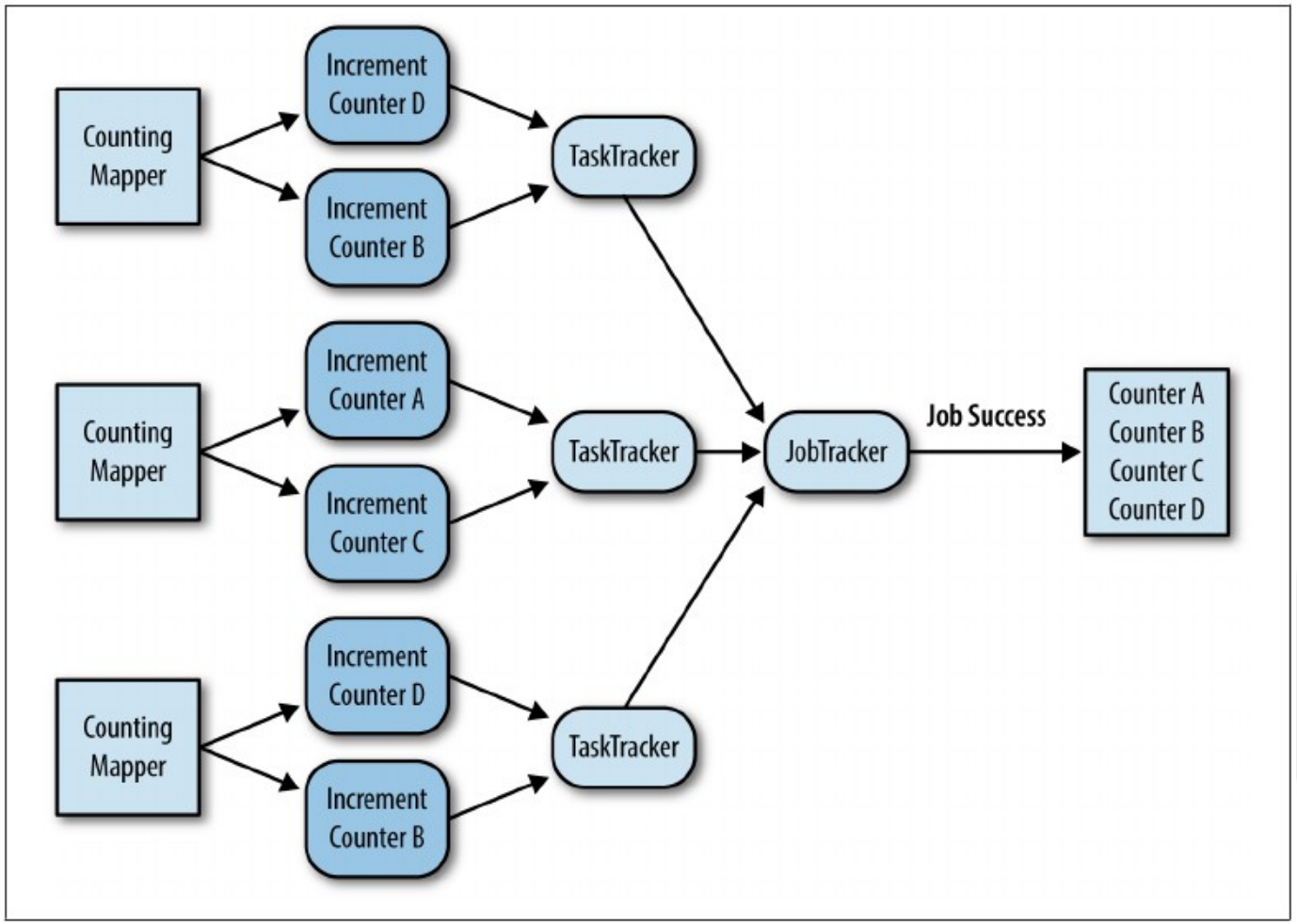
- COUNTING WITH COUNTERS
  - Intenção:
    - Disponibilizar um meio eficiente para obter sumarizações de grandes conjuntos de dados
  - Motivação:
    - Use o mecanismos de contagem do framework (hadoop) para obter sumarizações sem gerar nenhum par chave-valor e sem utilizar tarefas de reduce
    - Alguns contadores são automaticamente criados pelo hadoop (ex: # bytes lidos/gravados) e outros podem ser criados pelo desenvolvedor

Design Patterns para MR

- COUNTING WITH COUNTERS
  - Aplicabilidade:
    - Quando deseja-se obter contadores ou sumarizações de grandes conjuntos de dados e o número total de contadores é pequeno

Design Patterns para MR

- COUNTING WITH COUNTERS
  - Estrutura:



Design Patterns para MR

- COUNTING WITH COUNTERS
  - Usos conhecidos:
    - Contagem de número de registros, contagem de um número pequeno de instâncias únicas, sumarizações

Design Patterns para MR

- COUNTING WITH COUNTERS
  - Exemplo:

```
// For each token
boolean unknown = true;
for (String state : tokens) {

    // Check if it is a state
    if (states.contains(state)) {
        // If so, increment the state's counter by 1
        // and flag it as not unknown
        context.getCounter(STATE_COUNTER_GROUP, state)
                .increment(1);
        unknown = false;
        break;
    }
}

// If the state is unknown, increment the UNKNOWN_COUNTER counter
if (unknown) {
    context.getCounter(STATE_COUNTER_GROUP, UNKNOWN_COUNTER)
            .increment(1);
}
```

Design Patterns para MR

- COUNTING WITH COUNTERS
  - Driver code:

```
int code = job.waitForCompletion(true) ? 0 : 1;

if (code == 0) {
    for (Counter counter : job.getCounters().getGroup(
            CountNumUsersByStateMapper.STATE_COUNTER_GROUP)) {
        System.out.println(counter.getDisplayName() + "\t"
                + counter.getValue());
    }
}

// Clean up empty output directory
FileSystem.get(conf).delete(outputDir, true);

System.exit(code);
```

Design Patterns para MR

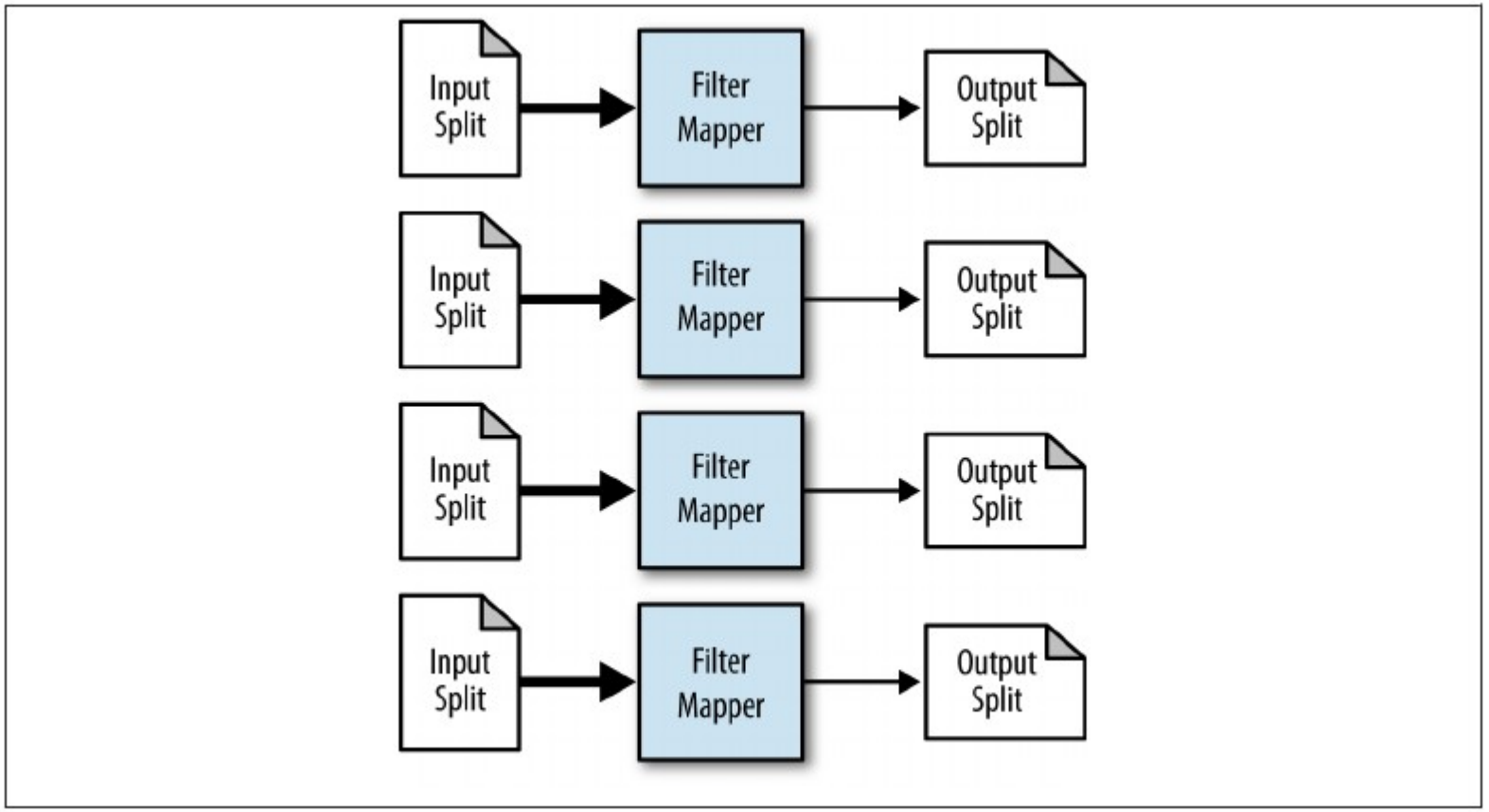
- FILTERING
  - Intenção:
    - Selecionar, no conjunto de dados, somente aqueles registros que são de interesse. Utiliza uma função de avaliação *f* que recebe um registro como argumento e retorna um valor booleano
  - Motivação:
    - O dataset é extremamente grande e deseja-se obter um subconjunto de dados para análise
    - Ex: obter somente os comentários que tem a ver com hadoop (possuem "hadoop" no texto ou estão marcados com uma determinada tag)
    - Os jobs de filtering são map-only

Design Patterns para MR

- FILTERING
  - Aplicabilidade:
    - Qualquer situação onde registros podem ser classificados através de critérios bem definidos

Design Patterns para MR

- FILTERING
  - Estrutura:



Design Patterns para MR

- FILTERING
  - Usos conhecidos:
    - Visão mais detalhada dos dados
    - Rastrear um fluxo de eventos (ex: comportamento de um determinado usuário dentre todos os outros)
    - Grep distribuído
    - Limpeza de dados (ex: remoção de registros incompletos ou em formato incorreto)
    - Amostragem randômica simples (função de avaliação retorna true com alguma probabilidade)
    - Remoção de dados com score baixo
  - Semelhanças:
    - select * from table where value < 3;

Design Patterns para MR

- FILTERING
  - Exemplo (distributed grep):

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

    private String mapRegex = null;

    public void setup(Context context) throws IOException,
        InterruptedException {

        mapRegex = context.getConfiguration().get("mapregex");
    }

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

        if (value.toString().matches(mapRegex)) {
           context.write(NullWritable.get(), value);
        }
    }
}
```

Design Patterns para MR

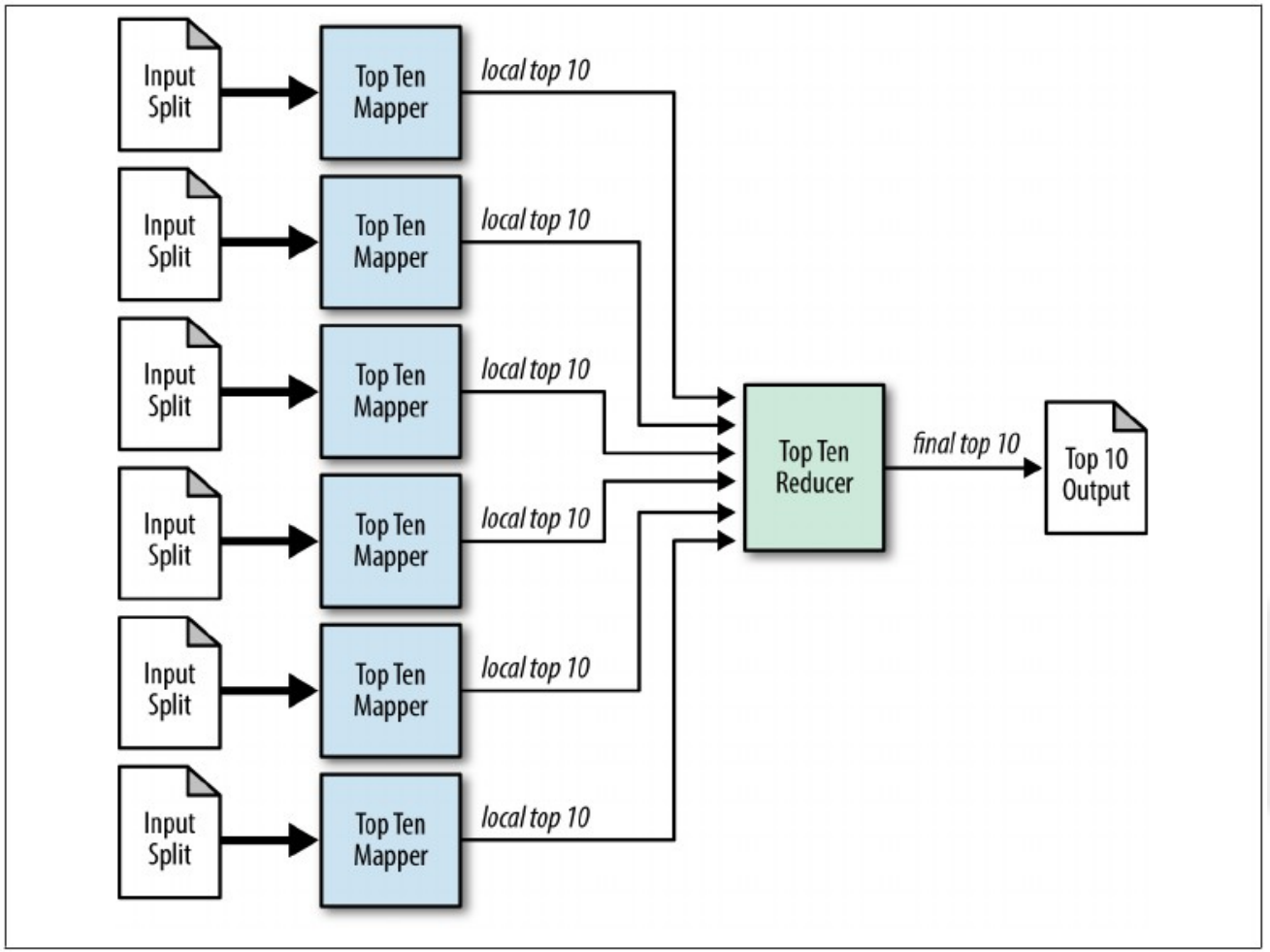
- TOP TEN
  - Intenção:
    - Recuperar um número relativamente pequeno de top K registros, de acordo com algum critério de ranking, independente de quão grande seja o dataset
  - Motivação:
    - No MapReduce, obtenção de ordem total (para então recuperar os top K) é extremamente custoso
    - Este pattern encontra os top K registros sem requerer a ordenação dos dados
  - Exemplos: quais posts foram mais curtidos no Facebook ? Quais os usuários mais antigos do seu site ? Qual a maior compra já feita na Amazon ?

Design Patterns para MR

- TOP TEN
  - Aplicabilidade:
    - O padrão requer uma função de comparação de registros, ou seja, dados dois registros deve-se saber qual é o "maior"
    - O número de registros de saída deve ser significativamente menor que o número de registros de entrada. Caso contrário, uma ordenação total seria mais sensato

Design Patterns para MR

- TOP TEN
  - Estrutura:



Design Patterns para MR

- TOP TEN
  - Usos conhecidos:
    - Análise de outliers
    - Seleção de dados interessantes
    - Dashboards “cativantes”
  - Semelhanças:
    - select * from table order by col4 desc limit 10;

Design Patterns para MR

- TOP TEN
  - Exemplo: top ten users by reputation (mapper)

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

  // Stores a map of user reputation to the record
  private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();

  public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
    Map<String, String> parsed = transformXmlToMap(value.toString());

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

    // Add this record to our map with the reputation as the key
    repToRecordMap.put(Integer.parseInt(reputation), new Text(value));

    // If we have more than ten records, remove the one with the lowest rep
    // As this tree map is sorted in descending order, the user with
    // the lowest reputation is the last key.
    if (repToRecordMap.size() > 10) {
```

Design Patterns para MR

- TOP TEN
  - Exemplo: top ten users by reputation (mapper)

```
      repToRecordMap.remove(repToRecordMap.firstKey());
    }
  }

  protected void cleanup(Context context) throws IOException,
      InterruptedException {
    // Output our ten records to the reducers with a null key
    for (Text t : repToRecordMap.values()) {
      context.write(NullWritable.get(), t);
    }
  }
}
```

Design Patterns para MR

- TOP TEN
  - Exemplo: top ten users by reputation (reducer)

```
public static class TopTenReducer extends
    Reducer<NullWritable, Text, NullWritable, Text> {

  // Stores a map of user reputation to the record
  // Overloads the comparator to order the reputations in descending order
  private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();

  public void reduce(NullWritable key, Iterable<Text> values,
      Context context) throws IOException, InterruptedException {
    for (Text value : values) {
      Map<String, String> parsed = transformXmlToMap(value.toString());

      repToRecordMap.put(Integer.parseInt(parsed.get("Reputation")),
          new Text(value));

      // If we have more than ten records, remove the one with the lowest rep
      // As this tree map is sorted in descending order, the user with
      // the lowest reputation is the last key.
      if (repToRecordMap.size() > 10) {
        repToRecordMap.remove(repToRecordMap.firstKey());
      }
    }
```

Design Patterns para MR

- TOP TEN
  - Exemplo: top ten users by reputation (reducer)

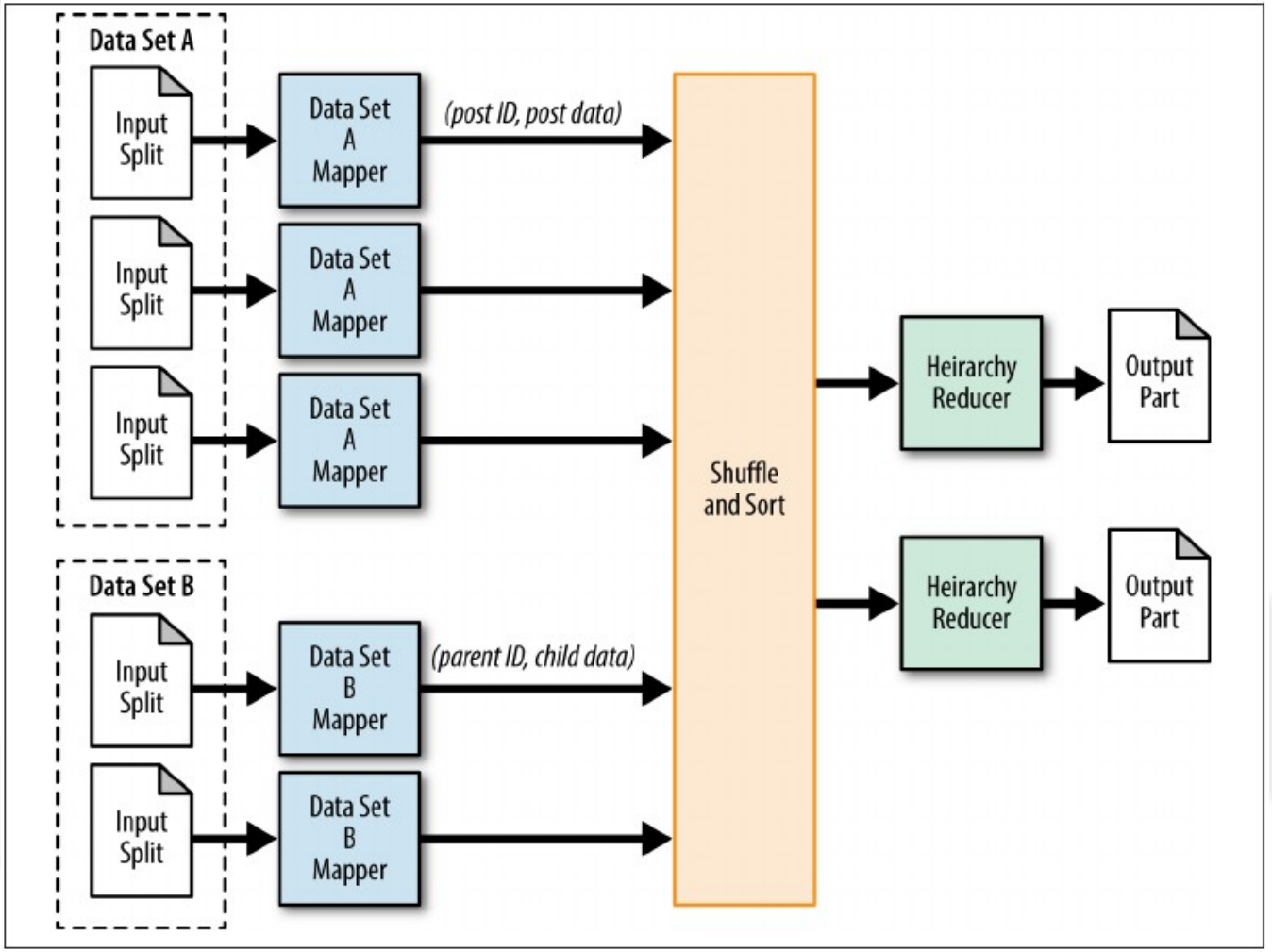
```
    for (Text t : repToRecordMap.descendingMap().values()) {
      // Output our ten records to the file system with a null key

      context.write(NullWritable.get(), t);
    }
  }
}
```

Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Intenção:
    - Transformar dados row-based em dados hierárquicos (ex: JSON ou XML)
  - Motivação:
    - Ao migrar dados de um RDBMS para o Hadoop é interessante reformatar os dados, de modo a tornar as operações mais rápidas
    - Dados desnormalizados funcionam melhor, pois evita-se a realização de joins custosos
  - Aplicabilidade:
    - Quando os dados são ligados por algum tipo de chave estrangeira e são estruturados e row-based

Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Estrutura:



Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Exemplo (driver):

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "PostCommentHierarchy");
    job.setJarByClass(PostCommentBuildingDriver.class);

    MultipleInputs.addInputPath(job, new Path(args[0]),
            TextInputFormat.class, PostMapper.class);

    MultipleInputs.addInputPath(job, new Path(args[1]),
            TextInputFormat.class, CommentMapper.class);

    job.setReducerClass(UserJoinReducer.class);

    job.setOutputFormatClass(TextOutputFormat.class);
    TextOutputFormat.setOutputPath(job, new Path(args[2]));

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(Text.class);

    System.exit(job.waitForCompletion(true) ? 0 : 2);
}
```

Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Exemplo (mapper1 - posts):

```
public static class PostMapper 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 = MRDPUtils.transformXmlToMap(value
                .toString());

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

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

Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Exemplo (mapper2 - comments):

```
public static class CommentMapper 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 = MRDPUtils.transformXmlToMap(value
                .toString());

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

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

Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Exemplo (reducer):

```
public static class PostCommentHierarchyReducer extends
        Reducer<Text, Text, Text, NullWritable> {

    private ArrayList<String> comments = new ArrayList<String>();
    private DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    private String post = null;

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        // Reset variables
        post = null;
        comments.clear();

        // For each input value
        for (Text t : values) {
            // If this is the post record, store it, minus the flag
            if (t.charAt(0) == 'P') {
                post = t.toString().substring(1, t.toString().length())
                        .trim();
            } else {
                // Else, it is a comment record. Add it to the list, minus
                // the flag
                comments.add(t.toString()
                        .substring(1, t.toString().length()).trim());
            }
        }
```

Design Patterns para MR

- STRUCTURED TO HIERARCHICAL
  - Exemplo (reducer):

```
        // If there are no comments, the comments list will simply be empty.

        // If post is not null, combine post with its comments.
        if (post != null) {
            // nest the comments underneath the post element
            String postWithCommentChildren = nestElements(post, comments);

            // write out the XML
            context.write(new Text(postWithCommentChildren),
                    NullWritable.get());
        }
    }
    ...
```

Pós-Graduação em Computação Distribuída e Ubíqua

INF628 - Engenharia de Software para Sistemas Distribuídos
Design Patterns para MapReduce

Sandro S. Andrade
sandroandrade@ifba.edu.br