COMP9313: Big Data Management

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

About the First Assignment

- Problem setting
- Example input and output are given
- Number of reducers: 1
- Make sure that each file can be compiled independently
- Remove all debugging relevant code
- Submission
  - Two java file
  - Two ways
  - Deadline: 01 Apr 2018, 09:59:5

Review of Lab 2

- Package a MapReduce job as a jar via command line
- Eclipse + Hadoop plugin
  - Connect to HDFS and manage files
  - Create MapReduce project
  - Writing MapReduce program
  - Debugging
    - Eclipse d
    - Print debug info to stdout/st p system logs
  - Package a MapReduce job as
  - Check logs of a MapReduce job
- Count the number of words that start with each letter

Letter Count

- Identify the input and output for a given problem:
  - Input: (docid, doc)
  - Output: (letter, count)
- Mapper design:
  - Input: (docid, doc)
  - Output: (lett
  - Map idea: fo in which the key is the starting letter, and the value is
- Reducer design:
  - Input: (letter, (1,1,…,1))
  - Output: (letter, count)
  - Reduce idea: aggregate all the values for the same key “letter”
- Combiner, Reducer and Main are the same as that in WordCount.java

Mapper

```
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 {
               St                                   (value.toString());
               w
                    //convert to low
                    char c = itr.next              ase().charAt(0);
                    //check whether the first letter is a character
                    if(c <= 'z' && c >= 'a'){
                         word.set(String.valueOf(c));
                         context.write(word, one);
                    }
               }
          }
}
```

MapReduce Algorithm Design Patterns

- In-mapper combining, where the functionality of the combiner is moved into the mapper.
- The related patterns “pairs” and “stripes” for keeping track of joint events from a large number of observations.

- “Order inversion vert the sequencing of computations int

- “Value-to-key conversion”, which pr ble solution for secondary sorting.

Cha ce III

Design Patt ey Conversion

Secondary Sort

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered

- What if want to sort value as well?
  - E.g., k → (v_1, r), (v_3, r), (v_4, r), (v_8, r)…

  - Google's Ma des built-in functionality
  - Unfortunatel

- Secondary Sort: sorting values associated with a key in the reduce phase, also called "value-to-key conversion"

Secondary Sort

- Sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis

  $(t1, m1, r_{80521})$

  $(t1, m2, r_{14209})$

  $(t1, m3, r_{76742})$

  …

  $(t2, m1, r_{21823})$

  $(t2, m2, r_{66508})$

  $(t2, m3, r_{98347})$

- We wish to reconstruct the activity at each individual sensor over time
- In a MapReduce program, a mapper may emit the following pair as the intermediate result

  $$m_1 \text{ -> } (t_1, r_{80521})$$

  - We need to sort the value according to the timestamp

Secondary Sort

- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?

- Solution 2:

  - “Value-to-ke rm composite intermediate

    - The mapper emits (m_1, t_1) ->

  - Let execution framework do the
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?
    - Sensor readings are split across multiple keys. Reducers need to know when all readings of a sensor have been processed
    - All pairs associated with the same sensor are shuffled to the same reducer (use partitioner)

How to Implement Secondary Sort

Secondary Sort : Another Example

- Consider the temperature data from a scientific experiment. Columns are year, month, day, and daily temperature, respectively:

```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```


- We want to output the temperature for every year-month with the values sorted in ascending order.

Solutions to the Secondary Sort Problem

- Use the *Value-to-Key Conversion* design pattern:
  - form a composite intermediate key, (K, V), where V is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V) into a reducer key, simply create a composite key
    - K: year-month
    - V : temperature data

- Let the MapReduce execution fram orting (rather than sorting in memory, let the framewor the cluster nodes).

- Preserve state across multiple key-value pairs to handle processing. Write your own partitioner: partition the mapper’s output by the natural key (year-month).

Secondary Sorting Keys

Customize The Composite Key

```
public class DateTemperaturePair
            implements Writable, WritableComparable<DateTemperaturePair> {
                        private Text yearMonth = new Text(); // natural key
                        private IntWritable temperature = new IntWritable(); // secondary key
                        … …
                        @Override
                        /**
                        *                                              f the keys.
                        */
                        public int compareTo(Date                           air) {
                                    int compareVal
this.yearMonth.compareTo(pair.getYearMonth());
                                    if (compareValue == 0) {
                                                compareValue =
temperature.compareTo(pair.getTemperature());
                                    }
                                    return compareValue; // sort ascending
                        }
                        … …
}
```

Customize The Partitioner

```
public class DateTemperaturePartitioner
          extends Partitioner<DateTemperaturePair, Text> {
               @Override
               public int getPartition(DateTemperaturePair pair, Text text, int
numberOfPartitions) {                                          non-negative
                                                               onth().hashCode() %
numberOfPartitions);
               }
}
```


Utilize the natural key only for partitioning

Grouping Comparator

- Controls which keys are grouped together for a single call to Reducer.reduce() function.

```
public class DateTemperatureGroupingComparator extends WritableComparator {
        … …
        protected DateTemperatureGroupingComparator(){
                super(DateTemperaturePair.class, true);
        }
        @Override
        /* This compa                                    get
to the reduce() method */
        public int compare(WritableComparab
                DateTemperaturePair pair = (DateTemperaturePair) wc1;
                DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
                return pair.getYearMonth().compareTo(pair2.getYearMonth());
        }
}
```

Consider the natural key only for grouping

- Configure the grouping comparator using Job object:

```
job.setGroupingComparatorClass(DateTemperatureGroupingComparator.class);
```

MapReduce Algorithm Design

- Aspects that are not under the control of the designer
  - Where a mapper or reducer will run
  - When a mapper or reducer begins or finishes
  - Which input key-value pairs are processed by a specific mapper
  - Which intermediate key-value pairs are processed by a specific reducer

- Aspects that ca

  - Construct data structures as ke

  - Execute user-specified initializa ation code for mappers and reducers (pre-process and post-process)
  - Preserve state across multiple input and intermediate keys in mappers and reducers (in-mapper combining)
  - Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys (order inversion)
  - Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer (partitioner)

MapReduce in Real World: Search Engine

- Information retrieval (IR)
  - Focus on textual information (= text/document retrieval)
  - Other possibilities include image, video, music, …
- Boolean Text retrieval
  - Each document or query is treated as a “bag” of words or terms. Word seque

  - Query terms e Boolean operators AND, OR, and NOT.

    - E.g., ((data AND mining) AN

  - Retrieval
    - Given a Boolean query, the system retrieves every document that makes the query logically true.
    - Called exact match
  - The retrieval results are usually quite poor because term frequency is not considered and results are not ranked

Boolean Text Retrieval: Inverted Index

- The inverted index of a document collection is basically a data structure that
  - attaches each distinctive term with a list of all documents that contains the term.
  - The documents containing a term are sorted in the list

- Thus, in retrieva

  - find the documents that contain .
  - multiple query terms are also e e will see soon.

Boolean Text Retrieval: Inverted Index

Search Using Inverted Index

- Given a query **q**, search has the following steps:
  - Step 1 (vocabulary search): find each term/word in q in the inverted index.
  - Step 2 (results merging): Merge results to find documents that contain all or some of the words/terms in q.

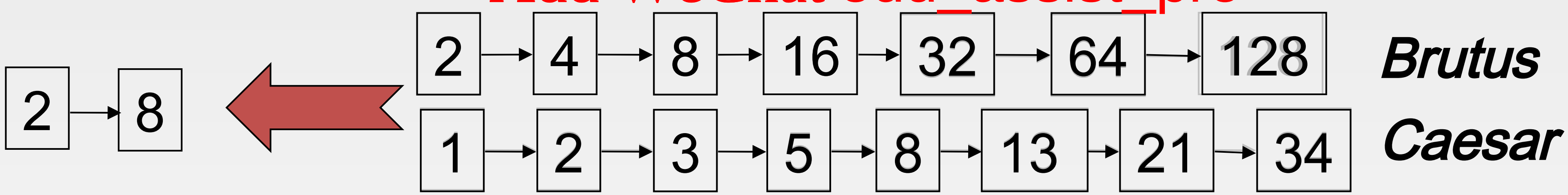
  - Step 3 (Rank score computation): To rank the resulting documents/

    - content-
    - link-based ranking

    - Not used in Boolean retrieva

Boolean Query Processing: AND

- Consider processing the query: ***Brutus*** *AND* ***Caesar***
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate *Caesar* in the Dictionary;
    - Retrieve its postings.
  - “Merge” the
    - Walk thr ... ously, in time linear in the total number of postings



If the list lengths are x and y, the merge takes O(x+y) operations.
Crucial: postings sorted by docID.

MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web,
- The retrieval pro
  - Must have sub-second respons
  - For the web, only need relativel

Perfect for MapReduce!

Uh… not so good…

MapReduce: Index Construction

- Input: documents: (docid, doc), ..
- Output: (term, [docid, docid, …])
  - E.g., (long, [1, 23, 49, 127, …])
    - The docid are sorted !! (used in query phase)
  - docid is an i ue integer. Not an external doc
- How to do it in MapReduce?

MapReduce: Index Construction

- A simple approach:
  - Each Map task is a document parser
    - Input: A stream of documents
      - (1, long ago …), (2, once upon …)
    - Output: A stream of (term, docid) tuples
      - (long, 2) …
  - Reducers c ms of inverted lists
    - Input: (long, [1, 127, 49, 2
    - The reducer sorts the value builds an inverted list
      - Longest inverted list must fit in memory
    - Output: (long, [1, 23, 49, 127, …])
- Problems?
  - Inefficient
  - docids are sorted in reducers

Ranked Text Retrieval

- Order documents by how likely they are to be relevant
  - Estimate relevance(q, d_i)
  - Sort documents by relevance
  - Display sorted results
- User model

  - Present hits lts first

  - At any point, g
- How do we estimate relevance?

  - Assume document is relevant if uery terms
  - Replace relevance(q, d_i) with sim(q, d_i)
  - Compute similarity of vector representations

- Vector space model/cosine similarity, language models, …

Term Weighting

- Term weights consist of two components
  - Local: how important is the term in this document?
  - Global: how important is the term in the collection?

- Here's the intuition:
  - Terms that uld get high weights
  - Terms that uld get low weights

- How do we capture this mathematic
  - TF: Term frequency (local)
  - IDF: Inverse document frequency (global)

TF.IDF Term Weighting

$$w_{i,j} = \mathrm{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

$w_{i,j}$ weight assigned to term *i* in document *j*

$\mathrm{tf}_{i,j}$ *i* in document *j*

N number of docum llection

n_i number of documents with term *i*

Retrieval in a Nutshell

- Look up postings lists corresponding to query terms
- Traverse postings for each query term
- Store partial query-document scores in accumulators
- Select top *k* res

MapReduce: Index Construction

- Input: documents: (docid, doc), ..
- Output: (t, [(docid, w_t), (docid, w), …])
  - w_t represents the term weight of t in docid
  - E.g., (long, [(1, 0.5), (23, 0.2), (49, 0.3), (127,0.4), …])
    - The doci ase)
- How this problem differs from the pr
  - TF computing
    - Easy. Can be done within the mapper
  - IDF computing
    - Known only after all documents containing a term t processed
  - Input and output of map and reduce?

Inverted Index: TF-IDF

MapReduce: Index Construction

- A simple approach:
  - Each Map task is a document parser
    - Input: A stream of documents
      - (1, long ago …), (2, once upon …)
    - Output: A stream of (term, [docid, tf]) tuples

      - (long,]) (upon, [2,1]) …

  - Reducers c ms of inverted lists
    - Input: (long, {[1,1], [127,2 …})

    - The reducer sorts the value builds an inverted list
      - Compute TF and IDF in reducer!
    - Output: (long, [(1, 0.5), (23, 0.2), (49, 0.3), (127,0.4), …])

MapReduce: Index Construction

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

MapReduce: Index Construction

- Inefficient: terms as keys, postings as values
  - docids are sorted in reducers
  - IDF can be computed only after all relevant documents received
  - Reducers must buffer all postings associated with key (to sort)
    - What if we run out of memory to buffer postings?
  - Improvemen

The First Improvement

- How to make Hadoop sort the docid, instead of doing it in reducers?
- Design pattern: value-to-key conversion, secondary sort
- Mapper output a stream of ([term, docid], tf) tuples

- Remember: you must implement a partitioner on term!

The Second Improvement

- How to avoid buffering all postings associated with key?

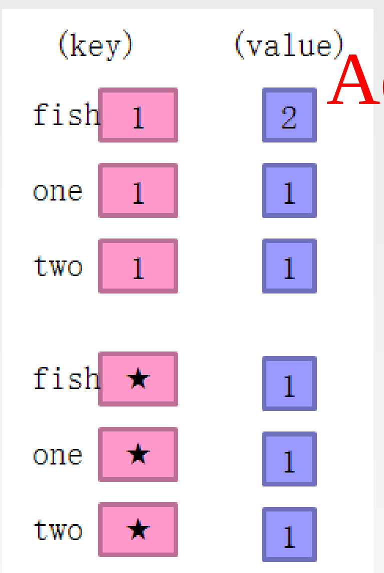
We'd like to store the DF at the front of the postings list

now the DF until we've
gs!

Sound familiar?
Design patter: Order inversion

The Second Improvement

- Getting the DF
  - In the mapper:
    - Emit “special” key-value pairs to keep track of DF
  - In the reducer:
    - Make sure “special” key-value pairs come first: process them to deter
  - Remember:



Emit normal key-value pairs…

Emit “special” key-value pairs to keep track of df…

Doc1: one fish, two fish

The Second Improvement

First, compute the DF by summing contributions from all "special" key-value pair…

Write the DF…

l ly define sort order to make
s -value pairs come first!

Retrieval with MapReduce?

- MapReduce is fundamentally batch-oriented
  - Optimized for throughput, not latency
  - Startup of mappers and reducers is expensive

- MapReduce is not suitable for real-time queries!
  - Use separat

- Real world search engines much m d sophisticated

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

MapReduce Counters

- Instrument Job's metrics
  - Gather statistics
    - Quality control – confirm what was expected.
      - E.g., count invalid records
    - Application level statistics.
  - Problem dia
  - Try to use c instead of log files
- Framework provides a set of built-in
  - For example bytes processed f tput
- User can create new counters
  - Number of records consumed
  - Number of errors or warnings

Built-in Counters

- Hadoop maintains some built-in counters for every job.
- Several groups for built-in counters
  - File System Counters – number of bytes read and written
  - Job Counters – documents number of map and reduce tasks launched, number of failed tasks
  - Map-Reduc cer, combiner input and output recor tistics

User-Defined Counters

- You can create your own counters
  - Counters are defined by a Java enum
    - serves to group related counters
    - E.g.,

```
enum Temperature {

}
```


- Increment counters in Reducer and sses
  - Counters are global: Framework accurately sums up counts across all maps and reduces to produce a grand total at the end of the job

Implement User-Defined Counters

- Retrieve Counter from Context object
  - Framework injects Context object into map and reduce methods

- Increment Counter's value
  - Can increment by 1 or more

Implement User-Defined Counters

- Get Counters from a finished job in Java
  - Counter counters = job.getCounters()

- Get the counter according to name
  - Counter c1 = counters.findCounter(Temperature.MISSING)

- Enumerate all c


```
for (CounterGroup group : counters) {
        System.out.println("* Counter Group:            layName() + " (" +
        group.getName() + ")");
        System.out.println("  number of counters in this group: " + group.size());
        for (Counter counter : group) {
                System.out.println("  - " + counter.getDisplayName() + ": " +
                counter.getName() + ": "+counter.getValue());
        }
}
```

MapReduce SequenceFile

- File operations based on binary format rather than text format
- SequenceFile class prvoides a persistent data structure for binary key-value pairs, e.g.,
  - Key: timestamp represented by a LongWritable
  - Value: quan y a Writable
- Use SequenceFile in MapReduce:
  - job.setinputFormatClass(Seque Format.class);
  - job.setOutputFormatClass(SequenceFileOutputFormat.class);
  - In Mapreduce by default *TextInputFormat*

MapReduce Input Formats

- InputSplit
  - A **chunk** of the input processed by a single map
  - Each split is divided into records
  - Split is just a reference to the data (doesn't contain the input data)

- RecordReader
  - Iterate over records
  - Used by the map task to generate record key-value pairs

- As a MapReduce application programmer, we do not need to deal with InputSplit directly, as they are created in InputFormat

- In MapReduce, by default TextInputFormat and LineRecordReader

MapReduce InputFormat

MapReduce OutputFormat

Detailed Hadoop MapReduce Data Flow

Creating Inverted Index

- Given you a large text file containing the contents of huge amount of webpages, in which each webpage starts with "<DOC>" and ends with "</DOC>", your task is to create an inverted index for these documents.
  - A sample file

- Procedure:
  - Implement a
  - Implement a custom InputForm to the CreateRecordReader() function self-defined RecordReader object
  - Configure the InputFormat class in the main function using job.setInputFormatClass()

- Try to finish this task using the sample file

Methods to Write MapReduce Jobs

- Typical – usually written in Java
  - MapReduce 2.0 API
  - MapReduce 1.0 API
- Streaming
  - Uses stdin and stdout
  - Can use an duce Functions
    - C#, Pyth
- Pipes
  - Often used with C++
- Abstraction libraries
  - Hive, Pig, etc… write in a higher level language, generate one or more MapReduce jobs

Number of Maps and Reduces

- Maps
  - The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.
  - The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks.
  - If you expect ksize of 128MB, you'll end up with 82,00 RJobConfig.NUM_MAPS, int) (which onl is used to set it even higher.

- Reduces
  - The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> * <*no. of maximum containers per node*>)
  - With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
  - Use job.setNumReduceTasks(int) to set the number

MapReduce Advantages

- Automatic Parallelization:
  - Depending on the size of RAW INPUT DATA ➔ instantiate multiple MAP tasks
  - Similarly, depending upon the number of intermediate <key, value> partitions ➔ instantiate multiple REDUCE tasks
- Run-time:
  - Data partitio
  - Task scheduling
  - Handling machine failures
  - Managing inter-machine communication
- Completely transparent to the programmer/analyst/user

The Need

- Special-purpose programs to process large amounts of data: crawled documents, Web Query Logs, etc.
- At Google and others (Yahoo!, Facebook):
  - Inverted index
  - Graph structure of the WEB documents
  - Summaries queries, etc.
  - Ad Optimiza
  - Spam filtering

Map Reduce vs Parallel DBMS

Pavlo et al., SIGMOD 2009, Stonebraker et al., CACM 2010, …

Practice : Design MapReduce Algorithms

- Counting total enrollments of two specified courses

- Input Files: A list of students with their enrolled courses

  Jamie: COMP9313, COMP9318

  Tom: COMP9331, COMP9313

  … …

- Mapper selects records and outputs
  - Input: Key – student, value – a l
  - Output: (COMP9313, 1), (COMP9318, 1), …

- Reducer accumulates counts
  - Input: (COMP9313, [1, 1, …]), (COMP9318, [1, 1, …])
  - Output: (COMP9313, 16), (COMP9318, 35)

Practice : Design MapReduce Algorithms

- Remove duplicate records
- Input: a list of records

  2013-11-01 aa
  2013-11-02 bb
  2013-11-03 cc
  2013-11-01 aa
  2013-11-0

- Mapper
  - Input (record_id, record)
  - Output (record, “”)
    - E.g., (2013-11-01 aa, “”), (2013-11-02 bb, “”), …
- Reducer
  - Input (record, [“”, “”, “”, …])
    - E.g., (2013-11-01 aa, [“”, “”]), (2013-11-02 bb, [“”]), …
  - Output (record, “”)

Practice : Design MapReduce Algorithms

- Assume that in an online shopping system, a huge log file stores the information of each transaction. Each line of the log is in format of "userID**\t** product**\t** price**\t** time". Your task is to use MapReduce to find out the top-5 expensive products purchased by each user in 2016

- Mapper:

  - Input(transa

  - initialize an ty queue Q of log record based on price)

  - map(): get local top-5 for each
  - cleanup(): emit the entries in H

- Reducer:
  - Input(userID, list of queues[])
  - get top-5 products from the list of queues

Practice : Design MapReduce Algorithms

- Reverse graph edge directions & output in node order

- Input: adjacency list of graph (3 nodes and 4 edges)

  (3, [1, 2]) (1, [3])

  (1, [2, 3]) ➔ (2, [1, 3])

- Note, the node_ids in the output val orted. But Hadoop only sorts on keys!

- Solutions: Secondary sort

Practice : Design MapReduce Algorithms

- Map
  - Input: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse direction)
  - Output: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1]).
    - Copy node_ids from value to key.

- Partition on Key both fields)
  - Input: (<1, , [1]), (<3, 1>, [1])
  - Output: (<1, 3>, [3]), (<2, 1>, [ ]), (<3, 1>, [1])

- Grouping comparator
  - Merge according to part of the key
  - Output: (<1, 3>, [3]), (<2, 1>, [1, 3]), (<3, 1>, [1])
    this will be the reducer's input
- Reducer
  - Merge according to part of the key
  - Output: (1, [3]), (2, [1, 3]), (3, [1])

Practice : Design MapReduce Algorithms

- Calculate the common friends for each pair of users in Facebook. Assume the friends are stored in format of Person->[List of Friends], e.g.: A -> [B C D], B -> [A C D E], C -> [A B D E], D -> [A B C E], E -> [B C D]. Your result should be like:
  - (A B) -> (C D)
  - (A C) -> (B D)
  - (A D) -> (B
  - (B C) -> (A
  - (B D) -> (A C E)
  - (B E) -> (C D)
  - (C D) -> (A B E)
  - (C E) -> (B D)
  - (D E) -> (B C)

Practice ： Design MapReduce Algorithms

- Mapper:
  - Input(user u, List of Friends [f_1, f_2, …,])
  - map(): for each friend f_i, emit (<u, f_i>, List of Friends [f_1, f_2, …,])

- Reducer:
  - Input(user u
  - Get the inter

- Example: http://stevekrenzel.com/articles/finding-friends

References

- Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.
- Hadoop The Definitive Guide. Hadoop I/O, and MapReduce Features chapters.