Standard front page

IT University of Copenhagen

Course exam with submission

Class code:
SAD2

Name of course:
Algorithm Design II

Course manager:
Rasmus Pagh

Course e-portfolio:

Thesis/project exam with project agreement

Supervisor:

Thesis or project title:

Name(s):	Birthdate and year:	ITU-mail:
1. Marcus Gregersen	17/02-1987	mabg@
2. Martin Faartoft	19/01-1983	mlfa@
3. Rick Marker	04/12-1988	rdam@
4.		@
5.		@
6.		@
7.		@

Finding Hollywood's most Popular Using Map-Reduce and Approximation

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Rick Marker
rdam@itu.dk

16. December 2013

1 Introduction

In this report, we try to answer the question: "How many unique co-actors has a given actor starred alongside", we call this the *Popular* problem. We will describe three methods that solves the problem: One Naïve implementation, One using the Map-Reduce framework provided by Hadoop, and another using an approximation algorithm. All three methods will be tested on the supplied IMDB Dataset[1].

To compare the speed of our Map-Reduce implementation and the accuracy of our approximation algorithm, a naïve sequential implementation has been created to compare with. For learning purposes we have projected our data from the dataset, to be of the form List<Movie, List<Actor», as this forces us to use two rounds in our Map-Reduce algorithm. The sequential algorithm receives the same input to be able to make a fair comparison. To further compare, our Map-Reduce algorithm will be tested on both a local setting and on Amazon Elastic MapReduce service[2].

The code for the three methods introduced in this report can be found in the Appendix or at `https://github.com/bagvendt/SAD2_Project`

2 Naïve Sequential Algorithm

In the sequential algorithm we build an inverted index, that maps movies to the actors that star in them, using a hash table. This allows us to check which actors appear in which film in constant time. Building the index takes $\mathcal{O}(a \cdot m)$ time, where a is the number of actors and m is the number of movies

When the inverted index is built, we iterate over all actors in the dataset, and for each actor we iterate over the movies they appear in, and for each of those, we note their co-actors. For each co-actor a given actor has starred with, we add it to that actors total. Even though all lookups are implemented to be done in constant time, having 3 nested loops, we get a running time of $\mathcal{O}\big(a^2 \cdot m\big)$. Finally we write the output, which takes another $\mathcal{O}(a \cdot m)$ time.

This gives our sequential algorithm a worst-case running time of $\mathcal{O}\big(a^2 \cdot m\big)$. When running the algorithm on our dataset, the actual running time is much better than the theoretical worst-case. We see this is intuitively true as the actor/movie relations can also be seen as a spare boolean matrix, which means we will only perform work on a small part of our matrix. This, in turn, yields a much lower running time for this dataset than in the worst

[1] IMDB is a site containing information about movies and actors, available at `www.imdb.com`

[2] `https://aws.amazon.com/elasticmapreduce/`

case.

3 Map-Reduce Algorithm

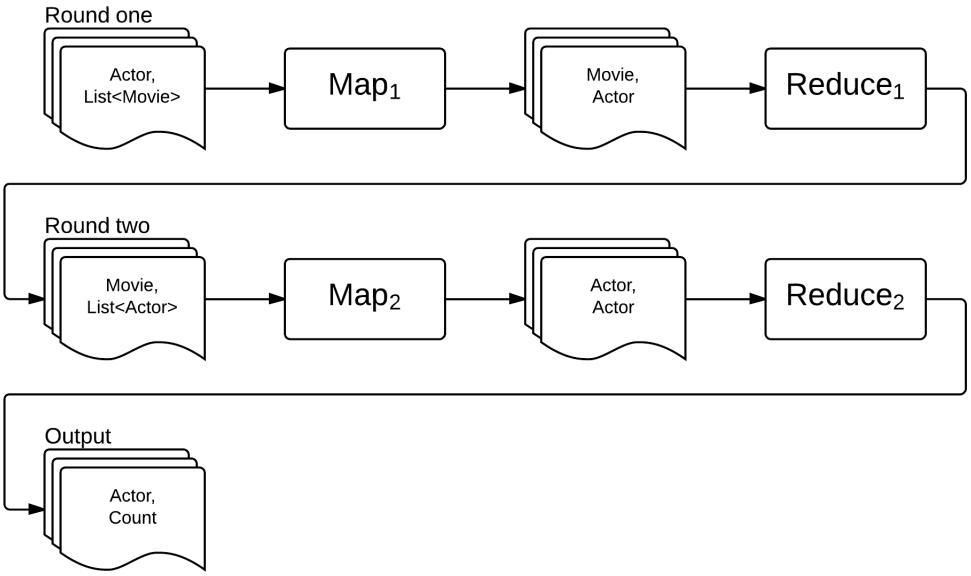

Figure 1: Map-Reduce overview

Solving the *Popular* problem using Map-Reduce with the specified input format, requires two rounds. The first round will pivot the input into a more suitable structure, while the second round will answer the actual question, that is count the number of unique pairings of actors.

3.1 Round One

Goal: Transform input pairs (Actor, List<Movie>) into output pairs (Movie, List<Actor>).

The mappers break down the (Actor, List<Movie>) pairs into a series of (Movie, Actor) pairs, one for each Movie in the input List<Movie>, essentially emitting the information: "this actor played in this movie".

The shuffle phase (handled by the Hadoop framework), directs each pair emitted by a mapper, to a reducer, such that every pair with a given key k, is sent as input to a single reducer R_k.

The reducers each receive a list of (Movie, Actor) pairs, and builds from those a single pair (Movie, List<Actor>) by simply appending each actor to

a list. The output of round one is thus, a pivot of the input data changing the shape from (Actor, List<Movie>) to (Movie, List<Actor>), essentially saying "this movie contained exactly these actors".

3.2 Round Two

Goal: Transform input pairs (Movie, List<Actor>) into output pairs (Actor, Count), where count is the number of unique co-actors this actor has starred with in a movie.

The mappers break down the (Movie, List<Actor>) pairs into all possible Actor-pairs along with their inverses.

```
function RoundTwoMap(...)
    for(actor a in actors)
        for(actor b in actors)
            if(a != b)
                emit(a, b)
```

This produces a number of pairs, quadratic in the number of actors in the movie, each capturing the information "Actor A played with Actor B in some movie".

The reducers each receive a list of ($Actor_{key}$, $Actor_{value}$) pairs, and using a Java HashSet[3], they filter out any duplicate pairs, and emit the final (Actor, count) pairs, one for each actor. These pairs give the number of unique co-actors, a given actor has had during his career.

3.3 Analysis

To gauge the efficiency of our implementation, we look at three important performance measures from a Map-Reduce point of view. The number of rounds in the computation, the amount of work done in each mapper/reducer (in the worst case), and the number of pairs emitted.

Number of rounds The algorithm runs in a constant number of rounds (2), independent of the problem size. As mentioned previously, this can be reduced to a single round, if the input has a more suitable structure.

Work distribution (skewness) The amount of work done in each mapper/reducer is close to being balanced. Obviously, the round two reducer assigned to the most productive actor in the dataset, will do more work than the reducer assigned to an actor that only star in a single movie, but in terms

[3]The 'Set' interface guarantees that no two equal objects can be contained in the set, at the same time.

of the input size, these numbers are very close to each other. This is only true because no actor has played in a significant fraction of the total number of movies in the dataset[4].

Number of pairs The number of pairs generated in round two, is proportional to the number of unique pairs of actors that starred in a movie together. For every movie that has A actors starring in it, $2 \cdot A(A-1)$ pairs will be emitted.

This is a large amount of pairs, but we cannot see how this number can be reduced, without relaxing the constraint that a pair of actors should be counted only once. If that constraint is lifted, the round two mapper can simply emit (Actor, Movies.Actors.count - 1), meaning that "this actor starred with n other actors in some movie". The reducers could then simply sum all those counts, to get the total. In that case the best guarantee we can give for our approximation for each actor, $\tilde{z}_i$, is $\tilde{z}_i \leq z_i|m_i|$, where m_i is that actors total number of movies and i is the actors index. This is the case when an actor has only played with the same group of people.

The relaxed version of round two produces drastically fewer pairs (one pair for each actor, for each movie they starred in), compared to the exact version (two pairs for each unique pair of actors per movie). By making the mapper slightly smarter, the shuffler and reducers do much less work. But it also solves a slightly different problem, than the one we are initially interested in. Overall the performance on the given dataset is acceptable, but for significantly larger datasets, the number of pairs emitted, might cause a noticeable slowdown.

3.4 Verification of results

We have designed the format of the output of our sequential algorithm, described in Section 2, in such a way that it conforms to the format of the Map-Reduce algorithm described in Section 3. The output is thus on the form:

```
[
  (Actor_ID_1, Count_1),
  (Actor_ID_2, Count_2),
  ...
  (Actor_ID_n, Count_n)
]
```

Since we have no guarantee that the row `Actor_ID` is sorted in the output of the Map-Reduce algorithm, we have implemented a simple Python script

[4]If this was not true, additional techniques, beyond the scope of this project, would be needed to balance the workload.

`sort_and_expand.py`. The role of this script is firstly to sort the input by `Actor_ID` and secondly to add the first and last name of the actor in question, for more readable output.

Having three different outputs produced by a: the sequential algorithm, b: the Map-Reduce algorithm run on a local machine, and c: the Map-Reduce algorithm run on the Amazon Elastic MapReduce service, we then transform these outputs using the script. Using the UNIX tool `diff`, whose job it is to output the difference between two files, we see that that all three outputs are exactly the same.

The fact that the two algorithms, developed by using different programming languages and different algorithmic paradigms, produce the same output, gives us a very high confidence in the correctness of both implementations.

The first 10 lines of the output, i.e the top 10 actors that has starred with most unique actors are:

```
(621468, 'Bess Flowers', 10834)
(372839, 'Lee Phelps', 6679)
(74450, 'John Carradine', 6447)
(212581, 'Stuart Holmes', 6318)
(152868, 'James Flavin', 6027)
(22585, 'Irving Bacon', 5957)
(233082, 'James Earl Jones', 5894)
(245158, 'Donald (I) Kerr', 5775)
(209799, 'Adolf Hitler', 5773)
(433904, 'Martin Sheen', 5764)
```

Taking a sample from the output we see from Bess Flowers' Wikipedia that she was a Hollywood extra that has appeared in over 700 movies. It seems safe to assume that since she has starred in so many movies she would also be starring with a lot of other actors

We realize that this is not a formal proof that the algorithm produces the correct output.

3.5 Benchmark

We have carried out a small scale benchmark experiment to test the difference in running time between the sequential and the Map-Reduce algorithms, the latter in both a local and a distributed setup.

For the local setup we used a 2.5 GHz Intel Core i5 chip (denoted as *Local** in the table).

In the distributed setup we used the online web service Amazon Elastic Mapreduce that runs a customized version of Hadoop.

The results of the experiment is seen in Table 1

Type	Local / Distributed	Instances	Instance type	Time (sec)
Sequential	Local	1	*Local**	66
Map-Reduce	Local	1	*Local**	244
Map-Reduce	Distributed	2	m1.small	840
Map-Reduce	Distributed	10	m1.small	600
Map-Reduce	Distributed	2	m1.xlarge	180
Map-Reduce	Distributed	10	m1.xlarge	120

Table 1: The results of our experiment. Time is the time used, from computation start to end. In the distributed setup the time values are multiples of 60, since the Amazon console only outputs execution time in minutes.

Our experiment does not contain enough data to make any sweeping conclusions, but it does illustrate a tendency in the distributed setup that more powerful instances yield a faster running time and more instances yield a faster running time.

The experiment also shows that the sequential algorithm yields the fastest running time, we have been able to achieve. It is our strong belief that if the dataset was larger we would see the distributed setup execute significantly faster than the sequential. It is our belief that the reason for the aquired result is the overhead associated to distributed systems and Map-Reduce specifically.

4 Approximating the "Popular" problem

Computing the exact solution, using our Map-Reduce approach, may not scale well with the size of the input, due to the number of pairs emitted. In the following, we suggest an algorithm for approximating the solution, building on previous work by Amossen et al. in [1].

4.1 Equivalence of problems

The Popular problem can be expressed as a boolean matrix multiplication. Consider a matrix m_1 of size $|actors| \cdot |movies|$, where the value 1 at position (x, y) means that actor x starred in movie y. The matrix multiplication: $M = m_1 \cdot m_2$, where $m_2 = m_1^T$, will then have the dimensions $|actors| \cdot |actors|$, and a 1 at position (x, y) indicates that actor x and y starred in at least one movie together. Summing the number of non-zero entries in $M_{i,j}$ for a given i, gives the number of unique actors that actor i starred with at least once, which is exactly the value we want to compute.
Additionally, matrix multiplication can be expressed as the *join-distinct* operation from relational algebra as follows: Consider the boolean matrix multiplication: $M = m_1 \cdot m_2$. Let the relation R_1 be the pairs corresponding to

the non-zero values in m_1, and let R_2 be the pairs corresponding to non-zero values in m_2. Now each unique pair in $R_1 \bowtie R_2$ will correspond to a non-zero value in M.

4.1.1 Related Work

Our work proposes a variant to the algorithm presented in [1]. They use an approximation technique presented by Bar-Yossef in [2], along with a clever sorting trick, to estimate the number of non-zero entries in boolean matrix products in expected linear time.

[2] presents a technique for estimating the number of distinct items in a stream, using $\mathcal{O}(1)$ space. If every item in the stream is hashed, and the smallest hash value, p, is stored. The number of distinct items in the stream, $\tilde{z}$, can be approximated by $\tilde{z} = 1/p$. The variance of this can be reduced by storing the k smallest hash values, and use $\tilde{z} = k/p$ as approximation.

4.2 Our Algorithm

Instead of approximating the total number of distinct values in the matrix multiplication $\pi_{ac}(R_1 \bowtie R_2)$, with $R_1(a, b)$, $R_2(b, c)$, as done by Amossen et. al., we estimate the number of distinct values of $\pi_{ac}(\sigma_{a=i}(R_1 \bowtie R_2))$ for $i \in W$ where $W \subseteq \pi_a(R_1)$ and assume that $|W| \ll |\pi_a(R_1)|$. W is the list of actors we are interested in following. We call W our 'watchlist'.
At a high level, our algorithm performs the following steps: It iterates over all values in $\mathcal{B}$, which is the set of values that has been used to join. For each value $i \in \mathcal{B}$, we find all the pairs that would be joined for that particular i, and with some probability, we look at that pair. This is done to reduce our work from polynomial time to linear time. Afterwards we use the approximation technique from [2], to compute estimates.

The algorithm uses the following functions and variables:

- $\mathcal{B}$: $\pi_b(R_1 \cup R_2)$
- $\mathcal{W}$: Chosen subset of $\pi_a(R_1)$ (the watchlist)
- $\mathcal{A}_i$: $\pi_a(\sigma_{b=i}(R_1))$
- $\mathcal{C}_i$: $\pi_c(\sigma_{b=i}(R_2))$
- F_s: Buffer containing the last $[0 : k]$ observed hash values for a given $s \in \mathcal{W}$
- S_s: Sketch containing the k smallest observed hash values for a given $s \in \mathcal{W}$

- p_s: k Probability that a new pair will be added to the sketch F_s
- p_{max}: The maximum value of $P_s, s \in \mathcal{W}$
- k: Size of the sketches
- Hash function: $h_1 : \mathbb{R} \to [0, 1]$
- Hash function: $h_2 : \mathbb{R} \to [0, 1]$
- Hash function: $h : [0, 1] \times [0, 1] \to [0, 1]$

Where h_1, h_2 and h are pairwise independent. And h is defined as:

$$h(x, y) = (h_1(x) - h_2(x)) \bmod 1$$

Approach Intuitively[5] we create a table for each value $i \in \mathcal{B}$. These tables have the dimensions $|\mathcal{A}_i \cap \mathcal{W}| \cdot | \times |\mathcal{C}_i|$, where the rows are sorted according to h_1 from top to bottom and the columns are sorted according to h_2 from left to right. Each entry in the table contains a computed value of h.

The algorithm starts at the top left of the table (row s, column t), iterating over the rows of the left-most column to find a cell where the h-value is smaller than the one above it. We know that since the row is sorted in h_1, that this cell is guaranteed to have the smallest value of h in the column. From this cell, we continue looping through the column, now comparing each value to p_{max}, to check if one of the sketches could possibly include the pair being considered. (We remark that this is different from [1], where only a single p is maintained and checked against).

After the check against p_{max}, the value of h is checked against p_s. If the value of h is lower, we add the pair (x_s, y_t) to the buffer F_s. When the current value of h is greater than p_{max} we move on to the next column, starting from the row where we found the minimum value in the previous column. This is more efficient, because it utilizes the fact that the columns are sorted according to h_2, limiting the number of iterations needed. When a buffer is filled, it is merged into the corresponding sketch, using the `COMBINE` procedure. `COMBINE` works by finding the median and then move smaller values into the sketch, in linear time[6]. Once all $i \in \mathcal{B}$ have been processed, we approximate $\tilde{z}_s$ by utilizing the result from [2] giving us: $\tilde{z}_s = k/p_s$

[5] Only the cells evaluated will have their h value computed

[6] The procedure is outline in greater detail in [1]

4.3 Pseudocode

Algorithm 1: Pseudocode for DisItemsPerRow

```
begin
    p_max ← p
    k ← ⌈9/ε²⌉
    F ← ∅
    for i ∈ B do
        A_π ← A_i ∩ W
        x ← A_π sorted according to h_1-value
        y ← C_i sorted according to h_2-value
        s̄ ← 1
        for t := 1 to |C_i| do
            while h(x_s̄, y_t) > h(x_{(s̄−1) mod |A_π|}, y_t) do
                s̄ ← (s̄ + 1) mod |A_π|
            end
            s ← s̄
            while h(x_s, y_t) < p_max do
                if h(x_s, y_t) < p_s then
                    F_s ← F_s ∪ {(x_s, y_t)}
                    if |F_s| = k then
                        (p_s, S_s) ← COMBINE(S_s, F_s)
                        p_max ← max(p)
                        F_s ← ∅
                    end
                end
                s ← (s + 1) mod |A_π|
            end
        end
    end
    R ← ∅
    for u ∈ W do
        (p_u, S_u) ← COMBINE(S_u, F_u)
        if |S_u| = k then
            R ← R ∪ {(k/p_u, u)}
        end
        else
            R ← R ∪ {(0, u)}
        end
    end
    return R
end
```

4.4 Analysis

Running Time We split the analysis of the running time into two. The first part considers the *while* loop on lines 15 to 24, and the second part considers the remaining work done.

while loop. Every iteration will add at most one pair (x_s, y_t) to one of the buffers, F_i. We can only safely exit the loop if $h(x_s, y_t) < p_{max}$, but a candidate pair will not be added unless $h(x_s, y_t) < p_s$, effectively wasting a number of iterations, proportional to $p_{max} - p_i$. Thus the expected number of iterations are $\mathcal{O}(p_{max}|\mathcal{W}||\mathcal{C}_i|)$.
Every call to `COMBINE` takes $\mathcal{O}(k)$ time, [1]. Since these calls happen, at most, every k iterations of the inner loop, this gives amortized linear running time for `COMBINE`.

Remaining work. The intersection on line 5, can be done in $O(|\mathcal{W}| + |\mathcal{C}_i|)$ time, using a hash table. Since $|W| < |C_i|$, this is equal to $O(|C_i|)$. Updating P_{max} on line 20 can be done in $log(|\mathcal{W}|)$ time, using a Heap.

The overall time complexity is dominated by the execution of the inner loop, $\mathcal{O}(n + \sum_i p_{max}|\mathcal{W}||\mathcal{W}_i|)$.

The worst-case running time is the same as [1] (assuming $p_{max} = 1$), but in actual use, we predict that our implementation will be significantly slower. This is due to the wasted work done in the inner loop, whenever $P_i < P_{max}$, and the fact that p_{max} will decrease at a lower rate than p in [1].

Space Usage The algorithm uses $\mathcal{O}(k \cdot |W|)$ space, since it has to maintain $\mathcal{W}$ individual sketches, each of size $\mathcal{O}(k)$. It is worth noting, that the space usage is independent of the input size n. This is a factor $|\mathcal{W}|$ more than [1], which is to be expected.

Error and Variance As we use the same approximation technique as the one presented in [2]. the same analysis applies to our algorithm. This means that for each exact solution, z_i, and the corresponding approximation, $\tilde{z}_i$ we know that $\tilde{z}_i \in [z_i - \epsilon, z_i + \epsilon]$, where $\epsilon = \sqrt{9/k}$. We can state this is the case with probability $2/3$.

If we wish to increase the confidence in the approximation, the error-probability $(2/3)$ can be made arbitrarily small, by rerunning the algorithm a number of times, and choosing the median.

4.5 Experiments

We have set up an experiment in which we run our algorithm 100 times on the data from the "Popular" problem. On each repetition, we generate new hash functions h_1 and h_2. Furthermore we have selected W to be the top 10 actors introduced in Section 3.4.

We start each repetition with p=1. This is done to ensure that we fill the sketches. We have run the experiment twice, 100 for $k = 256$ and 100 for $k = 1024$, to see the effect of choice of k on the variance.

In every iteration and for each actor we calculate a ratio, "observed estimate" / "actual value". By sorting these values, we can draw the ratio / cumulative distribution function graph, for each choice of k. These graphs are presented in Figures 2 and 3.

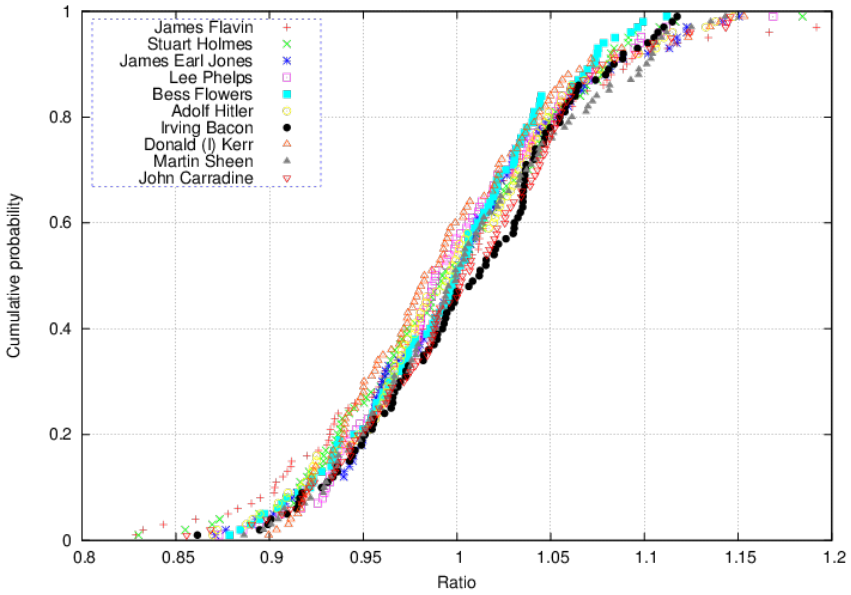

Figure 2: k=256

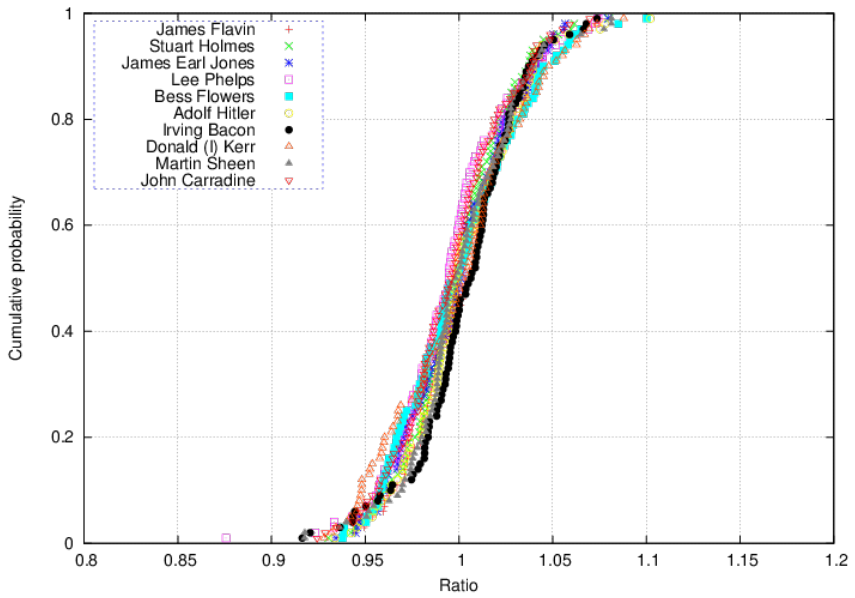

Figure 3: k=1024

We have compared the observed and expected ϵ for both k values in Table 2. A choice of $k = 1024$ corresponds to $\epsilon = \sqrt{9/1024} = 0.0936$. The $\epsilon - k$ relationship makes intuitive sense. When you increase k, ϵ decreases, and you get a better approximation. However, it is also important to note, that if k is chosen to be too large, the sketch S will never be filled, and no meaningful approximation can be given.

k	Expected ϵ	Observed ϵ
256	0.1875	0.054
1024	0.0936	0.039

Table 2: Observed and expected ϵ values, as a function of k.

4.6 Suggested Improvements

If $|W| \ll n$ and the matrix is sparse, $|A_\pi| = 1$ will occur often, Corresponding to iterating over a movie starring exactly one of the actors on the watchlist. In this case, the inner loop is poorly optimized. Firstly p_i could be used, instead of p_{max}, and secondly it could be exploited that the values of $h(x_s, y_t)$ becomes smaller mod 1, when t increases. Using a trick analogous to lines 11 to 13 in the pseudocode, could lower the bound on the inner loop from $\mathcal{O}(p_{max}|\mathcal{A}_i||\mathcal{C}_i|)$ to $\mathcal{O}(p|\mathcal{C}_i|)$.

We observe that the size estimation algorithm could be parallelized by spawning a thread for each movie $i \in \mathcal{B}$. Each time some S and F is to

be updated, either by adding to a pair to F or by calling the procedure COMBINE on them, a mutex lock could be introduced to enforce atomicity.

5 Conclusion

In this report we have presented three methods to find the most popular actors in the IMDB dataset.

Firstly we presented a simple Naïve solution that just iterates over all possible actor-actor combinations. The purpose of presenting this solution was merely to use its results as the ground truth for the following two methods.

Secondly we have introduced a solution to the problem in a distributed environment, using techniques that conform to the Map-Reduce framework as it is implemented in Apache Hadoop.

Lastly we have presented an algorithm for approximating the size of boolean matrix products per row. The algorithm runs in linear time, if p_{max} is chosen correctly. The approximations lie within a factor $1 \pm \epsilon$ of the actual value, with a probability of 2/3.

References

[1] Rasmus Resen Amossen, Andrea Campagna, Rasmus Pagh Better Size Estimation for Sparse Matrix Products. Algorithmica, March 2013.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. Springer-Verlag, 2002. (RANDOM '02)

Appendix

Popular.java - Map-Reduce

```
1  package dk.itu.sad2;
2
3  import org.apache.hadoop.conf.Configuration;
4  import org.apache.hadoop.fs.Path;
5  import org.apache.hadoop.io.IntWritable;
6  import org.apache.hadoop.io.LongWritable;
7  import org.apache.hadoop.io.Text;
8  import org.apache.hadoop.mapred.jobcontrol.JobControl;
9  import org.apache.hadoop.mapreduce.Job;
10 //import org.apache.hadoop.mapred.jobcontrol.*;
11 import org.apache.hadoop.mapreduce.Mapper;
12 import org.apache.hadoop.mapreduce.Reducer;
13 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
14 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
15 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
16 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
17
18 import java.io.IOException;
19 import java.util.ArrayList;
20 import java.util.HashSet;
21 import java.util.StringTokenizer;
22
23 /**
24  */
25 public class Popular {
26
27     public static class RoundOneMap extends Mapper<LongWritable,
       Text, Text, Text> {
28         // Input <Actor, List<Movies>>
29         // Returns emits <Movie, Actor>
30         private Text actor = new Text();
31         private Text movie = new Text();
32
33         public void map(LongWritable key, Text value, Context
       context) throws IOException, InterruptedException {
34             String line = value.toString();
35             StringTokenizer tokenizer = new StringTokenizer(line
       );
36             actor.set(tokenizer.nextToken());
```

```
37              while (tokenizer.hasMoreTokens()) {
                    movie.set(tokenizer.nextToken());
39                  context.write(movie, actor);
                }
41          }
        }
43
        public static class RoundOneReduce extends Reducer<Text,
        Text, Text, Text> {
45          // Input List<Movie, Actor>
            // Emits <Movie, List<Actor>
47          public void reduce(Text movie, Iterable<Text> actors,
        Context context)
                    throws IOException, InterruptedException {
49              StringBuilder result = new StringBuilder();
                for(Text actor : actors) {
51                  result.append(actor.toString() + " ");
                }
53              context.write(movie, new Text(result.toString()));
            }
55      }

57      public static class RoundTwoMap extends Mapper<LongWritable,
         Text, Text, Text> {
            // Input <Movie, List<Actor>
59          // Emits <Actor,Actor>

61          public void map(LongWritable key, Text value, Context
        context) throws IOException, InterruptedException {
                String line = value.toString();
63              StringTokenizer tokenizer = new StringTokenizer(line
        );

65              ArrayList<String> actors = new ArrayList<String>();
                tokenizer.nextToken();
67              while (tokenizer.hasMoreTokens()) {
                    actors.add(tokenizer.nextToken());
69              }

71              for(int i = 0; i < actors.size(); i++){
                    for(int j = 0; j < actors.size(); j++){
73                      String first = actors.get(i);
                        String second = actors.get(j);
75                      if (!first.equals(second))
                            context.write(new Text(first), new Text(
        second));
77                  }
                }
79          }
        }
81
        public static class RoundTwoReduce extends Reducer<Text,
        Text, Text, IntWritable> {
83          // Input <Actor, Actor>
```

```
        // Emits <Actor, count>
85
        public void reduce(Text actor, Iterable<Text> coactors,
    Context context)
87              throws IOException, InterruptedException {
            HashSet<Text> actorList = new HashSet<Text>();
89          for (Text coactor : coactors){
                actorList.add(coactor);
91          }
            context.write(actor, new IntWritable(actorList.size
    ()));
93      }
    }
95
    public static void main(String[] args) throws Exception {
97      Configuration conf = new Configuration();

99      Job roundOneJob = new Job(conf, "Round01");
        roundOneJob.setJarByClass(dk.itu.sad2.Popular.class);
101
        roundOneJob.setOutputKeyClass(Text.class);
103     roundOneJob.setOutputValueClass(Text.class);

105     roundOneJob.setMapperClass(RoundOneMap.class);
        roundOneJob.setReducerClass(RoundOneReduce.class);
107
        roundOneJob.setInputFormatClass(TextInputFormat.class);
109     roundOneJob.setOutputFormatClass(TextOutputFormat.class)
    ;

111     FileInputFormat.addInputPath(roundOneJob, new Path(args
    [0]));
        FileOutputFormat.setOutputPath(roundOneJob, new Path(
    args[1]));
113


115     roundOneJob.waitForCompletion(true);

117     if (roundOneJob.isSuccessful()) {
            Job roundTwoJob = new Job(conf, "Round02");
119         roundTwoJob.setJarByClass(dk.itu.sad2.Popular.class)
    ;

121         roundTwoJob.setOutputKeyClass(Text.class);
            roundTwoJob.setOutputValueClass(Text.class);
123
            roundTwoJob.setMapperClass(RoundTwoMap.class);
125         roundTwoJob.setReducerClass(RoundTwoReduce.class);

127         roundTwoJob.setInputFormatClass(TextInputFormat.
    class);
            roundTwoJob.setOutputFormatClass(TextOutputFormat.
    class);
129
```

```
            FileInputFormat.addInputPath(roundTwoJob, new Path(
    args[2]));
            FileOutputFormat.setOutputPath(roundTwoJob, new Path
    (args[3]));
            roundTwoJob.waitForCompletion(true);
        }
    }

}
```

size_estimator.py - Our implementation of the method introduced in Section 4.2

```
import pickle
import random

k = 256
#num_actors = 817718
#num_movies_with_actors = 300252
#num_movies_total = 388269
#res 2: 84546500.0004 (k: 200)
#res 3: 84546500.0 (k: 1000)
#res(k=500) = 105683125
#res(k=200) = 84545600

look_at = [621468, 372839, 74450, 212581, 152868, 22585,
           233082, 245158, 209799, 433904] # Top 10 film_count
    actors

m = 845465  #largest item to be hashed

prime = 845491.0

#a_1 = 1258.0
a_1 = random.random() * 10000
#b_1 = 8968.0
b_1 = random.random() * 10000
#a_2 = 1176.0
a_2 = random.random() * 10000
#b_2 = 759.0
b_2 = random.random() * 10000

def h1(x):
    return (((a_1 * x + b_1) % prime) % max(look_at)) / max(
    look_at)

def h2(x):
    return (((a_2 * x + b_2) % prime) % m) / m

def h(x,y):
    val = (h1(x) - h2(y)) % 1.0
    if h1(x) > 1.0 or h2(y) > 1.0:
        print "ALARM", h1(x), h2(y)
        exit()
```

```
40      return val

42  def sort_by_h1(Ai):
        return sorted(Ai, key=lambda x: h1(x))
44

46  def sort_by_h2(Ci):
        return sorted(Ci, key=lambda x: h2(x))
48
    def get_actor_dict():
50      #actor->list of movies
        with open('../projections/cached.pickle', 'rb') as handle:
52          return pickle.load(handle)

54  def build_movie_to_actor_index(actor_dict):
        #movie->list of actors
56      #create the "who played in this movie index"
        movie_dict = {}
58      for actor_id in actor_dict:
            movies = actor_dict[actor_id]
60          for movie_id in movies:
                if (not movie_id in movie_dict):
62                  movie_dict[movie_id] = []
                movie_dict[movie_id].append(actor_id)
64      return movie_dict

66  def dis_items(movie_to_actor_index):
        #setup
68      B = movie_to_actor_index.keys()
        p_init = 1
70      p_max = p_init

72      buffer_dict = {}
        for key in look_at:
74          buffer_dict[key] = (set(), set(), p_init)

76      #algorithm
        pos = 0
78      for i in B:
            pos += 1
80          Ai = list(set(look_at).intersection(set(
        movie_to_actor_index[i])))
            if len(Ai) == 0:
82              continue

84          Ci = movie_to_actor_index[i]

86          x = sort_by_h1(Ai)
            y = sort_by_h2(Ci)
88          s_bar = 0  # 0-indexes in python
            for t in range(0, len(Ci)):
90              while h(x[s_bar], y[t]) > h(x[s_bar - 1], y[t]):
                    s_bar = (s_bar + 1) % len(Ai)
92              s = s_bar
```

```
                first = True
94              while h(x[s], y[t]) < p_max and (s_bar != s or first
    ):
                    first = False
96                  (S, F, p) = buffer_dict[x[s]]
                    if h(x[s], y[t]) < p:
98                      F.add((x[s], y[t]))
                        if len(F) == k:
100                         (p, S) = combine(S, F)
                            buffer_dict[x[s]] = (S, set(), p)
102                         p_max = find_p_max(buffer_dict)
                    s = (s + 1) % len(Ai)
104     combine_all(buffer_dict)
        for key in buffer_dict:
106         (S, F, p) = buffer_dict[key]
            if len(S) == k:
108             print key, int(k/p)
            else:
110             print key, None

112 def combine(S, F):
        temp_list = list(S.union(F))
114     temp_list = sorted(temp_list, key=lambda t: h(t[0], t[1]))

116     """
        find the k smallest elements in S union F, set them to S and
         return S and
118     the median element
        """
120     i = min(k-1, len(temp_list) - 1)
        x, y = temp_list[i]
122     v = h(x, y)
        S = set(temp_list[0:k])
124     #print "Combined, new p=" + str(v)
        return v, S
126
    def find_p_max(buffer_dict):
128     p_max = 1
        for key in buffer_dict:
130         (S, F, p) = buffer_dict[key]
            if p > p_max:
132             p_max = p
        return p_max
134
    def combine_all(buffer_dict):
136     for key in buffer_dict:
            (S, F, p) = buffer_dict[key]
138         if len(S) + len(F) > 0:
                (p, S) = combine(S, F)
140             buffer_dict[key] = (S, set(), p)

142 actor_dict = get_actor_dict()
    index = build_movie_to_actor_index(actor_dict)
144 dis_items(index)
```

Popular.py - Project sql data into text file

```
import sys
import pickle

sys.setrecursionlimit(50000)

USE_CACHE = True

if not USE_CACHE:
  import MySQLdb as mdb

SQL_GET_ACTORS_IN_MOVIES = """
SELECT A.id, M.id FROM actors as A
  JOIN roles R
  ON R.`actor_id` = A.`id`
    JOIN movies M
    ON M.`id` = R.`movie_id` """

def get_data():
  con = mdb.connect('127.0.0.1', 'root', '', 'imdb');
  cur = con.cursor()
  cur.execute(SQL_GET_ACTORS_IN_MOVIES)
  rows = cur.fetchall()

  actor_dict = {}

  for row in rows:
    actor_id = int(row[0])
    movie_id = int(row[1])

    if actor_id in actor_dict:
      movie_list = actor_dict[actor_id]
      movie_list.append(movie_id)
    else:
      movie_list = [movie_id]
    actor_dict[actor_id] = movie_list

  return actor_dict

def print_data(data):
  for key in data:
    val = data[key]
    to_print = str(key) + " "
    for movie_id in val:
      to_print += str(movie_id) + " "
    print to_print

if not USE_CACHE:
  data = get_data()
  with open('cached.pickle', 'wb') as handle:
    pickle.dump(data, handle)
else:
  with open('cached.pickle', 'rb') as handle:
```

```
    data = pickle.load(handle)

print_data(data)
```

sequential_popular.py - Sequential implementation of Popular

```
import sys
import pickle

def get_actor_dict():
  #actor->list of movies
  with open('../projections/cached.pickle', 'rb') as handle:
    return pickle.load(handle)

def build_reverse_index(actor_dict):
  #movie->list of actors
  #create the "who played in this movie index"
  movie_dict = {}
  for actor_id in actor_dict:
    movies = actor_dict[actor_id]
    for movie_id in movies:
      if (not movie_id in movie_dict):
        movie_dict[movie_id] = []
      movie_dict[movie_id].append(actor_id)
  return movie_dict

def count_co_actors_for_actor(actor_id, actor_dict, movie_dict):
  movies = actor_dict[actor_id]
  co_actors = {}
  for movie in movies:
    for actor in movie_dict[movie]:
      if actor != actor_id: #playing in a movie with yourself
    doesn't count
        co_actors[actor] = 1 #set bool flag to signal that actor
     played with co_actor
  return len(co_actors)

def build_output_list(actor_dict, movie_dict):
  output_list = []
  for actor_id in actor_dict:
    count = count_co_actors_for_actor(actor_id, actor_dict,
    movie_dict)
    if count > 0: #discard actors with zero count
      output_list.append(str(actor_id) + "\t" + str(count))
  return output_list

def do_print(output_list):
  for line in output_list:
    print(line)

actor_dict = get_actor_dict()
movie_dict = build_reverse_index(actor_dict)
do_print(build_output_list(actor_dict, movie_dict))
```

sort_and_expand - Sort the output, and add actor names

```
import pickle

USE_CACHE = True

if not USE_CACHE:
  import MySQLdb as mdb

SQL_GET_ACTOR_NAMES = """SELECT id,first_name, last_name FROM
    actors"""

def get_actor_names_from_db():
  con = mdb.connect('127.0.0.1', 'root', '', 'imdb');
  cur = con.cursor()
  cur.execute(SQL_GET_ACTOR_NAMES)
  rows = cur.fetchall()
  names = {}
  for row in rows:
    actor_id = int(row[0])
    first_name = str(row[1])
    last_name = str(row[2])
    names[actor_id] = first_name + " " + last_name
  return names

def read_output(location):
  data = []
  with open(location, 'r') as f:
    read_data = f.read().splitlines()
    for line in read_data:
      split = line.split("\t")
      actor_id = int(split[0])
      count = int(split[1])
      val = (actor_id, count)
      data.append(val)
  return data

def expand_with_actor_names(sorted_list, names):
  new_list = []
  for val in sorted_list:
    actor_id, count = val
    actor_name = names[actor_id]
    new_val = (actor_id, actor_name, count)
    new_list.append(new_val)
  return new_list

if not USE_CACHE:
  actor_names = get_actor_names_from_db()
  with open('cached.pickle', 'wb') as handle:
    pickle.dump(actor_names, handle)
else:
  with open('cached.pickle', 'rb') as handle:
    actor_names = pickle.load(handle)


#data = read_output("../data/Actor-Number.out")
```

```
54 data = read_output("../data/sequential_output.out")
   #print len(data)
56

58 data = expand_with_actor_names(data, actor_names)
   result = sorted(data,key=lambda x: (x[2], -x[0]))
60
   for val in result:
62   print val
```