Spark Execution on Fact Checking and Related Workloads

Brett Walenz
Chaoren Liu

1 Overview

The goal of this project was to gain insight into aspects of the Spark execution engine. In particular, we wished to gain insight into the effect different execution plans has on the overall running time, bandwidth required, and computation time across the cluster. Our task is a computation-heavy task derived from computational fact-finding[1], where the goal can be thought of as minimizing or maximizing the result of an aggregate query under varying parameters. We'd like to answer the following question:

Which two congressional representatives had the lowest voting agreement percentage with a span of at least 120 days, after January 1st 2012?

2 Dataset

The dataset is a collection of congressional voting records from GovTrack.us after January 1st, 2012, consisting of 2770 bills and approximately 900,000 individual votes. The data is initial in an SQL database, but we've denormalized the data to have the following structure:

Bill Id	Rep Id	Vote	Type	Chamber	Year	Date	Result	Name
h81-113.2013	A000361	Yea	procedural	h	2013	2013-03-19	Passed	Rodney Alexander
h81-113.2013	B001256	Yea	procedural	h	2013	2013-03-19	Passed	Michele Bachmann
h81-113.2013	B001279	Nay	procedural	h	2013	2013-03-19	Passed	Ron Barber

3 Approaches

A very basic approach for finding the vote correlation between representatives is the following. We first parse the votes and key them by billId. We then self-join, and transform the data into (agree, disagree) tuples. The last stage, reduceByKey, sums the number of times two people agree and disagree.

```
votes = load_votes(context)
votes = votes.map(parseVote)
joined = votes.join(votes)
counted = joined.map(count_votes)
counted = counted.reduceByKey(reduce_count)
```

This approach finds the vote correlation between all pairs of congressional representatives, but it doesnt consider the interval of comparison.

3.1 A Note on Intervals

We need a way then, to construct interval sets, which consist of a collection of intervals. For instance, a set of intervals might be exhaustive: [(1/1/2011, 12/31/2011), (1/2/2011, 12/31/2011), (1/1/2011, 1/30/2011), (1/2/2011, 1/30/2011),... (12/01/2011, 12/31/2011), (12/02/2011, 12/31/2011)]

The above set contains all possible intervals from January 1st, 2011 to December 31st, 2011. The interval contains approximately d(d-1)/2 intervals, or 66300 intervals for just one year. We can obviously reduce this set of intervals by noting many intervals contain the same data, or no data at all. And many intervals may be too small for our condition (spanning 100 votes). Thus, each interval in the set must span at least 120 days. We used the Pandas Python data analysis library for this task[2].

3.2 Approach 1 - All in One Shot

A naive expanded approach may then be something such as:

Listing 1: Naive approach

```
votes = load_votes(context)
votes = votes.map(parseVote)
intervals = create_interval_set(start, end, parameters)
intervals = intervals.filter([100 votes, no empty data, identical sets])
joined = votes.join(votes)
joined = joined.cartesian(intervals)
joined = joined.filter((votes not in each interval))
counted = joined.map(count_votes)
counted = counted.reduceByKey(reduce_count)
```

The algorithm is a fairly 'traditional' Spark approach. We parse each vote line into a $(billid, (values))$ representation, self-join the bills, and cartesian join the interval set. Its important to note that the above algorithm might be exceedingly large due to the cartesian join performed. Then we map the votes and construct a new key that looks similar to: $entity_a : entity_b : start : end$.

3.3 Approach 2 - Interval Iterations

Listing 2 is similar to Listing 1, but instead of performing a cartesian join we iterate over the interval set. In this approach, we don't create new keys of the form $entity - a : entity - b : start : end$, instead we store a map on the drive that holds the lowest vote percentage for an original key $entity - a : entity - b$.

Listing 2: Iterative approach

```
interval_map = dict()
for (id, start, end) in intervals:
  func = partial(filter_by_int, start=start, end=end)
  temp = counted.filter(func)
  results = temp.reduceByKey(reduce_count)
  results = results.filter(filter_count)
  result_list = results.collect()
  """ Store the lowest result in a dictionary. """
  for result in result_list:
    (key, (agree, disagree, pct)) = result
    if pct_map[key] > pct:
      pct_map[key] = pct
      interval_map[key] = result
```

3.4 Approach 3 - Grouped Key Computations

We can note that keys of the form $entity - a : entity - b$ have a fairly small number of values: the number of shared votes the two representatives have. If our interval set is also fairly small, we can perform this computation in one batch on a single node without worrying about memory.

Listing 3: Grouped approach

```
raw_votes = load_votes(context)
big_intervals = interval.interval_set('1/1/2012', '1/1/2014', freq='15D',
    max_delta=pandas.Timedelta(days=120))
bills = raw_votes.map(keyByBillId)
```

```
    joined = bills.join(bills, 128)
    joined = joined.filter(filter_join)
    counted = joined.map(count_votes)
    big_ints_rdd = context.parallelize([big_intervals], 1)
    grouped = counted.groupByKey(128)
    int_grouped = grouped.cartesian(big_ints_rdd)
    results = int_grouped.map(calculate_percent_grouped)
```

Listing 3 does exactly as described. The first part of the algorithm is the same as the first two approaches: self-join the bills, and emit a key, (1, 0, date) if the vote agrees, and a key, (0, 1, date) if the votes do not agree. Next, we parallelize our intervals so they are all within one array and one partition, and then we group the counted votes by key. Then we cartesian the intervals with the grouped data, producing a data set with a rough schema:

Listing 4: Schema of grouped function

```
[(key, list(agree, disagree, date)), list(start_date, end_date)]
```

Listing 5: Grouped percent function

```
def calculate_percent_grouped(((key, counts), intervals)):
  low = 1
  for id, start, end in intervals:
    total_a = 0
    total_d = 0
    pct = 1
    for agree, disagree, d in counts:
      if d >= start and d <= end:
        total_a += agree
        total_d += disagree
    if total_a == 0 and total_d == 0:
      total_d = 1
    pct = total_a / float(total_a + total_d)
    if pct < low:
      low = pct
      low_tuple = (start, end, total_a, total_d)
  return (key, low_tuple)
```

Listing 5 shows the worker function performed by the slaves. Every key has a collection of (agree, disagree, date) values, and a set of intervals. Then we iterate through the intervals and for each interval calculate the vote correlation. If it's the new low, save it as such.

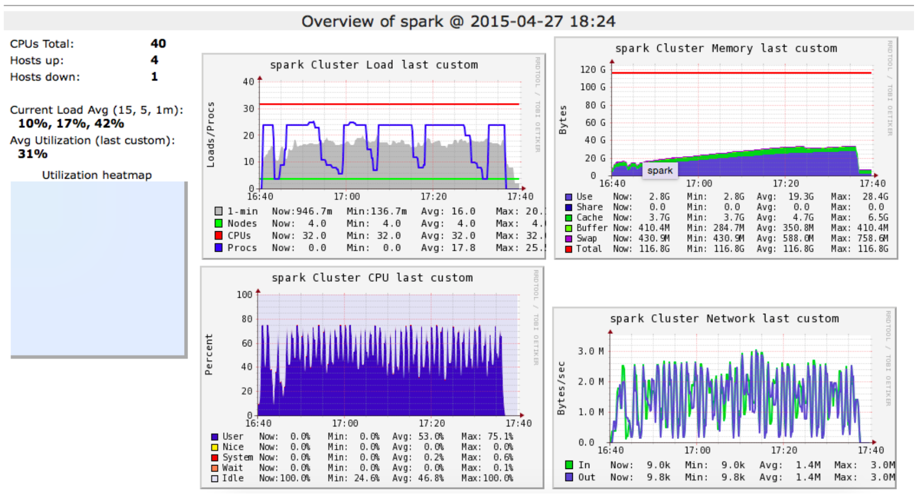
4 Experiments

We tested our approaches on a 5 node Amazon EC2 cluster running Spark 1.2.0. Each EC2 instance was an m3.2xlarge, which is an 8 vCPU, 30 GiB machine, and 2 x 80 GB SSD's, although these weren't used during the runs. The voting file is an uncompressed text file with size 250 MB, consisting of roughly 1000 representatives between the House of Representatives and Senate. The number of joins resulting from this is approximately 130,000, and the number of intervals analyzed was 364, or about 47×10^6 distinct computations.

Runtime (minutes)		
Naive	Iterative	Grouped
32	47	7

Figure 1: Overall runtime

The iterative approach is the slowest approach, and if we examine the Ganglia report from the run we can quickly see why.


Overview of spark @ 2015-04-27 18:24
Utilization heatmap
spark Cluster Load last custom
Loads/Procs
spark Cluster Memory last custom
Bytes
spark
spark Cluster CPU last custom
Percent
spark Cluster Network last custom
Bytes/sec

```
CPUs Total:          40
Hosts up:            4
Hosts down:          1

Current Load Avg (15, 5, 1m):
  10%, 17%, 42%
Avg Utilization (last custom):
  31%
```

```
1-min   Now:946.7m  Min:136.7m  Avg: 16.0   Max: 20.
Nodes   Now:  4.0   Min:  4.0   Avg:  4.0   Max:  4.
CPUs    Now: 32.0   Min: 32.0   Avg: 32.0   Max: 32.
Procs   Now:  0.0   Min:  0.0   Avg: 17.8   Max: 25.
```

```
Use    Now:   2.8G  Min:   2.8G  Avg:  19.3G  Max:  28.4G
Share  Now:   0.0   Min:   0.0   Avg:   0.0   Max:   0.0
Cache  Now:   3.7G  Min:   3.7G  Avg:   4.7G  Max:   6.5G
Buffer Now: 410.4M  Min: 284.7M  Avg: 350.8M  Max: 410.4M
Swap   Now: 430.9M  Min: 430.9M  Avg: 588.0M  Max: 758.6M
Total  Now: 116.8G  Min: 116.8G  Avg: 116.8G  Max: 116.8G
```

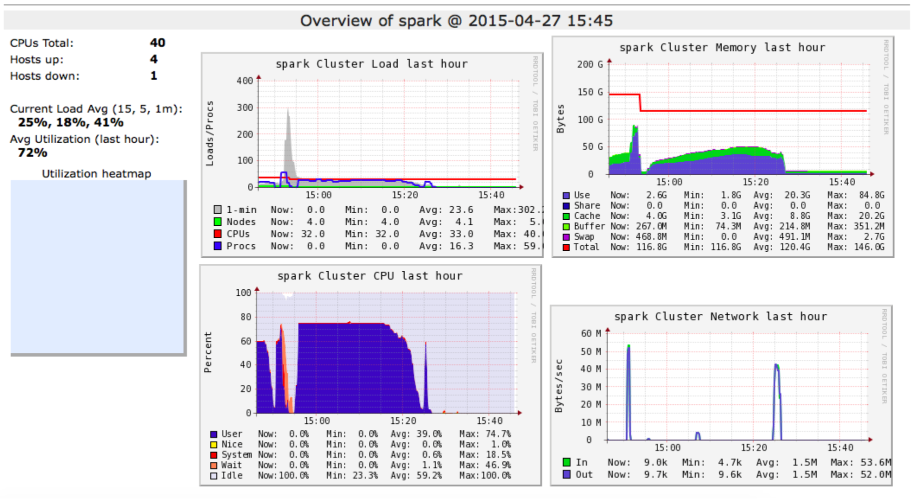
```
User   Now:  0.0%  Min:  0.0%  Avg: 53.0%  Max: 75.1%
Nice   Now:  0.0%  Min:  0.0%  Avg:  0.0%  Max:  0.0%
System Now:  0.0%  Min:  0.0%  Avg:  0.2%  Max:  0.6%
Wait   Now:  0.0%  Min:  0.0%  Avg:  0.0%  Max:  0.1%
Idle   Now:100.0%  Min: 24.6%  Avg: 46.8%  Max:100.0%
```

```
In    Now:  9.0k  Min:  9.0k  Avg:  1.4M  Max:  3.0M
Out   Now:  9.8k  Min:  9.8k  Avg:  1.4M  Max:  3.0M
```

Figure 2: Ganglia display for iterative approach

In Figure 2, we can see the bursty nature of both the CPU and bandwidth. Overall CPU utilization was only 31%, and cluster bandwidth was relatively high with an average of 1.4 MB/sec. The iterative approach induces this type of behavior because every iteration incurs overhead on starting a new set of Spark jobs, while at the end of each iteration a *collect*() is called, incurring a high bandwidth cost at each iteration.

The so-called Naive approach performs better and we see much higher CPU utilization overall. However, we assert that the bottleneck in this case is the network overhead incurred at the end of the job: approaching 50MB/sec. The reason for this is that we create an extremely dense set of keys of the form *entity* : *entity* : *start* : *end*, which as noted can be quite large. So a very large shuffle occurs in order for the reduceByKey to function, and then the results from that can be collected.



Overview of spark @ 2015-04-27 15:45

```
CPUs Total:        40
Hosts up:          4
Hosts down:        1

Current Load Avg (15, 5, 1m):
  25%, 18%, 41%
Avg Utilization (last hour):
  72%
```

```
spark Cluster Load last hour
1-min   Now:  0.0   Min:  0.0   Avg: 23.6   Max:302.
Nodes   Now:  4.0   Min:  4.0   Avg:  4.1   Max:  5.
CPUs    Now: 32.0   Min: 32.0   Avg: 33.0   Max: 40.
Procs   Now:  0.0   Min:  0.0   Avg: 16.3   Max: 59.
```

```
spark Cluster Memory last hour
Use    Now:   2.6G   Min:   1.8G   Avg:  20.3G   Max:  84.8G
Share  Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
Cache  Now:   4.0G   Min:   3.1G   Avg:   8.8G   Max:  20.2G
Buffer Now: 267.0M   Min:  74.3M   Avg: 214.8M   Max: 351.2M
Swap   Now: 468.8M   Min:   0.0    Avg: 491.1M   Max:   2.7G
Total  Now: 116.8G   Min: 116.8G   Avg: 120.4G   Max: 146.0G
```

```
spark Cluster CPU last hour
User   Now:  0.0%   Min:  0.0%   Avg: 39.0%   Max: 74.7%
Nice   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  1.0%
System Now:  0.0%   Min:  0.0%   Avg:  0.6%   Max: 18.5%
Wait   Now:  0.0%   Min:  0.0%   Avg:  1.1%   Max: 46.9%
Idle   Now:100.0%   Min: 23.3%   Avg: 59.2%   Max:100.0%
```

```
spark Cluster Network last hour
In    Now:  9.0k   Min:  4.7k   Avg:  1.5M   Max: 53.6M
Out   Now:  9.7k   Min:  9.6k   Avg:  1.5M   Max: 52.0M
```

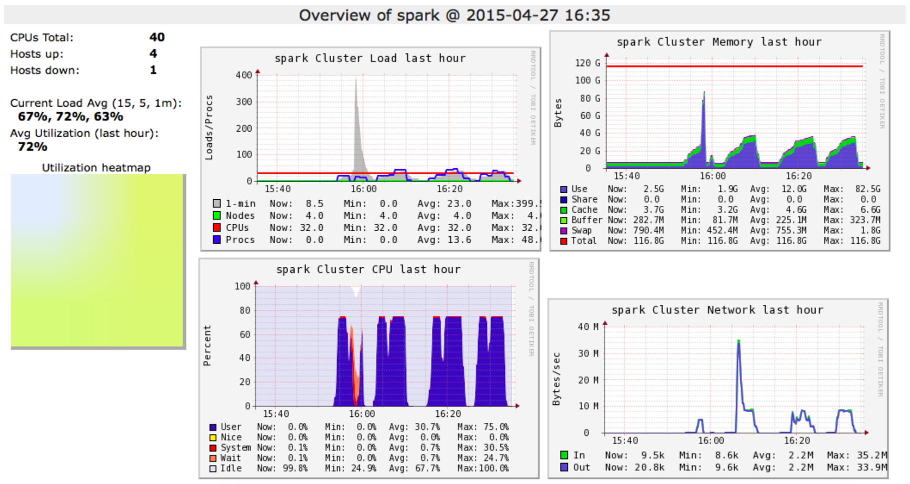
Figure 3: Ganglia display for naive approach

This led to the third approach with the realization that each computation isn't that expensive: summing the agree and disagree votes for each interval is a straightforward process as long as the data is within one node and we can iterative over the data. In addition, we save almost the entirety of the shuffle from the Naive approach since we calculate the minimum for a key on a node, and send only the min back to the driver.

Figure 4 is actually the display of multiple runs, since each run was under 8 minutes. We can see that while active, the grouped approach has roughly the same CPU utilization as the Naive approach but overall uses less memory. This is because we are performing a cartesian in chunks, rather than row by row, resulting in a lot less duplication.

In the end, the grouped approach closely resembles traditional parallel computing concepts, but it's impressive the ease with which Spark supports this use case. The only drawback to this approach is that we have to be explicitly aware of the size of the work being performed, as we can overload a nodes memory if a collection is too large (be it a key collection or the set of intervals).

Lastly, a note about data analytics and performance. A recent paper, *Making Sense of Performance in Data Analytics Frameworks*[3] discussed some counter-findings to the accepted mantras about commmon performance pitfalls in data analytics. However, in our


Overview of spark @ 2015-04-27 16:35
Utilization heatmap
spark Cluster Load last hour
Loads/Procs
spark Cluster Memory last hour
Bytes
spark Cluster CPU last hour
Percent
spark Cluster Network last hour
Bytes/sec
RRDTOOL / TOBI OETIKER

```
CPUs Total:          40
Hosts up:            4
Hosts down:          1

Current Load Avg (15, 5, 1m):
  67%, 72%, 63%
Avg Utilization (last hour):
  72%
```

```
1-min   Now:   8.5   Min:   0.0   Avg: 23.0   Max:399.
Nodes   Now:   4.0   Min:   4.0   Avg:  4.0   Max:  4.
CPUs    Now:  32.0   Min:  32.0   Avg: 32.0   Max: 32.
Procs   Now:   0.0   Min:   0.0   Avg: 13.6   Max: 48.
```

```
Use     Now:   2.5G   Min:   1.9G   Avg:  12.0G   Max:  82.5G
Share   Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
Cache   Now:   3.7G   Min:   3.2G   Avg:   4.6G   Max:   6.6G
Buffer  Now: 282.7M   Min:  81.7M   Avg: 225.1M   Max: 323.7M
Swap    Now: 790.4M   Min: 452.4M   Avg: 755.3M   Max:   1.8G
Total   Now: 116.8G   Min: 116.8G   Avg: 116.8G   Max: 116.8G
```

```
User    Now:  0.0%   Min:  0.0%   Avg: 30.7%   Max: 75.0%
Nice    Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
System  Now:  0.1%   Min:  0.0%   Avg:  0.7%   Max: 30.5%
Wait    Now:  0.1%   Min:  0.0%   Avg:  0.7%   Max: 24.7%
Idle    Now: 99.8%   Min: 24.9%   Avg: 67.7%   Max:100.0%
```

```
In    Now:  9.5k   Min:  8.6k   Avg:  2.2M   Max: 35.2M
Out   Now: 20.8k   Min:  9.6k   Avg:  2.2M   Max: 33.9M
```

Figure 4: Ganglia display for grouped approach

case our data was small enough that we were able to drastically change the algorithm since we had more flexibility in memory availability. It should be noted, however, that the very existence of the grouped approach is because of the extreme slowness of the reduceByKey and shuffle in the Naive approach. There may be some space where small data, large computation workloads can be optimized in similar fashion.

5 Conclusion

In this project, we examined three different approaches to the problem of finding vote correlation among United States congressional representatives. We noticed that a grouped approach was the quickest, and leveraged the fact that the values for each key were relatively small, meaning we could fit everything onto one machine and perform all computation at a node, avoiding expensive shuffles and repartitionings.

References

[1] S. Cohen, C. Li, J. Yang, and C. Yu. Computational journalism: A call to arms to database researchers. In *CIDR*, volume 2011, pages 148–151, 2011.

[2] W. McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[3] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks.

A Code

All the code for this project can be found at: https://github.com/bwalenz/cs516-team. We've included the code for the grouped approach, but have omitted the others for brevity.

B Grouped Approach Code

```
from pyspark import SparkContext
from functools import partial
from collections import defaultdict
import StringIO
import csv
import datetime
import permute.interval as interval
import pandas
import numpy as np
import sqlite3

votes_file = '2012-curr-full-votes.csv'
#master = "local[4]"
master = "spark://ec2-54-83-184-241.compute-1.amazonaws.com:7077"
def load_votes(context):
    votes_data = context.textFile(votes_file, use_unicode=False).cache()
    return votes_data


def keyByBillId(line):
    """
    Key by the bill id to be used in the self join step. The tuple looks like:
    (bill_id, (person_id, date, vote))
    """
    df = '%Y-%m-%d'
    input = StringIO.StringIO(line)
    reader = csv.reader(input)
    tup = reader.next()
    if tup:
      tup[6] = datetime.datetime.strptime(tup[6], df).date()
    reduced = []
    #person id
    reduced.append(tup[1])
    #date
    reduced.append(tup[6])
    #vote (yay, nay, etc)
    reduced.append(tup[2])
    return (tup[0].strip(), [tup[1].strip(), tup[6], tup[2].strip()])

def count_votes((key, joined_tuple)):
    """
    Map function that counts the votes between two entries.
    Called on a self-joined bills entry, so the incoming tuple looks like:
    (bill_id, ((person_id, date, vote), (person_id, date, vote))
    """
    left = joined_tuple[0]
    right = joined_tuple[1]
    left_id = left[0]
    right_id = right[0]
    key = left_id + ":" + right_id
    agree = 0
    disagree = 0
    if left[2] == right[2]:
        agree = 1
    else:
        disagree = 1
    return (key, (agree, disagree, left[1]))


def reduce_count(left, right):
    agree = left[0] + right[0]
    dis = left[1] + right[1]
    return (agree, dis)


def filter_count((key, (agree, dis))):
    """
    Filter any entries that don't have at least 50 votes.
    This is also needed because some entries get filtered out completely
    and so the previous reduce_count stage is not called for some tuples,
    resulting in the third entry in this tuple being a date object instead
    of a percent.
    """

    if agree + dis < 50:
        return False
    else:
        return True
```

```
def filter_join((key, (left, right))):
    """
    Avoids duplicate calculations/entries
    ex: ("p_id_1:p_id_2", "p_id_2:p_id_1"), removes the right one.
    """
    if left[0] < right[0]:
        return True
    return False

def calculate_percent_grouped_2(((key, counts), intervals)):
  low = 1
  for id, start, end in intervals:
    total_a = 0
    total_d = 0
    pct = 1
    for agree, disagree, d in counts:
      if d >= start and d <= end:
        total_a += agree
        total_d += disagree
    if total_a == 0 and total_d == 0:
      total_d = 1
    pct = total_a / float(total_a + total_d)
    if pct < low:
      low = pct
      low_tuple = (start, end, total_a, total_d)
  return (key, low_tuple)

def run(context):
    """ Data is in the following format: (bill_id, person_id, vote, type, chamber,
         year, date, session, status, extra).
        1. Key everything by bill_id, parse the date correctly, return (bill_id, (
            person_id, vote, date))
        2. Self join bills RDD to itself.
        3. Remove duplicate entries (via left.person_id < right.person_id) and
            comparisons with self.
        4. Map the joined data to a (person_id:person_id, (agree, disagree)) RDD.
        5. Grouped all the counts by key into one array and partition.
        6. Cartesian join the intervals.
        7. Run the grouped computation function on each group.
    """
    raw_votes = load_votes(context)
    big_intervals = interval.interval_set('1/1/2012', '1/1/2014', freq='15D',
        max_delta=pandas.Timedelta(days=120))
    bills = raw_votes.map(keyByBillId)
    joined = bills.join(bills, 128)
    joined = joined.filter(filter_join)
    counted = joined.map(count_votes)
    big_ints_rdd = context.parallelize([big_intervals], 1)
    grouped = counted.groupByKey(128)
    int_grouped = grouped.cartesian(big_ints_rdd)
    results = int_grouped.map(calculate_percent_grouped_2)

    return results

if __name__ == "__main__":
    context = SparkContext(master, "Congress Correlation")
    results = run(context).collect()
    with open('output.csv', 'w') as csvfile:
      writer = csv.writer(csvfile)
      for result in results:
        (persons, (start, end, agree, disagree)) = result
        person_split = persons.split(":")
        writer.writerow([person_split[0], person_split[1], str(start), str(end),
            str(agree), str(disagree), str((agree / (agree + disagree)))])
```