Applying Meta-Blocking to Improve Efficiency in Entity Resolution

Tobias Ammann
tag@adnm.ch

May 11, 2014

Abstract

This paper compares two implementations of meta-blocking in terms of runtime and memory usage, and measures the accuracy of meta-blocking using a subset of the Musicbrainz database. We find that the implementation using a reversed index is more efficient than the naive implementation. Furthermore, we find that the dataset in its current form is unsuitable for meta-blocking, due to incomplete records and the presence of high-frequency tokens, which cause both implementations to approach $O(n^2)$ runtime and memory consumption (n being the number of records).

1 Introduction

Real world datasets often contain duplicate records representing the same entity. There are many reasons for this: data entry mistakes, merging of different data sources, etc. The task of finding these duplicates is called entity resolution (ER). The main problem of ER is its runtime complexity of $O(n^2)$ (n being the number of records), which makes it impractical to exhaustively compare all records with each other. The runtime cost can be improved by intelligently dividing records into blocks and only comparing records within the same block. One way to create such blocks is to assign all entities that share the same token to the same block, e.g. John Smith, Joe Smith, and Fred Smith are all assigned to the block "Smith". Meta-blocking [1] is an additional step that is inserted between the creation of the blocks and comparing the entities. Meta-blocking transforms one set of blocks into another set of blocks to further improve the efficiency of any blocking algorithm.

2 Meta-Blocking

The input to meta-blocking is a set of blocks. Each block is itself a set of entities and represents some kind of connection between the entities in the set, e.g. the same surname. The output of meta-blocking is a list of entity pairs that are promising candidates for a comparison. These pairs can be viewed a independent blocks, one block per entity pair.

Meta-blocking aims to increase the efficiency of blocking ER by reducing redundancy present in the input blocks. This is done using three ideas: graphs, weighting, pruning.

1. Meta-blocking uses a graph to represent the entity-to-block relationships. Vertices represent entities that are connected by weighted edges if the entities share one or more blocks.

2. The weight of an edge is computed as the number of blocks that two entities share. Hence sharing multiple blocks results in a higher likelihood of being included in the output.

3. All edges with a below average weight are pruned from the graph, which only leaves the more similar entities for further consideration.

In the remainder of section 2 we present two different implementations of meta-blocking: The first implementation, $BATCH$, creates the graph in a naive way. The second implementation, $REVIDX$, processes the data with the help of an inverted index.

2.1 Batch Processing Implementation

Given a set $\bar{B}$ of blocks, BATCH generates a graph $G(E, N)$ and prunes G as follows:

1. Let $\bar{E}$ be a bag of sorted edges. For each block in $\bar{B}$, insert all entity pairs in $\bar{E}$. Keep the two entities e_1 and e_2 in each pair sorted ($e_1 < e_2$) to avoid duplicates.

2. Scan $\bar{E}$ to compute the average edge weight W_{avg} by dividing the number of entity pairs in $\bar{E}$ by the number of distinct edges: $W_{avg} = \frac{|\bar{E}|}{N_{distinct}}$.

3. Scan $\bar{E}$ to output all distinct edges whose frequency is above average ($W_{pair} \geq W_{avg}$).

Algorithm 1 BATCH($\bar{B}_{input}$)

Input: $\bar{B}_{input}$.
Output: $\bar{B}_{output}$.
  $\bar{E}$: Bag of edges (including duplicates).
  `// Graph construction:`
  $\bar{E}$ = all entity pairs of all blocks in $\bar{B}$.
  sort $\bar{E}$.
  $N_{distinct} = 1$
  $pair_{last} = \bar{E}_0$
  for $pair$ in $\bar{E}_{1..N}$ **do**
    if $pair \neq pair_{last}$ **then**
      $N_{distinct}$++
      $pair_{last} = pair$
    end if
  end for
  `// Graph pruning:`
  $W_{avg} = \frac{|\bar{E}|}{N_{distinct}}$
  $pair_{last} = \bar{E}_0$.
  $W_{pair} = 1$.
  for $pair$ in $\bar{E}_{1..N}$ **do**
    if $pair \neq pair_{last}$ **then**
      if $W_{pair} \geq W_{avg}$ **then**
        add $pair$ to $\bar{B}_{output}$.
      end if
      $W_{pair} = 0$
      $pair_{last} = pair$
    end if
    W_{pair}++
  end for
  if $W_{pair} \geq W_{avg}$ **then**
    add $pair$ to $\bar{B}_{output}$.
  end if
  return $\bar{B}_{output}$.

2.2 Reverse Index Implementation

The $REVIDX$ implementation is based on [1]. $REVIDX$ does not keep track of the entire graph. Instead, it works on each input block separately. First, it calculates the weight of all edges and the number of distinct edges in a given block to compute the average weight. It then does a second scan

during which it again calculates each edge weight and then adds all edges with an above average weight to the list of output blocks.

In order for the edge weight calculation to be efficient, $REVIDX$ uses a reversed index $\bar{R}$ to store the blocks associated with each entity. It ensures the correct computation by iterating through the blocks in sorted order, and by keeping each entity's blocks in the reversed index in the same order. With these constraints on ordering, $REVIDX$ can avoid keeping track of all edges.

Algorithm 2 GETWEIGHT(b, $\bar{R}$, $pair$)

Input: b (current block), $\bar{R}$, $pair$.
Output: W_{pair}.

```
for b_i ∈ R̄_{pair_0} do
  for b_j ∈ R̄_{pair_1} do
    if b_i = b_j and not compared before b. then
      W_pair++
    else
      return -1
    end if
  end for
end for
return W_pair
```

Algorithm 3 ReverseIndex($\bar{B}_{input}$)

Input: $\bar{B}_{input}$
Output: $\bar{B}_{output}$

```
// Reversed Index:
```
$\bar{R}$: Reversed Index storing each entity's blocks.
```
// Graph construction:
```
$W_{total} = 0$
$N_{distinct} = 0$
for $\bar{b} \in \bar{B}_{input}$ in sorted order **do**
  for $pair \in \bar{b}$ **do**
    W_{pair} = GetWeight(b, $\bar{R}$, $pair$)
    if $w \neq -1$ **then**
      $W_{total} = W_{total} + W_{pair}$
      $N_{distinct}$++
    end if
  end for
end for
```
// Graph pruning:
```
$W_{avg} = W_{total}$ / $N_{distinct}$
for $\bar{b} \in \bar{B}_{input}$ in sorted order **do**
  for $pair \in \bar{b}$ **do**
    W_{pair} = GetWeight(b, R, $pair$)
    if $W_{pair} \geq W_{avg}$ **then**
      add $pair$ to $\bar{B}_{output}$
    end if
  end for
end for
return $\bar{B}_{output}$

3 Evaluation

We ran both implementations on a real-world dataset to measure accuracy, runtime, and memory usage.

3.1 Dataset

The dataset used to analyse both implementations is a subset of the Musicbrainz database. Each record in the dataset describes an artist by *name, type, area, gender, comment, begin year*, and *end year*. Additionally, each record contains an attribute *cluster* that identifies records that describe the same artist. To create the input blocks, the text of each input field was tokenised to yield single word tokens. The following table shows how the blocks are distributed depending on the size of the dataset.

N				Block Size			Blocks per Entity		
Records	Clusters	Blocks	1-E./B.	Min.	Max.	Avg.	Min.	Max.	Avg.
1000	696	1818	1416	1	558	3.15	1	15	5.72
2000	1309	3185	2440	1	1179	3.53	1	15	5.62
5000	3392	6708	4919	1	2794	3.96	1	23	5.32
10000	7133	11658	8394	1	5211	4.38	1	23	5.1
20000	12925	19835	13864	1	12768	5.01	1	23	4.97
30000	20098	27378	19041	1	18481	5.39	1	23	4.92

3.1.1 Notes and Observation on the Dataset

1. *1-E. / B.* is the number of blocks which only contain one entity. These blocks create no edges and are discarded during meta-blocking. On average 73.21% of blocks are discarded.

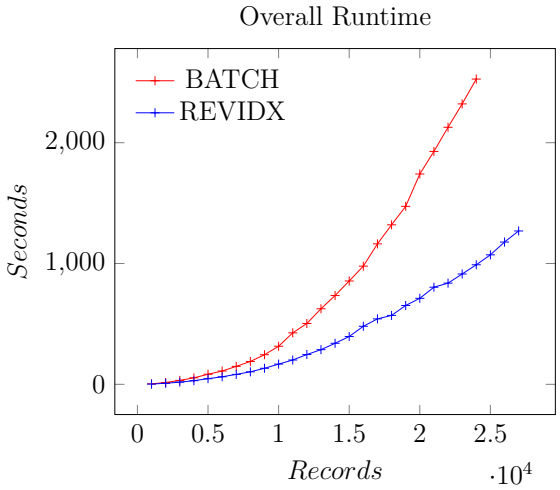
2. The decreasing average number of blocks per entity hints at a large number of sparse records. Given the number of fields in the dataset, we expect a lower bound of 6 blocks per entity for complete records.

3. The increasing maximum and average block sizes indicates the presence of high frequency tokens. On average 58.03% of all records share the largest block.

3.2 Performance analysis

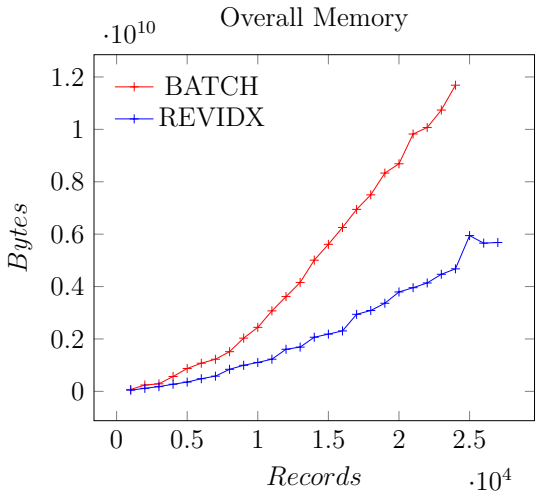
3.2.1 Comparison of Runtime

We measured the runtime of BATCH for increments of 1000 records up to 24000. Above 24000 BATCH runs out of memory. REVIDX was run up to 27000 records. The runtime increased polynomially for both implementations, because of the growing average and maximum block size. REVIDX was more efficient than BATCH for any number of records.



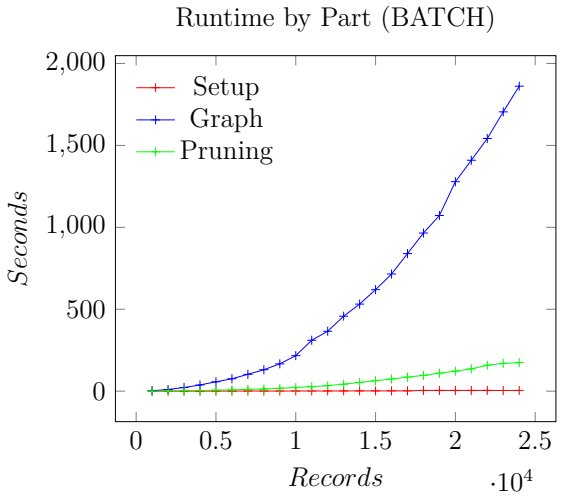
3.2.2 Comparison of Memory Usage

In terms of memory usage, BATCH required substantially more memory, because it keeps a sorted bag of all edges. REVIDX does not save any edges, thus its memory usage is dominated by the list of output blocks $\bar{B}_{output}$.

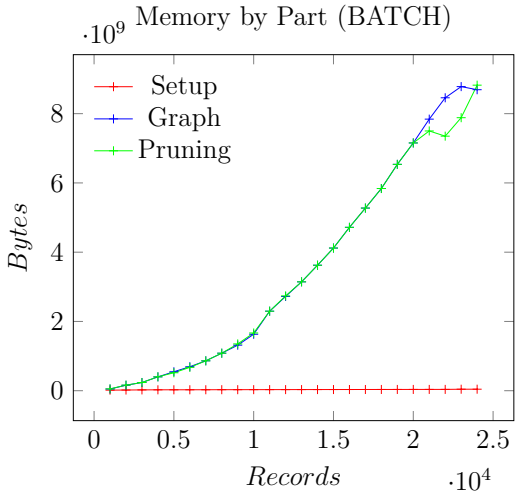


3.2.3 Detailed Analysis of BATCH

The runtime of BATCH is dominated by the construction of the graph, i.e. inserting all edges into $\bar{E}$ (*Graph*). *Pruning* is fast because it only involves two linear scans of $\bar{E}$. Tokenising the records prior to meta-blocking is virtually free (*Setup*).

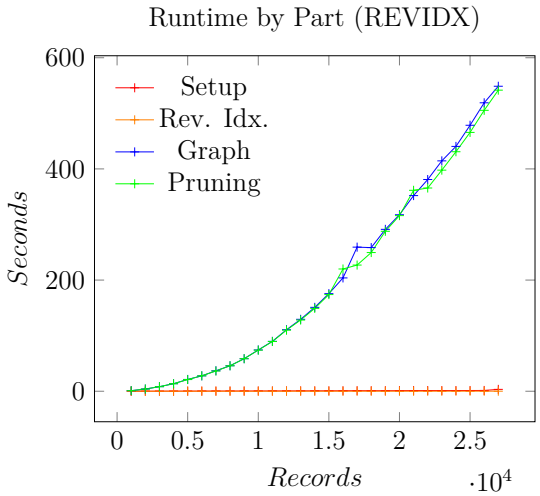


The memory consumption of BATCH is also dominated by the construction of the *Graph*. The later *Pruning* stage, only consumes marginally more memory for $\bar{B}_{output}$. The small reduction in memory consumption above 25 K records is an artefact of the implementation of $\bar{E}$.

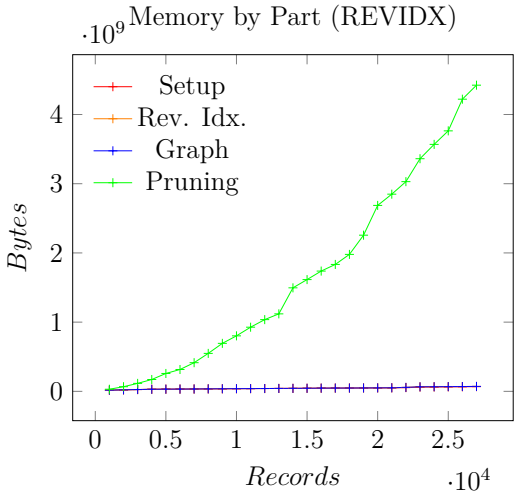


3.2.4 Detailed Analysis of REVIDX

Tokenising the records (*Setup*) and the creation of the reversed index (*Rev. Idx.*) are very fast and negligible compared to the runtime cost of calculating the weight of each edge twice, once during the calculation of W_{avg} and $N_{distinct}$ (*Graph*) and once during *Pruning*. Unlike BATCH, which stores the edge weights, REVIDX has to do duplicate work which slows down pruning.



In terms of memory usage, REVIDX requires very little memory until it stores the output blocks $\bar{B}_{output}$ (*Pruning*). The memory usage for $\bar{B}_{output}$ depends on the dataset. The polynomial increase of memory usage during *Pruning* is a consequence of the high number of false positive results.

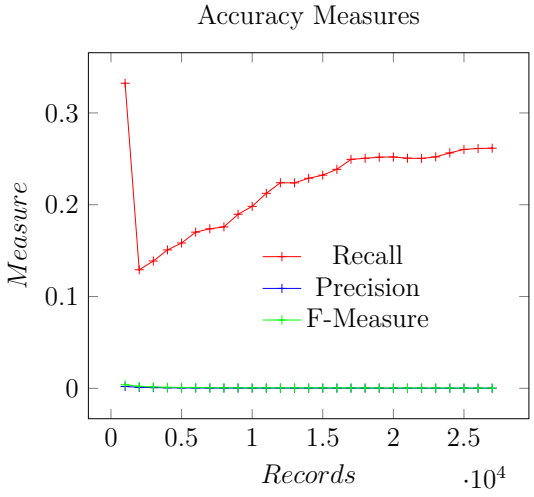


3.3 Accuracy of the method

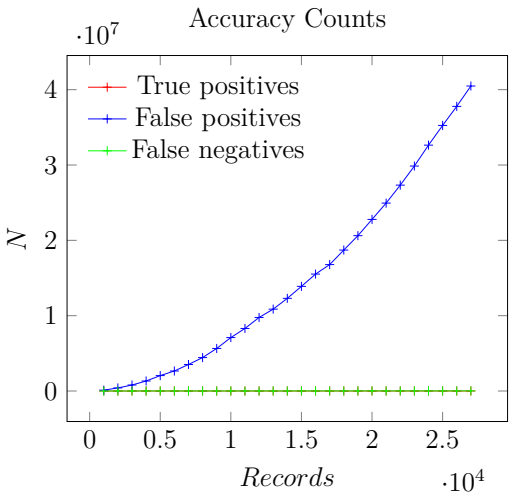
The quality of the output generated by meta-blocking was measured using *precision*, *recall*, and *f-measure*, by comparing $\bar{B}_{output}$ against a list of entity pairs generated using the *cluster* attribute of the dataset.

1. The *Precision* measures how many of the returned results are actually correct, and is defined as: $Precision = \frac{N_{true\ positive}}{N_{true\ positive} \div N_{false\ positive}}$
2. The *Recall* measures how many of the correct results are present in the output, and is defined as: $Recall = \frac{N_{true\ positive}}{N_{true\ positive} \div N_{false\ negative}}$
3. The F-Measure is defined as follows: $F\text{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$

F-measure was on average 0.00072 for all from 1000 to 27000 records. Precision was on average 0.00036. Recall increased with the size of the dataset, but stayed under 0.3 with only one exception.



There comparatively high recall results from the number of false positives $N_{false\ positives}$ increasing polynomially with the size of the dataset.



As can be seen in the example output tables below, some blocks are shared by a large number of unrelated entities, e.g. *type, area*, and music terms in *comment*. The rapidly growing maximum block size in the dataset confirms this. These blocks are what causes the number of false positives to grow polynomially with the number of records considered, and *recall* to increase.

Another problem observed in this dataset is that many records describing the same artist do not share any identifying blocks. Fantasy names, and sparse records mean that many duplicate records only share non-identifying information, e.g. Arthur Smith and Morgan Reno are the same artist, but since these are fake names, the two records only share *type*.

We also observe that many of the correctly discovered entity pairs were included in $\bar{B}_{output}$ on the basis of such non-identifying tokens rather than a more identifying attribute like *name*.

The example output tables below are based on the output of meta-blocking on 1000 records.

3.3.1 Output: True Positives

Weight	Id	Cluster	Name	Type	Area	Gender	Comment	Begin Year	End Year
2	344	344	Violent Femmes	Group	United States			1980	2009
	679870	344	Matt Haines	Person	United States	Male			
2	344	344	Violent Femmes	Group	United States			1980	2009
	66930	344	The Rip-Off Artist	Person	United States	Male			
2	258876	284	Lützenkirchen	Person					
	366859	284	Tobias Lützenkirchen	Person					
2	203514	1237	Mark J	Person					
	475805	1237	Mark Wiltshire	Person					
2	374936	742	SMP	Person	Germany		German trance producer		
	504953	742	High Noon at Salinas	Person	Germany				
2	466616	533	Adel	Person					
	671438	533	Adel Hafsi	Person	Germany	Male		1971	
2	659602	249	Jimmy Barnatán	Person					
	659603	249	Jaime Barnatán Pereda	Person				1981	
2	379	379	Glen Campbell	Person	United States	Male		1936	
	155358	379	Wedlock	Person	Netherlands	Male	Dutch DJ Patrick van Kerckhoven		
2	621002	78	Eased	Person				1970	
	640791	78	Dellé	Person	Germany		German reggae artist	1970	
2	275620	716	Outolintu	Person	Finland				
	479796	716	Overflow	Person	Finland		Finnish electronica producer Jürgen Sachau		
2	1587	1587	Deep Purple	Group	United Kingdom			1968	
	73899	1587	Ils	Person	United Kingdom	Male			
2	466616	533	Adel	Person					
	475218	533	Adel Dior	Person	Germany	Male		1971	
2	428727	1299	おみむらまゆこ	Person				1976	
	567370	1299	麻績村まゆ子	Person				1976	
2	104061	379	Asylum	Person	Netherlands		Dutch gabber producer Patrick van Kerckhoven		
	167028	379	DJ Ruffneck	Person	Netherlands				
2	167028	379	DJ Ruffneck	Person	Netherlands				
	241653	379	Ruffneck Alliance	Person					
2	108996	379	Morlock	Person			DJ Patrick van Kerckhoven - has song "Der Energy"		
	167028	379	DJ Ruffneck	Person	Netherlands				
2	94575	554	Celldweller	Person	United States	Male			
	276655	554	Klayton	Person		Male			
2	161356	742	Sunlounger	Person	Germany		trance artist Roger Shah		
	390575	742	Magic Wave	Person	Germany				

3.3.2 Output: False Positives

Weight	Id	Cluster	Name	Type	Area	Gender	Comment	Begin Year	End Year
7	637609	609	Izzy	Person	New Zealand		NZ hip hop artist		
	637611	751	PKS	Person	New Zealand		NZ hip hop artist		
7	525660	331	R'Ma	Person	New Zealand		NZ hip hop artist		
	637611	751	PKS	Person	New Zealand		NZ hip hop artist		
7	525659	181	Factor	Person	New Zealand		NZ hip hop artist		
	525660	331	R'Ma	Person	New Zealand		NZ hip hop artist		
7	681	681	The Romantics	Group	United States		US new wave band	1977	
	1149	1149	Missing Persons	Group	United States		US new wave band	1980	1986
7	365	365	Incubus	Group	United States		US alternative rock band	1991	
	933	933	Toad the Wet Sprocket	Group	United States		US alternative rock band	1986	
7	933	933	Toad the Wet Sprocket	Group	United States		US alternative rock band	1986	
	1217	1217	The Flys	Group	United States		US rock band	1993	
7	94	94	John Williams	Person	United States	Male	soundtrack composer & conductor	1932	
	1338	1338	Jerry Goldsmith	Person	United States	Male	soundtrack composer & conductor	1929	2004
7	525659	181	Factor	Person	New Zealand		NZ hip hop artist		
	637609	609	Izzy	Person	New Zealand		NZ hip hop artist		
7	1217	1217	The Flys	Group	United States		US rock band	1993	
	1585	1585	The Faint	Group	United States		US indie rock band	1994	
7	349	349	Yes	Group	United Kingdom		British progressive rock band	1968	
	1435	1435	Cream	Group	United Kingdom		British 1960s rock band	1966	1968
7	525659	181	Factor	Person	New Zealand		NZ hip hop artist		
	637611	751	PKS	Person	New Zealand		NZ hip hop artist		
7	1474	1474	The Ataris	Group	United States		US pop/punk band	1994	
	1585	1585	The Faint	Group	United States		US indie rock band	1994	
7	525660	331	R'Ma	Person	New Zealand		NZ hip hop artist		
	637609	609	Izzy	Person	New Zealand		NZ hip hop artist		
7	933	933	Toad the Wet Sprocket	Group	United States		US alternative rock band	1986	
	1585	1585	The Faint	Group	United States		US indie rock band	1994	
7	508	508	face to face	Group	United States		90s California punk band	1991	
	1503	1503	Rancid	Group	United States		Berkeley, California punk band	1991	
6	48	48	Helium	Group	United States		US indie rock featuring Mary Timony	1992	
	1585	1585	The Faint	Group	United States		US indie rock band	1994	
6	1000	1000	Tool	Group	United States		US progressive metal band	1990	
	1101	1101	Live	Group	United States		US alt rock band	1990	
6	933	933	Toad the Wet Sprocket	Group	United States		US alternative rock band	1986	
	1474	1474	The Ataris	Group	United States		US pop/punk band	1994	

3.3.3 Output: False Negatives

Weight	Id	Cluster	Name	Type	Area	Gender	Comment	Begin Year	End Year
1	416908	1569	Arthur Smith	Person	United Kingdom		UK DJ		
	471637	1569	Morgan Reno	Person					
1	241653	379	Ruffneck Alliance	Person					
	476782	379	Phoenix	Person	Netherlands	Male	Dutch Hardcore producer Patrick van Kerckhoven		
1	114703	344	Control X	Person					
	679870	344	Matt Haines	Person	United States	Male			
1	66154	1180	Shiva Chandra	Person	Germany	Male	Psychedelic trance artist	1972	
	211212	1180	Daniel Vernunft	Person					
1	240483	1513	Willem Faber	Person				1970	
	682143	1513	Talespin	Person					
1	131031	284	Karosa	Person					
	134438	284	LXR	Person					
1	330895	742	Pasha	Person			remix alias for Roger Shah	1972	
	390575	742	Magic Wave	Person	Germany				
1	139556	363	Photon Inc.	Person		Male			
	307115	363	The Don	Person			House artist Nathaniel Pierre Jones		
1	719	719	Lena	Person	Germany	Female	German house vocalist Lena Mahrt		
	501307	719	Lysander Pearson	Person					
1	437349	363	P-Ditty	Person					
	748767	363	Simon Says	Person	United States	Male	US house artist		
1	432408	735	Mat Ranson	Person					
	579143	735	Kwaidan	Person					
1	435109	1041	佐藤和奈	Person	Japan	Female		1981	
	739590	1041	棚町薫	Person					
1	118559	363	X Fade	Person					
	278594	363	Yvette	Person			trance alias for DJ Pierre		
1	128364	363	Raving Lunatics	Person					
	278594	363	Yvette	Person			trance alias for DJ Pierre		
1	276406	573	Boduf Songs	Person					
	493211	573	Mat Sweet	Person					
1	534677	19	弘世	Person				1979	
	552924	19	アルトノイラント	Person					
1	118559	363	X Fade	Person					
	437349	363	P-Ditty	Person					
1	441526	1104	Peter Waldmann	Person					
	464883	1104	DJ Gorge	Person					

4 Conclusion

Meta-blocking is very susceptible to problematic datasets. A few very common token and otherwise sparse records leads to the number of false positives growing polynomially with the number of records considered. Consequently, recall increases, but both precision and f-measure approach zero.

Furthermore, the large number of false positives affects runtime and memory usage for both implementations. In the case of all records sharing one token, the performance of meta-blocking becomes equivalent to the worst-case for ER of $O(n^2)$ (for n records).

REVIDX is a better implementation than BATCH in terms of runtime and memory consumption. However, neither implementation can handle the described problems of the dataset, since they are affected by it in the same way. Both implementation are still bound by the $O(n^2)$ of the ER problem, and all differences are essentially constant factors.

References

[1] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. Meta-blocking: Taking entity resolution to the next level. *IEEE Transactions on Knowledge and Data Engineering*, 99.

A Source Code

A.1 Online Repository

An electronic version of this work is available at Github:
`https://github.com/betabrain/fa-uzh-14`

A.2 BATCH

```
import sqlite3
import leveldb
import time
import string
import os
import collections
import struct
import shutil
import itertools
import functools
import pprint
import sys
import resource
import types
import blessings
import codecs
import sh
from psutil import Process as P; P = P()

# config

if len(sys.argv) == 2:
    n_records = int(sys.argv[-1])
else:
    n_records = 500

bad_values = set(list(string.ascii_letters + string.digits))

time_started = time.clock()

stats = {
    'Records.N': n_records,
    # 'time_started': time_started
}

# helpers

def info(*args, **kwargs):
    print >>sys.stderr, 'ARGS', args
    print >>sys.stderr, 'KWARGS', kwargs

def get_du(p):
    if os.path.exists(p):
        return int(str(sh.du('-k', p)).split()[0]) * 1024
    else:
        return 0L

```

```
48 get_wdb = functools.partial(get_du, 'batch.sqlite3')
49 get_ldb = functools.partial(get_du, 'batch.leveldb')
50
51 class timer(object):
52     def __init__(self, name='<block>'):
53         self.name = name
54         self.start_sys  = 0.0
55         self.start_user = 0.0
56         self.start_rss = 0L
57         self.start_disk = 0L
58     def __enter__(self):
59         cput = P.cpu_times()
60         memi = P.memory_info_ex()
61         self.start_sys  = cput.system
62         self.start_user = cput.user
63         self.start_rss  = memi.rss
64         self.start_disk = get_wdb() + get_ldb()
65     def __exit__(self, *args):
66         cput = P.cpu_times()
67         memi = P.memory_info_ex()
68         self.stop_sys  = cput.system
69         self.stop_user = cput.user
70         self.stop_rss  = memi.rss
71         self.stop_disk = get_wdb() + get_ldb()
72         t_elapsed_sys  = self.stop_sys - self.start_sys
73         t_elapsed_user = self.stop_user - self.start_user
74         t_elapsed = t_elapsed_sys + t_elapsed_user
75         print >>sys.stderr, blessings.Terminal().yellow('timer:␣
               {}␣took␣{}␣(user:␣{},␣sys:␣{})␣seconds.'.format(self.
               name, t_elapsed, t_elapsed_user, t_elapsed_sys))
76         print >>sys.stderr, blessings.Terminal().yellow('timer:␣
               rss␣=␣{}␣MiB.␣(change:␣{}␣MiB).'.format(self.stop_rss
               /1048576.0, (self.stop_rss-self.start_rss)/1048576.0)
               )
77         print >>sys.stderr, blessings.Terminal().yellow('timer:␣
               disk␣=␣{}␣MiB.␣(change:␣{}␣MiB).'.format(self.
               stop_disk/1048576.0, (self.stop_disk-self.start_disk)
               /1048576.0))
78         print >>sys.stderr
79         stats[self.name+'.Memory'] = self.stop_rss + self.
               stop_disk
80         stats[self.name+'.Runtime'] = t_elapsed
81
82 def main():
83     info('retry3.py␣started.')
84
85     # opening connections to all databases and some necessary
           cleaning and setup.
86     #  - db_s: data source
```

```
87      #  - db_w: in memory working set
88      #  - db_e: on disk leveldb hashtable
89      #
90
91      with timer('Setup'):
92          db_s = sqlite3.connect('cleaned.sqlite3')
93          cu_s = db_s.cursor()
94          #db_w = sqlite3.connect(':memory:')
95          if os.path.exists('batch.sqlite3'):
96              os.remove('batch.sqlite3')
97          db_w = sqlite3.connect('batch.sqlite3')
98          cu_w = db_w.cursor()
99
100         cu_w.execute('''
101 ␣␣␣␣␣␣␣␣␣␣␣␣CREATE␣TABLE␣profile␣(id␣integer␣not␣null,␣cluster␣
        integer␣not␣null,␣block␣integer␣not␣null);
102 ␣␣␣␣␣␣␣␣''')
103         db_w.commit()
104
105         if os.path.exists('batch.leveldb'):
106             info('cleaning␣up␣old␣hashtable...')
107             shutil.rmtree('batch.leveldb')
108
109         megabyte = 1024**2
110         db_e = leveldb.LevelDB('batch.leveldb', \
111                                block_cache_size=128*megabyte, \
112                                write_buffer_size=128*megabyte)
113
114         info('databases␣ready.')
115
116         block_keys = {}
117         block_to_value = {}
118         clusters = collections.defaultdict(set)
119
120
121         ok_chars  = string.ascii_letters + string.digits + '␣'
122
123         sane_str = lambda c: c in ok_chars
124
125         for record in cu_s.execute('SELECT␣id,␣cluster,␣name,␣
                sort_name,␣type,␣area,␣gender,␣comment,␣begin_year,␣
                end_year␣FROM␣artist_sample␣ORDER␣BY␣cluster,␣id␣
                LIMIT␣{};'.format(n_records)):
126             _id = int(record[0])
127             _cl = int(record[1])
128
129             clusters[_cl].add(_id)
130
131             for value in record[2:]:
```

```
132                 if value:
133                     value = unicode(value).strip()
134 
135                     if value:
136                         values = u''.join(filter(sane_str, value
                               )).lower().split()
137 
138                         for value in values:
139 
140                             if value in bad_values:
141                                 continue
142 
143                             block = block_keys.get(value, None)
144 
145                             if block == None:
146                                 block = len(block_keys)
147                                 block_keys[value] = block
148                                 block_to_value[block] = value
149 
150                             cu_w.execute('INSERT INTO profile (
                                   id, cluster, block) VALUES
                                   (?,?,?);', (_id, _cl, block))
151 
152         cu_s.close()
153         db_s.close()
154         del cu_s, db_s
155 
156         cu_w.execute('CREATE INDEX iprofblock ON profile (block)
                ;')
157         db_w.commit()
158 
159 
160     with timer('Graph'):
161         info('creating graph...')
162 
163         # add all edges of the graph by adding them to a
                hashtable.
164         # use some hacks to keep the memory usage low.
165         #
166         packer = struct.Struct('>I').pack
167         unpacker = struct.Struct('>I').unpack
168         def pack(n):
169             return packer(n)
170 
171         def unpack(s):
172             return unpacker(s)[0]
173 
174         def add_edges(block, ids):
175             if len(ids) < 2:
```

```
176                 return 0L
177
178             #print 'adding:', block, ids
179
180             b_block = pack(block)
181
182             ids = list(set(ids))
183             ids.sort()
184             b_ids = map(pack, ids)
185
186             n_edges = 0L
187             wb = leveldb.WriteBatch()
188
189             for edge in itertools.combinations(b_ids, 2):
190                 wb.Put(edge[0] + edge[1] + b_block, '')
191                 n_edges += 1
192
193             db_e.Write(wb)
194
195             return n_edges
196
197         with timer('meta 2-insert'):
198
199             n_edges = 0L
200             last_block = None
201             block_members = []
202
203             for _id, block in cu_w.execute('SELECT id, block
                    FROM profile ORDER BY block;'):
204                 if block != last_block:
205                     n_edges += add_edges(last_block,
                            block_members)
206                     last_block = block
207                     block_members = [_id]
208
209                 else:
210                     block_members.append(_id)
211
212             if block_members:
213                 n_edges += add_edges(last_block, block_members)
214
215             info('edges inserted.', n_edges=n_edges)
216
217
218         with timer('meta 2-counting'):
219
220             info('calculate edge weights...')
221
```

```
222             # scan through all edges and count them to calculate
                    their edge weight.
223             # calculate their average.
224             #
225             cu_w.execute('''
226 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣CREATE␣TABLE␣edges␣(
227 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣n1␣integer␣not␣null,
228 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣n2␣integer␣not␣null,
229 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣weight␣integer
230 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣);
231 ␣␣␣␣␣␣␣␣␣␣␣␣''')
232             db_w.commit()
233
234             b_pre_edge = '\x00'*12
235             b_post_edge = '\xff'*12
236
237             last_edge = b_post_edge
238             weight = 0L
239             n_distinct_edges = 0L
240             total_weight = 0L
241
242             edges = db_e.RangeIter(key_from=b_pre_edge, key_to=
                    b_post_edge, include_value=False)
243             for edge in edges:
244                 if edge.startswith(last_edge):
245                     weight += 1
246                 else:
247                     if weight:
248                         total_weight += weight
249                         n1 = unpack(last_edge[:4])
250                         n2 = unpack(last_edge[4:])
251                         cu_w.execute('INSERT␣INTO␣edges␣(n1,␣n2,
                                ␣weight)␣VALUES␣(?,?,?);', (n1, n2,
                                weight))
252                     weight = 1L
253                     n_distinct_edges += 1
254                     last_edge = edge[:8]
255
256             if weight:
257                 n1 = unpack(last_edge[:4])
258                 n2 = unpack(last_edge[4:])
259                 total_weight += weight
260                 cu_w.execute('INSERT␣INTO␣edges␣(n1,␣n2,␣weight)
                        ␣VALUES␣(?,?,?);', (n1, n2, weight))
261
262             db_w.commit()
263
264             avg_weight = float(total_weight) / n_distinct_edges
265
```

```
266             info('edges counted up.', n_distinct_edges=
                    n_distinct_edges, total_weight=total_weight,
                    avg_weight=avg_weight)
267
268             stats['Distinct Edges.N'] = n_distinct_edges
269             stats['Total Weight.N'] = total_weight
270             stats['Average Weight.N'] = avg_weight
271
272     with timer('Pruning'):
273         info('pruning graph...')
274
275         cu_w.execute('''
276 ..........DELETE FROM edges WHERE weight < ?;
277 ........''', (avg_weight,))
278         db_w.commit()
279
280         info('edges saved.')
281
282
283     with timer('Scoring'):
284         info('scoring metablocking run...')
285
286         ground_truth = map(lambda entities: set(itertools.
                combinations(sorted(entities), 2)), clusters.values()
                )
287         while len(ground_truth) > 1:
288             for _ in xrange(len(ground_truth)/2):
289                 tmp = ground_truth.pop(0)
290                 tmp = tmp.union(ground_truth.pop(0))
291                 ground_truth.append(tmp)
292         ground_truth = ground_truth[0]
293
294         print >>sys.stderr, '# ground_truth:', len(ground_truth)
295         stats['Ground Truth Entity Pairs.N'] = len(ground_truth)
296
297         meta_pairs = set(cu_w.execute('''
298 ..........SELECT n1, n2 FROM edges;
299 ........''').fetchall())
300         stats['Output Entity Pairs.N'] = len(meta_pairs)
301
302         n_cluster_pairs = len(ground_truth)
303         n_meta_pairs = len(meta_pairs)
304
305         # true positive: PAIR found in INPUT and OUTPUT blocks.
306         n_true_positive = len(ground_truth.intersection(
                meta_pairs))
307         # false positive: PAIR found in OUTPUT but not in INPUT.
308         n_false_positive = len(meta_pairs - ground_truth)
309         # true negative: PAIR found neither in INPUT nor OUTPUT.
```

```
310         n_true_negative = '------'
311         # false negative: PAIR found in INPUT but not in OUTPUT.
312         n_false_negative = len(ground_truth - meta_pairs)
313
314         stats['True Positives.N'] = n_true_positive
315         stats['False Positives.N'] = n_false_positive
316         stats['False Negatives.N'] = n_false_negative
317
318
319         # recall and precision:
320         recall = float(n_true_positive) / (n_true_positive +
                n_false_negative)
321         precision = float(n_true_positive) / (n_true_positive +
                n_false_positive)
322
323         # f-measure
324         f_measure = 2 * precision * recall / (precision + recall
                )
325
326         stats['Recall.Recall'] = recall
327         stats['Precision.Precision'] = precision
328         stats['F-Measure.F-Measure'] = f_measure
329
330     with timer('post 1-paperstats'):
331         # PC
332         cluster_pairs_sharing_block = set(cu_w.execute('''
333 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣SELECT␣DISTINCT␣p1.id,␣p2.id␣FROM␣profile␣AS␣p1,␣
        profile␣AS␣p2
334 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣WHERE␣p1.cluster␣=␣p2.cluster␣AND
335 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣p1.block␣=␣p2.block␣AND
336 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣p1.id␣<␣p2.id;
337 ␣␣␣␣␣␣␣␣''').fetchall())
338
339         Din = len(cluster_pairs_sharing_block)
340         Dout = len(meta_pairs.intersection(
                cluster_pairs_sharing_block))
341         PC = float(Dout) / Din
342
343         # RR
344         n_edges_remaining = cu_w.execute('SELECT␣count(*)␣FROM␣
                edges;').fetchone()[0]
345
346         #RR_complete = 1.0 - float(n_edges_remaining) / (n_edges
                + n_edges_skipped)
347         #RR_cheating = 1.0 - float(n_edges_remaining) / n_edges
348         RR = 1.0 - float(n_edges_remaining) / n_edges
349
350         # PQ
351         PQ = float(n_true_positive) / n_edges
```

```
352
353             stats['PC'] = PC
354             stats['RR'] = RR
355             stats['PQ'] = PQ
356
357             cu_w.close()
358             db_w.close()
359
360             info('batch.py ended.')
361
362 main()
363
364 time_stopped = time.clock()
365 stats['Overall Runtime.Runtime'] = time_stopped - time_started
366
367 print 'BATCH', stats
```

A.3 REVIDX

```
from collections import defaultdict as hashtable
from pprint import pprint
from blessings import Terminal as T
from functools import partial
from itertools import combinations, chain
from sqlite3 import connect
from string import ascii_letters, digits
from time import clock
from psutil import Process as P; P = P()
from os.path import exists
from shutil import rmtree
from leveldb import LevelDB, WriteBatch
from operator import itemgetter
from multiprocessing import Pool
from sys import stderr as err
from sys import argv
from sh import du

# config

if len(argv) == 2:
    n_records = int(argv[-1])
else:
    n_records = 500

print >>err, '--- STARTING: n =', n_records, '---'

bad_values = set(list(ascii_letters + digits))

time_started = clock()
stats = {'Records.N': n_records,
         }

# helpers

def _merge(a):
    if len(a) == 2:
        return a[0].union(a[1])
    else:
        return a[0]

class timer(object):
    def __init__(self, name='<block>'):
        self.name = name
        self.start_sys  = 0.0
        self.start_user = 0.0
        self.start_rss = 0L
```

```
48          self.start_disk = 0L
49      def __enter__(self):
50          cput = P.cpu_times()
51          memi = P.memory_info_ex()
52          self.start_sys  = cput.system
53          self.start_user = cput.user
54          self.start_rss  = memi.rss
55          self.start_disk = 0L
56      def __exit__(self, *args):
57          cput = P.cpu_times()
58          memi = P.memory_info_ex()
59          self.stop_sys  = cput.system
60          self.stop_user = cput.user
61          self.stop_rss  = memi.rss
62          self.stop_disk = 0L
63          t_elapsed_sys  = self.stop_sys - self.start_sys
64          t_elapsed_user = self.stop_user - self.start_user
65          t_elapsed = t_elapsed_sys + t_elapsed_user
66          print >>err, T().yellow('timer: {} took {} (user: {}, 
                sys: {}) seconds.').format(self.name, t_elapsed,
                t_elapsed_user, t_elapsed_sys))
67          print >>err, T().yellow('timer: rss = {} MiB. (change: 
                {} MiB).').format(self.stop_rss/1048576.0, (self.
                stop_rss-self.start_rss)/1048576.0))
68          print >>err, T().yellow('timer: disk = {} MiB. (change: 
                {} MiB.').format(self.stop_disk/1048576.0, (self.
                stop_disk-self.start_disk)/1048576.0))
69          print >>err
70          stats[self.name+'.Memory'] = self.stop_rss + self.
                stop_disk
71          stats[self.name+'.Runtime'] = t_elapsed
72
73  def all_combinations(entities):
74      return combinations(entities, 2)
75
76  c = lambda v: T().bold_bright_black(str(v))
77  b = lambda v: T().bold_bright_red(str(v))
78  e = lambda v: T().underline_white(str(v))
79
80  def show(d, f1, f2):
81      for k, s in d.items():
82          k = str(k)
83          print ' +', f1(k), '.'*(20-len(k)), '[', ' '.join(map(f2
                , sorted(s))), ']'
84      return
85
86  # load the table into memory
87
```

```
88  print >>err, c('# step 0: reading the table into memory and
        encoding attributes')
89  print >>err, c('#          through numbers to increase
        performance')
90  print >>err, c('#          (this is not part of metablocking)')
91  print >>err
92
93  with timer('Setup'):
94      block_keys = {}
95      block_to_value = {}
96      table = hashtable(set)
97      clusters = hashtable(set)
98
99      db = connect('cleaned.sqlite3')
100     cu = db.cursor()
101
102     ok_chars  = ascii_letters + digits + ' '
103
104     sane_str = lambda c: c in ok_chars
105
106     for record in cu.execute('SELECT id, cluster, name,
            sort_name, type, area, gender, comment, begin_year,
            end_year FROM artist_sample ORDER BY cluster, id LIMIT
            {};'.format(n_records)):
107         _id = int(record[0])
108         _cl = int(record[1])
109
110         clusters[_cl].add(_id)
111
112         for value in record[2:]:
113             if value:
114                 value = unicode(value).strip()
115
116                 if value:
117                     values = u''.join(filter(sane_str, value)).
                            lower().split()
118
119                     for value in values:
120
121                         if value in bad_values:
122                             continue
123
124                         block = block_keys.get(value, None)
125
126                         if block == None:
127                             block = len(block_keys)
128                             block_keys[value] = block
129                             block_to_value[block] = value
130
```

```
131                             table[_id].add(block)
132
133         cu.close()
134         db.close()
135         del cu, db
136
137
138         print >>err, c('# step 1: transform the table into a
                collection of blocks')
139         print >>err, c('#          (this is not part of metablocking)
                ')
140         print >>err
141
142         def extract_blocks(table):
143             blocks = hashtable(set)
144             for entity, attributes in table.items(): # do entities
                    need to be sorted in block?
145                 for attribute in attributes:
146                     blocks[attribute].add(entity)
147             for block, entities in blocks.items(): # yes they do!!!
148                 entities = list(entities)
149                 entities.sort()
150                 blocks[block] = entities
151             return blocks
152
153         blocks = extract_blocks(table)
154
155         del table
156
157     print >>err, c('# meta 1: create the reverse index from all
            blocks')
158     print >>err, c('#          (this is where metablocking starts)')
159     print >>err, c('#')
160     print >>err, c('#          the blocks in the reverse index have
            to be')
161     print >>err, c('#          in the same order as we process the
            blocks')
162     print >>err, c('#          for the sum calculation to work.')
163     print >>err
164
165     def build_rev_idx(blocks):
166         rev_idx = hashtable(list) # must be a hashtable of SORTED
                lists
167         for block, entities in sorted(blocks.items()): # add blocks
                in SORTED order.
168             for entity in entities:
169                 rev_idx[entity].append(block)
170         return rev_idx
171
```

```
172 with timer('RevIdx'):
173     rev_idx = build_rev_idx(blocks)
174
175 print >>err, c('# meta 2: calculate the "total_weight", "
        n_distinct_edges",')
176 print >>err, c('#          and "avg_weight" by iterating through
        all blocks')
177 print >>err, c('#          in sorted order.')
178 print >>err
179
180 def get_weight(block, e1, e2):
181     blocks_e1 = rev_idx[e1]
182     blocks_e2 = rev_idx[e2]
183
184     common_blocks = 0L
185     first_common = False
186     for b1 in blocks_e1:
187         for b2 in blocks_e2:
188             if b1 == b2:
189                 common_blocks += 1
190
191                 if not first_common:
192                     first_common = True
193                     if b1 != block:
194                         return -1 # error code
195                     else:
196                         pass
197                 else:
198                     pass
199             else:
200                 pass
201
202     return common_blocks
203
204 with timer('Graph'):
205     print >>err, 'CALCULATING total_weight, n_distinct_edges,
            average_weight'
206     total_weight = 0L
207     n_distinct_edges = 0L
208
209     for block, entities in sorted(blocks.items()):
210         for e1, e2 in all_combinations(blocks[block]):
211             weight = get_weight(block, e1, e2)
212             if weight != -1:
213                 total_weight += weight
214                 n_distinct_edges += 1
215
216     average_weight = float(total_weight) / n_distinct_edges
217     stats['Total Weight.N'] = total_weight
```

```
218     stats['Distinct Edges.N'] = n_distinct_edges
219     stats['Average Weight.N'] = average_weight
220
221 print >>err, c('# meta 3: re-iterate through all blocks and
        apply the pruning')
222 print >>err, c('#          criterion. create the output blocks.')
223 print >>err
224
225
226 with timer('Pruning'):
227     # print 'APPLY PRUNING CRITERION AND OUTPUT NEW BLOCKS'
228     new_blocks = hashtable(set)
229
230     for block, entities in sorted(blocks.items()):
231         for e1, e2 in all_combinations(blocks[block]):
232             weight = get_weight(block, e1, e2)
233             if weight < average_weight:
234                 pass
235             else:
236                 new_blocks[block].add((e1, e2))
237
238 print >>err, c('# post 1: measure stuff')
239 print >>err, c('#          (this is not part of metablocking
        anymore.)')
240 print >>err
241
242 with timer('Scoring'):
243
244     ground_truth = set()
245
246     n_true_positive = 0L
247
248     for cluster, entities in clusters.items():
249         if len(entities) > 1:
250             ground_truth = ground_truth.union(sorted(
                    all_combinations(entities)))
251
252     stats['Ground Truth Entity Pairs.N'] = len(ground_truth)
253
254     all_comparisons = list(new_blocks.values())
255     while len(all_comparisons) > 1:
256         for _ in xrange(len(all_comparisons)/2):
257             tmp = all_comparisons.pop(0)
258             tmp = tmp.union(all_comparisons.pop(0))
259             all_comparisons.append(tmp)
260     all_comparisons = all_comparisons[0]
261     stats['Output Entity Pairs.N'] = len(all_comparisons)
262
```

```
263         n_true_positive += len(ground_truth.intersection(
                all_comparisons))
264         n_false_positive = len(all_comparisons - ground_truth)
265         n_false_negative = len(ground_truth - all_comparisons)
266 
267         stats['True Positives.N'] = n_true_positive
268         stats['False Positives.N'] = n_false_positive
269         stats['False Negatives.N'] = n_false_negative
270 
271         recall = float(n_true_positive) / (n_true_positive +
                n_false_negative)
272         precision = float(n_true_positive) / (n_true_positive +
                n_false_positive)
273         f_measure = 2 * precision * recall / (precision + recall)
274 
275         stats['Recall.Recall'] = recall
276         stats['Precision.Precision'] = precision
277         stats['F-Measure.F-Measure'] = f_measure
278 
279 time_stopped = clock()
280 stats['Overall Runtime.Runtime'] = time_stopped - time_started
281 
282 print 'REVIDX', stats
```

A.4 Description of Dataset

```
1  import sqlite3
2  import collections
3  import string
4  import tabulate
5
6  # helpers
7
8  ok_chars  = string.ascii_letters + string.digits + ' '
9  sane_str = lambda c: c in ok_chars
10 bad_values = set(string.ascii_letters + string.digits) # single
       letters/digits
11
12 query_string = '''
13 ␣␣␣␣SELECT␣id,
14 ␣␣␣␣␣␣␣␣␣␣␣␣cluster,
15 ␣␣␣␣␣␣␣␣␣␣␣␣name,
16 ␣␣␣␣␣␣␣␣␣␣␣␣sort_name,
17 ␣␣␣␣␣␣␣␣␣␣␣␣type,
18 ␣␣␣␣␣␣␣␣␣␣␣␣area,
19 ␣␣␣␣␣␣␣␣␣␣␣␣gender,
20 ␣␣␣␣␣␣␣␣␣␣␣␣comment,
21 ␣␣␣␣␣␣␣␣␣␣␣␣begin_year,
22 ␣␣␣␣␣␣␣␣␣␣␣␣end_year
23 ␣␣␣␣␣␣FROM␣artist_sample
24 ␣␣ORDER␣BY␣cluster,␣id;
25 '''
26
27 def extract_stats(ht):
28     n_ht = len(ht)
29     s_min =  999999999
30     s_max = -999999999
31     s_sum = 0L
32     for k, s in ht.items():
33         s_min = min(s_min, len(s))
34         s_max = max(s_max, len(s))
35         s_sum += len(s)
36     s_avg = float(s_sum) / n_ht
37
38     return n_ht, s_min, s_max, s_avg
39
40 step_size =  [1000, 2000, 5000, 10000, 20000, 30000]
41 stop_size = max(step_size)
42
43 # connect to database
44 db = sqlite3.connect('cleaned.sqlite3')
45 cu = db.cursor()
46
```

```
47 # value->bid and bid->value can stay the same across subsets
48 value_to_bid = {}
49 bid_to_value = {}
50
51 # output statistics / helpers
52 stats = collections.defaultdict(list)
53
54 def dpt(k, y):
55     stats[k].append(y)
56
57 # associations... kept globally for incremental approach.
58 entity2block = collections.defaultdict(set)
59 block2entity = collections.defaultdict(set)
60 entity2clust = collections.defaultdict(set)
61 clust2entity = collections.defaultdict(set)
62 block2clustr = collections.defaultdict(set)
63 clustr2block = collections.defaultdict(set)
64
65 for record in cu.execute(query_string):
66     _id = int(record[0])
67     _cl = int(record[1])
68
69     # add cluster-entity associations
70     clust2entity[_cl].add(_id)
71     entity2clust[_id].add(_cl)
72
73     for value in record[2:]:
74         if value:
75             # value is not none
76
77             value = unicode(value).strip()
78
79             if value:
80                 # value is not an empty string
81
82                 values = u''.join(filter(sane_str, value)).lower
                        ().split()
83
84                 for value in values:
85
86                     if value in bad_values:
87                         continue
88
89                     bid = value_to_bid.get(value, None)
90
91                     if bid == None:
92                         bid = len(value_to_bid)
93                         value_to_bid[value] = bid
94                         bid_to_value[bid] = value
```

```
95
96                      # add entity-block, and cluster-block
                            associations
97                      entity2block[_id].add(bid)
98                      block2entity[bid].add(_id)
99                      clustr2block[_cl].add(bid)
100                     block2clustr[bid].add(_cl)
101
102     n_records = len(entity2block)
103
104     if n_records in step_size:
105         # calculate statistics
106         print 'calculating statistics... n_records=', n_records
107
108         #EC = extract_stats(entity2clust)
109         CE = extract_stats(clust2entity)
110         EB = extract_stats(entity2block)
111         BE = extract_stats(block2entity)
112         #CB = extract_stats(clustr2block)
113         #BC = extract_stats(block2clustr)
114
115         # 1. table of input blocks
116         # ------------------------
117
118         #  - n_records
119         dpt('n-records', n_records)
120
121         #  - n_blocks
122         dpt('n-blocks', BE[0])
123
124         #  - n_clusters
125         dpt('n-clusters', CE[0])
126
127         #  - block size: min, max, avg
128         dpt('blocksize-min', BE[1])
129         dpt('blocksize-max', BE[2])
130         dpt('blocksize-avg', "{0:.2f}".format(BE[3]))
131
132         #  - n_sebs (single entity blocks)
133         dpt('n-sebs', len(filter(lambda (k, v): len(v)==1,
                block2entity.items())))
134
135         #  - bpe: min, max, avg (blocks per entity)
136         dpt('bpe-min', EB[1])
137         dpt('bpe-max', EB[2])
138         dpt('bpe-avg', "{0:.2f}".format(EB[3]))
139
140     if n_records == stop_size:
141         break
```

```
142
143 cu.close()
144 db.close()
145
146 # output data for rendering
147 n_records = stats['n-records']
148
149 for k in stats:
150     with file('report/dataset-stats/'+k, 'w') as fh:
151         for i, v in enumerate(stats[k]):
152             print >>fh, n_records[i], v
153
154 table = []
155
156 headers = [
157     'n-records',
158     'n-clusters',
159     'n-blocks',
160     'n-sebs',
161     'blocksize-min',
162     'blocksize-max',
163     'blocksize-avg',
164     'bpe-min',
165     'bpe-max',
166     'bpe-avg',
167 ]
168 print_header = [
169     'Records',
170     'Clusters',
171     'Blocks',
172     '1-E. Blocks.',
173     'Min.',
174     'Max.',
175     'Avg.',
176     'Min.',
177     'Max.',
178     'Avg.',
179 ]
180
181 for i in xrange(len(n_records)):
182     row = []
183     for k in headers:
184         row.append(stats[k][i])
185     table.append(row)
186
187 with file('report/dataset-table.tex', 'w') as fh:
188     print >>fh, tabulate.tabulate(table, headers=print_header,
        tablefmt='latex')
```