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.

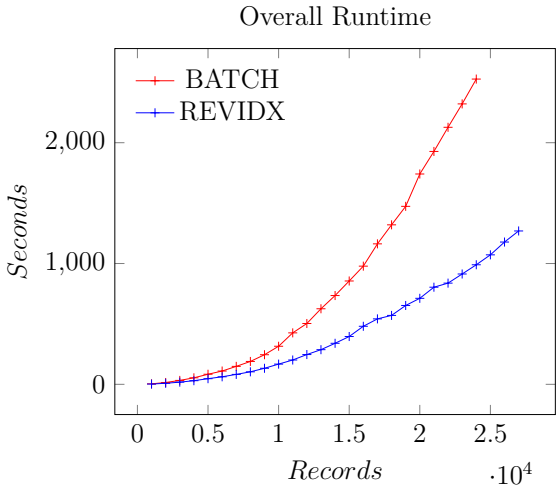
2. The decreasing average number of blocks per entity hints at a large number of sparse records. Given the number of fields in the dataset, we expect a lower bound of 6 blocks per entity for complete records.

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

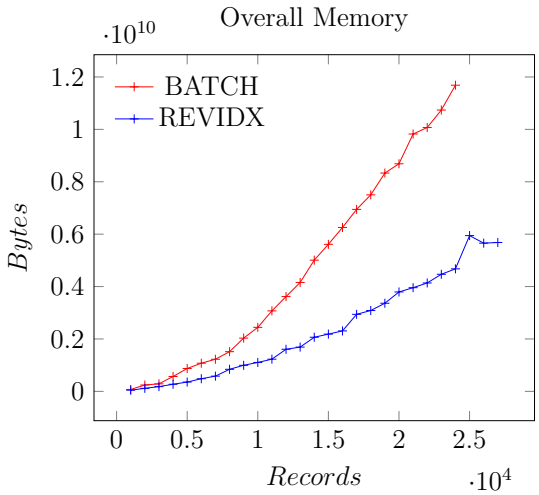
3.2.1 Comparison of Runtime

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.



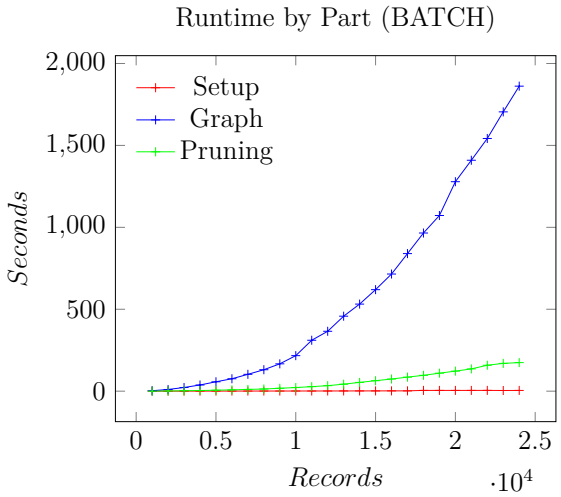
3.2.2 Comparison of Memory Usage

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 $\bar{B}_{output}$.

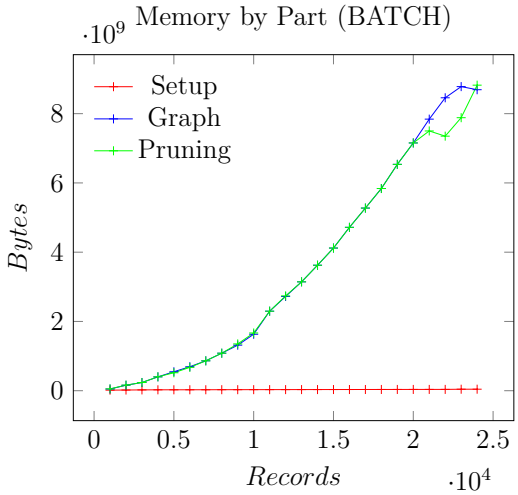


3.2.3 Detailed Analysis of BATCH

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*).

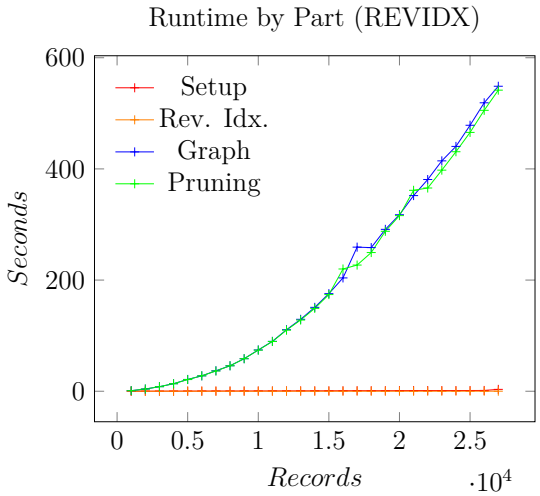


The memory consumption of BATCH is also dominated by the construction of the *Graph*. The later *Pruning* stage, only consumes marginally more memory for $\bar{B}_{output}$. The small reduction in memory consumption above 25 K records is an artefact of the implementation of $\bar{E}$.

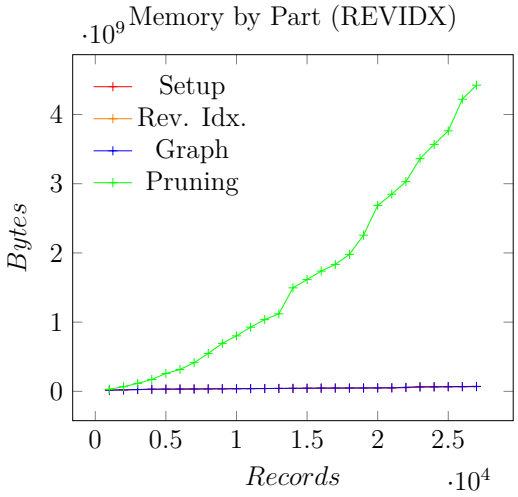


3.2.4 Detailed Analysis of REVIDX

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.

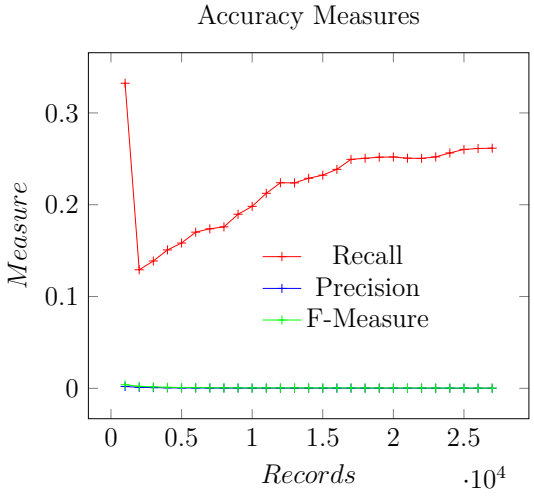


3.3 Accuracy of the method

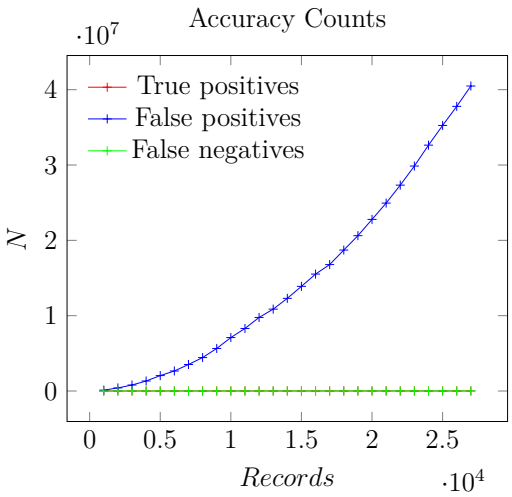
The quality of the output generated by meta-blocking was measured using *precision*, *recall*, and *f-measure*, by comparing $\bar{B}_{output}$ against a list of entity pairs generated using the *cluster* attribute of the dataset.

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	佐藤利奈	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+'.user']  = t_elapsed_user
80         #stats[self.name+'.sys']   = t_elapsed_sys
81         #stats[self.name+'.total'] = t_elapsed
82         #stats[self.name+'.rss'] = self.stop_rss
83         #stats[self.name+'.disk'] = self.stop_disk
84         stats[self.name+'.Memory'] = self.stop_rss + self.
               stop_disk
85         stats[self.name+'.Runtime'] = t_elapsed
86
87 def main():
```

```
88      info('retry3.py started.')
89
90      # opening connections to all databases and some necessary
            cleaning and setup.
91      #  - db_s: data source
92      #  - db_w: in memory working set
93      #  - db_e: on disk leveldb hashtable
94      #
95
96      with timer('Setup'):
97          db_s = sqlite3.connect('cleaned.sqlite3')
98          cu_s = db_s.cursor()
99          #db_w = sqlite3.connect(':memory:')
100         if os.path.exists('batch.sqlite3'):
101             os.remove('batch.sqlite3')
102         db_w = sqlite3.connect('batch.sqlite3')
103         cu_w = db_w.cursor()
104
105         cu_w.execute('''
106 ␣␣␣␣␣␣␣␣␣␣␣␣␣CREATE␣TABLE␣profile␣(id␣integer␣not␣null,␣cluster␣
        integer␣not␣null,␣block␣integer␣not␣null);
107 ␣␣␣␣␣␣␣␣''')
108         db_w.commit()
109
110         #cu_w.execute('''
111         #    CREATE TABLE cluster (id INT, cluster INT)
112         #''')
113         #db_w.commit()
114
115         if os.path.exists('batch.leveldb'):
116             info('cleaning up old hashtable...')
117             shutil.rmtree('batch.leveldb')
118
119         megabyte = 1024**2
120         db_e = leveldb.LevelDB('batch.leveldb', \
121                                block_cache_size=128*megabyte, \
122                                write_buffer_size=128*megabyte)
123
124         info('databases ready.')
125
126         #info('reading blocks...')
127
128         # reading all data from the source database,
                preprocessing, and encoding.
129         # this results in a table of (profile, block)
                associations.
130         # Also extract the ground truth (profile, cluster) into
                another table for later.
131         #
```

```
132         #block_counts = collections.Counter()
133         #block_ids    = dict()
134
135         #not_none  = lambda v: v
136         #clean_str = lambda v: unicode(v).strip()
137         #not_empty = lambda v: len(v)
138
139         #ok_chars  = string.ascii_letters + string.digits + '- '
140         #sane_str  = lambda c: c in ok_chars
141
142         #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 limit
                {};'.format(n_records)):
143         #    _id = int(record[0])
144         #    _cl = int(record[1])
145
146         #    tokens = filter(not_none, record[2:])
147         #    tokens = map(clean_str, tokens)
148         #    tokens = filter(not_empty, tokens)
149         #    tokens = u' '.join(tokens)
150         #    tokens = u''.join(filter(sane_str, tokens)).lower()
151         #    tokens = tokens.split()
152         #    tokens = set(tokens)
153
154         #    for token in tokens:
155         #        block_counts[token] += 1
156         #        block_id = block_ids.get(token, None)
157         #        if block_id == None:
158         #            block_id = len(block_ids)
159         #            block_ids[token] = block_id
160         #        cu_w.execute('INSERT INTO profile (id, cluster,
              block) VALUES (?, ?, ?);', (_id, _cl, block_id))
161
162         #db_w.commit()
163
164         #cu_w.execute('CREATE INDEX iprofblock ON profile (block
              );')
165         #db_w.commit()
166
167         #with codecs.open('blocks.txt', 'w', 'utf-8') as fh:
168         #    for value in block_ids:
169         #        print >>fh, value, block_counts[value]
170
171         #info('blocks extracted.')
172
173         block_keys = {}
174         block_to_value = {}
175         clusters = collections.defaultdict(set)
```

```
176
177
178         ok_chars  = string.ascii_letters + string.digits + ' '
179
180         sane_str = lambda c: c in ok_chars
181
182         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)):
183             _id = int(record[0])
184             _cl = int(record[1])
185
186             clusters[_cl].add(_id)
187
188             for value in record[2:]:
189                 if value:
190                     value = unicode(value).strip()
191
192                     if value:
193                         values = u''.join(filter(sane_str, value
                                )).lower().split()
194
195                         for value in values:
196
197                             if value in bad_values:
198                                 continue
199
200                             block = block_keys.get(value, None)
201
202                             if block == None:
203                                 block = len(block_keys)
204                                 block_keys[value] = block
205                                 block_to_value[block] = value
206
207                             cu_w.execute('INSERT INTO profile (
                                    id, cluster, block) VALUES
                                    (?,?,?);', (_id, _cl, block))
208
209         cu_s.close()
210         db_s.close()
211         del cu_s, db_s
212
213         cu_w.execute('CREATE INDEX iprofblock ON profile (block)
                ;')
214         db_w.commit()
215
216         #n_associations = cu_w.execute('SELECT count(*) FROM
                profile;').fetchone()[0]
```

```
217         #n_blocks = cu_w.execute('SELECT count(DISTINCT block)
                FROM profile;').fetchone()[0]
218         #n_avg_assoc_per_block = float(n_associations) /
                n_blocks
219         #info('number of associations retreived.',
                n_associations=n_associations, n_blocks=n_blocks,
                n_avg_assoc_per_block=n_avg_assoc_per_block)
220
221         #info('removing bad blocks...')
222
223         ## some blocks contain too many profiles to be
                computationally feasable,
224         ## hence they need to be skipped.
225         ## furthermore, skip all blocks with just one profile.
226         ##
227         def fak(n):
228             return reduce(lambda x,y: x*y, xrange(1, n+1), 1)
229
230         def combs(n, k):
231             return fak(n)/fak(k)/fak(n-k)
232
233         #n_min_profiles = 2
234         #n_max_profiles = 1500 # why?
235
236         # first, collect some statistics.
237         #n_rare_profile_blocks = len(cu_w.execute('SELECT count(
                block) FROM profile GROUP BY block HAVING count(id) <
                 {};'.format(n_min_profiles)).fetchall())
238         #n_frequent_profile_blocks = len(cu_w.execute('SELECT
                count(block) FROM profile GROUP BY block HAVING count
                (id) > {};'.format(n_max_profiles)).fetchall())
239
240         #info('collected bad blocks statistics',
                n_rare_profile_blocks=n_rare_profile_blocks,
                n_frequent_profile_blocks=n_frequent_profile_blocks)
241
242         #cu_w.execute('''
243         #    CREATE TABLE badblocks
244         #        AS SELECT block, count(id) AS count FROM
                profile GROUP BY block
245         #            HAVING count(id) < {} OR count(id) > {};
246         #'''.format(n_min_profiles, n_max_profiles))
247         #db_w.commit()
248
249         #cu_w.execute('CREATE INDEX ibb ON badblocks (block);')
250         #db_w.commit()
251
252         #cu_w.execute('CREATE INDEX ibc ON badblocks (count);')
253         #db_w.commit()
```

```
254
255         #n_edges_skipped = 0L
256         #n_bad_blocks = 0L
257
258         #ids2value = dict(map(lambda (a, b): (b, a), block_ids.
                items()))
259
260         #def mem(cnt):
261         #    cnt *= (cnt - 1)
262         #    cnt *= 8.0
263         #    if cnt < 1024: return str(int(cnt)) + 'B'
264         #    cnt /= 1024
265         #    if cnt < 1024: return str(int(cnt)) + 'KB'
266         #    cnt /= 1024
267         #    if cnt < 1024: return str(int(cnt)) + 'MB'
268         #    cnt /= 1024
269         #    if cnt < 1024: return str(int(cnt)) + 'GB'
270         #    cnt /= 1024
271         #    return str(int(cnt)) + 'TB'
272
273
274         #with codecs.open('badblocks.txt', 'w', 'utf-8') as fh:
275         #    for block, count in cu_w.execute('SELECT block,
                count FROM badblocks ORDER BY count DESC;'):
276         #        n_bad_blocks += 1
277         #        n_edges_skipped += combs(count, 2)
278         #    pprint >>fh, '\t'.join([str(block), str(count), mem
                (count), ids2value[block]])
279
280         #cu_w.execute('''
281         #    CREATE TABLE clean_profile
282         #        AS SELECT p.id, p.block FROM profile AS p
283         #            WHERE NOT EXISTS(SELECT * FROM badblocks
                WHERE block=p.block);
284         #''')
285         #db_w.commit()
286
287         #n_clean_associations = cu_w.execute('SELECT count(*)
                FROM clean_profile;').fetchone()[0]
288         #n_clean_blocks = cu_w.execute('SELECT count(DISTINCT
                block) FROM clean_profile;').fetchone()[0]
289         #n_avg_clean_assoc_per_block = float(
                n_clean_associations) / n_clean_blocks
290
291         #info('bad blocks removed.', n_bad_blocks=n_bad_blocks,
                n_edges_skipped=n_edges_skipped, \
292         #       n_clean_associations=n_clean_associations,
                n_clean_blocks=n_clean_blocks, \
```

```
293             #       n_avg_clean_assoc_per_block=
                    n_avg_clean_assoc_per_block)
294
295         with timer('Graph'):
296             info('creating graph...')
297
298             # add all edges of the graph by adding them to a
                    hashtable.
299             # use some hacks to keep the memory usage low.
300             #
301             packer = struct.Struct('>I').pack
302             unpacker = struct.Struct('>I').unpack
303             def pack(n):
304                 return packer(n)
305
306             def unpack(s):
307                 return unpacker(s)[0]
308
309             def add_edges(block, ids):
310                 if len(ids) < 2:
311                     return 0L
312
313                 #print 'adding:', block, ids
314
315                 b_block = pack(block)
316
317                 ids = list(set(ids))
318                 ids.sort()
319                 b_ids = map(pack, ids)
320
321                 n_edges = 0L
322                 wb = leveldb.WriteBatch()
323
324                 for edge in itertools.combinations(b_ids, 2):
325                     wb.Put(edge[0] + edge[1] + b_block, '')
326                     n_edges += 1
327
328                 db_e.Write(wb)
329
330                 return n_edges
331
332             with timer('meta 2-insert'):
333
334                 n_edges = 0L
335                 last_block = None
336                 block_members = []
337
338                 for _id, block in cu_w.execute('SELECT id, block
                        FROM profile ORDER BY block;'):
```

```
339                     if block != last_block:
340                         n_edges += add_edges(last_block,
                                block_members)
341                         last_block = block
342                         block_members = [_id]
343 
344                     else:
345                         block_members.append(_id)
346 
347                 if block_members:
348                     n_edges += add_edges(last_block, block_members)
349 
350                 info('edges inserted.', n_edges=n_edges)
351 
352             #cu_w.execute('DROP TABLE clean_profile;')
353             #db_w.commit()
354 
355             #info('temporary table "clean_profile" dropped.')
356 
357             with timer('meta 2-counting'):
358 
359                 info('calculate edge weights...')
360 
361                 # scan through all edges and count them to calculate
                        their edge weight.
362                 # calculate their average.
363                 #
364                 cu_w.execute('''
365 ................CREATE TABLE edges (
366 ....................n1 integer not null,
367 ....................n2 integer not null,
368 ....................weight integer
369 ................);
370 ............''')
371                 db_w.commit()
372 
373                 b_pre_edge = '\x00'*12
374                 b_post_edge = '\xff'*12
375 
376                 last_edge = b_post_edge
377                 weight = 0L
378                 n_distinct_edges = 0L
379                 total_weight = 0L
380 
381                 edges = db_e.RangeIter(key_from=b_pre_edge, key_to=
                        b_post_edge, include_value=False)
382                 for edge in edges:
383                     if edge.startswith(last_edge):
384                         weight += 1
```

```
385                     else:
386                         if weight:
387                             total_weight += weight
388                             n1 = unpack(last_edge[:4])
389                             n2 = unpack(last_edge[4:])
390                             cu_w.execute('INSERT INTO edges (n1, n2,
                                    weight) VALUES (?,?,?);', (n1, n2,
                                    weight))
391                         weight = 1L
392                         n_distinct_edges += 1
393                         last_edge = edge[:8]
394
395                 if weight:
396                     n1 = unpack(last_edge[:4])
397                     n2 = unpack(last_edge[4:])
398                     total_weight += weight
399                     cu_w.execute('INSERT INTO edges (n1, n2, weight)
                            VALUES (?,?,?);', (n1, n2, weight))
400
401                 db_w.commit()
402
403                 avg_weight = float(total_weight) / n_distinct_edges
404
405                 info('edges counted up.', n_distinct_edges=
                        n_distinct_edges, total_weight=total_weight,
                        avg_weight=avg_weight)
406
407                 stats['Distinct Edges.N'] = n_distinct_edges
408                 stats['Total Weight.N'] = total_weight
409                 stats['Average Weight.N'] = avg_weight
410
411         with timer('Pruning'):
412             info('pruning graph...')
413
414             cu_w.execute('''
415 ⊔⊔⊔⊔⊔⊔⊔⊔⊔⊔⊔⊔DELETE FROM edges WHERE weight < ?;
416 ⊔⊔⊔⊔⊔⊔⊔⊔''', (avg_weight,))
417             db_w.commit()
418
419             #n_edges_remaining = cu_w.execute('SELECT count(*) FROM
                    edges;').fetchone()[0]
420
421             #info('graph pruned.', n_edges_remaining=
                    n_edges_remaining)
422
423             #with codecs.open('edges.txt', 'w', 'utf-8') as fh:
424             #    for p1, p2, w in cu_w.execute('SELECT n1, n2,
                    weight FROM edges ORDER BY n1, n2;'):
425             #        print >>fh, p1, p2, w
```

```
426
427             info('edges saved.')
428
429
430         with timer('Scoring'):
431             info('scoring metablocking run...')
432
433             # calculate the f-measure for the output blocks.
434             # calculate the accuracy and efficiency of the current
                    metablocking run.
435             # 1. PC "pair completeness": Dout / Din
436             #     with D.. = number duplicates that share at least
                    one block.
437             #
438             # 2. RR "reduction ratio": 1 - (Cout / Cin)
439             #     with C.. = number of comparisons
440             #
441             # 3. PQ "pairs quality": Dout / Cout
442
443             #cluster_pairs = set(cu_w.execute('''
444             #    SELECT DISTINCT p1.id, p2.id FROM profile AS p1,
                    profile AS p2
445             #        WHERE p1.cluster = p2.cluster AND
446             #              p1.id < p2.id;
447             #''').fetchall())
448
449             ground_truth = map(lambda entities: set(itertools.
                    combinations(sorted(entities), 2)), clusters.values()
                    )
450             while len(ground_truth) > 1:
451                 for _ in xrange(len(ground_truth)/2):
452                     tmp = ground_truth.pop(0)
453                     tmp = tmp.union(ground_truth.pop(0))
454                     ground_truth.append(tmp)
455             ground_truth = ground_truth[0]
456
457             print >>sys.stderr, '# ground_truth:', len(ground_truth)
458             stats['Ground Truth Entity Pairs.N'] = len(ground_truth)
459
460             meta_pairs = set(cu_w.execute('''
461 ␣␣␣␣␣␣␣␣␣␣␣␣SELECT n1, n2 FROM edges;
462 ␣␣␣␣␣␣␣␣''').fetchall())
463             stats['Output Entity Pairs.N'] = len(meta_pairs)
464
465             n_cluster_pairs = len(ground_truth)
466             n_meta_pairs = len(meta_pairs)
467
468             #with codecs.open('cluster-pairs.txt', 'w', 'utf-8') as
                    fh:
```

```
469             #    for p1, p2 in sorted(cluster_pairs):
470             #        print >>fh, p1, p2
471
472             #with codecs.open('metablocking-pairs.txt', 'w', 'utf
                    -8') as fh:
473             #    for p1, p2 in sorted(meta_pairs):
474             #        print >>fh, p1, p2
475
476             # true positive: PAIR found in INPUT and OUTPUT blocks.
477             n_true_positive = len(ground_truth.intersection(
                    meta_pairs))
478             # false positive: PAIR found in OUTPUT but not in INPUT.
479             n_false_positive = len(meta_pairs - ground_truth)
480             # true negative: PAIR found neither in INPUT nor OUTPUT.
481             n_true_negative = '------'
482             # false negative: PAIR found in INPUT but not in OUTPUT.
483             n_false_negative = len(ground_truth - meta_pairs)
484
485             stats['True Positives.N'] = n_true_positive
486             stats['False Positives.N'] = n_false_positive
487             stats['False Negatives.N'] = n_false_negative
488
489
490             # recall and precision:
491             recall = float(n_true_positive) / (n_true_positive +
                    n_false_negative)
492             precision = float(n_true_positive) / (n_true_positive +
                    n_false_positive)
493
494             # f-measure
495             f_measure = 2 * precision * recall / (precision + recall
                    )
496
497             stats['Recall.Recall'] = recall
498             stats['Precision.Precision'] = precision
499             stats['F-Measure.F-Measure'] = f_measure
500
501         with timer('post 1-paperstats'):
502             # PC
503             cluster_pairs_sharing_block = set(cu_w.execute('''
504 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣SELECT␣DISTINCT␣p1.id,␣p2.id␣FROM␣profile␣AS␣p1,␣
        profile␣AS␣p2
505 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣WHERE␣p1.cluster␣=␣p2.cluster␣AND
506 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣p1.block␣=␣p2.block␣AND
507 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣p1.id␣<␣p2.id;
508 ␣␣␣␣␣␣␣␣''').fetchall())
509
510             Din = len(cluster_pairs_sharing_block)
```

```
511             Dout = len(meta_pairs.intersection(
                    cluster_pairs_sharing_block))
512             PC = float(Dout) / Din
513
514             # RR
515             n_edges_remaining = cu_w.execute('SELECT count(*) FROM
                    edges;').fetchone()[0]
516
517             #RR_complete = 1.0 - float(n_edges_remaining) / (n_edges
                    + n_edges_skipped)
518             #RR_cheating = 1.0 - float(n_edges_remaining) / n_edges
519             RR = 1.0 - float(n_edges_remaining) / n_edges
520
521             # PQ
522             PQ = float(n_true_positive) / n_edges
523
524             stats['PC'] = PC
525             stats['RR'] = RR
526             stats['PQ'] = PQ
527
528             info('metablocking run analysed.', \
529                  _0={
530                      'n_cluster_pairs': n_cluster_pairs,
531                      'n_meta_pairs': n_meta_pairs,
532                      }, \
533                  _1={
534                      'n_true_positive': n_true_positive,
535                      'n_false_positive': n_false_positive,
536                      'n_true_negative': n_true_negative,
537                      'n_false_negative': n_false_negative,
538                      }, \
539                  _2={
540                      'precision': precision,
541                      'recall': recall,
542                      }, \
543                  _3={
544                      'f_measure': f_measure,
545                      }, \
546                  _4={
547                      'PC': PC,
548                      #'RR_complete': RR_complete,
549                      #'RR_cheating': RR_cheating,
550                      'RR': RR,
551                      'PQ': PQ,
552                      })
553
554             info('work completed.')
555
556             cu_w.close()
```

```
557             db_w.close()
558
559             info('batch.py ended.')
560
561 main()
562
563 time_stopped = time.clock()
564 #stats['time_stopped'] = time_stopped
565 stats['Overall Runtime.Runtime'] = time_stopped - time_started
566
567 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,
         #'t_start': time_started,
         }

# 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
```

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

```
89          print '␣+', f1(k), '.'*(20-len(k)), '[', '␣'.join(map(f2
                , sorted(s))), ']'
90      return
91
92  # load the table into memory
93
94  print >>err, c('#␣step␣0:␣reading␣the␣table␣into␣memory␣and␣
        encoding␣attributes')
95  print >>err, c('#␣␣␣␣␣␣␣␣␣through␣numbers␣to␣increase␣
        performance')
96  print >>err, c('#␣␣␣␣␣␣␣␣␣(this␣is␣not␣part␣of␣metablocking)')
97  print >>err
98
99  with timer('Setup'):
100     block_keys = {}
101     block_to_value = {}
102     table = hashtable(set)
103     clusters = hashtable(set)
104
105     db = connect('cleaned.sqlite3')
106     cu = db.cursor()
107
108     ok_chars  = ascii_letters + digits + '␣'
109
110     sane_str = lambda c: c in ok_chars
111
112     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)):
113         _id = int(record[0])
114         _cl = int(record[1])
115
116         clusters[_cl].add(_id)
117
118         for value in record[2:]:
119             if value:
120                 value = unicode(value).strip()
121
122                 if value:
123                     values = u''.join(filter(sane_str, value)).
                            lower().split()
124
125                     for value in values:
126
127                         if value in bad_values:
128                             continue
129
130                         block = block_keys.get(value, None)
```

```
131
132                             if block == None:
133                                 block = len(block_keys)
134                                 block_keys[value] = block
135                                 block_to_value[block] = value
136
137                             table[_id].add(block)
138
139     cu.close()
140     db.close()
141     del cu, db
142
143
144     print >>err, c('# step 1: transform the table into a
            collection of blocks')
145     print >>err, c('#          (this is not part of metablocking)
            ')
146     print >>err
147
148     def extract_blocks(table):
149         blocks = hashtable(set)
150         for entity, attributes in table.items(): # do entities
                need to be sorted in block?
151             for attribute in attributes:
152                 blocks[attribute].add(entity)
153         for block, entities in blocks.items(): # yes they do!!!
154             entities = list(entities)
155             entities.sort()
156             blocks[block] = entities
157         return blocks
158
159     blocks = extract_blocks(table)
160
161     del table
162
163 print >>err, c('# meta 1: create the reverse index from all
        blocks')
164 print >>err, c('#          (this is where metablocking starts)')
165 print >>err, c('#')
166 print >>err, c('#          the blocks in the reverse index have
        to be')
167 print >>err, c('#          in the same order as we process the
        blocks')
168 print >>err, c('#          for the sum calculation to work.')
169 print >>err
170
171 def build_rev_idx(blocks):
172     rev_idx = hashtable(list) # must be a hashtable of SORTED
            lists
```

```
173     for block, entities in sorted(blocks.items()): # add blocks
            in SORTED order.
174         for entity in entities:
175             rev_idx[entity].append(block)
176     return rev_idx
177
178 with timer('RevIdx'):
179     rev_idx = build_rev_idx(blocks)
180
181 #print 'REVERSE INDEX:'
182 #show(rev_idx, e, b)
183 #print
184
185 print >>err, c('# meta 2: calculate the "total_weight", "
        n_distinct_edges",')
186 print >>err, c('#          and "avg_weight" by iterating through
        all blocks')
187 print >>err, c('#          in sorted order.')
188 print >>err
189
190 def get_weight(block, e1, e2):
191     #print '    \- get_weight:', '(current:', b(block), ') ', \
192     #                           e(e1), ' '*(10-len(str(e1))),
            '-', \
193     #                           e(e2), ' '*(10-len(str(e2)))
194
195     blocks_e1 = rev_idx[e1]
196     blocks_e2 = rev_idx[e2]
197
198     #print '          \- rev_idx[e1]:', e(e1), '.'*(15-len(str(
            e1))), '[', \
199     #                                  ' '.join(map(b,
            blocks_e1)), ']'
200     #print '          \- rev_idx[e2]:', e(e2), '.'*(15-len(str(
            e2))), '[', \
201     #                                  ' '.join(map(b,
            blocks_e2)), ']'
202     #print '           /'
203
204     common_blocks = 0L
205     first_common = False
206     for b1 in blocks_e1:
207         for b2 in blocks_e2:
208             #print '          /', b(b1), ' '*(10-len(str(b1))),
                    '==', \
209             #                   b(b2), ' '*(10-len(str(b2))),
210
211
212             if b1 == b2:
```

```
213                     common_blocks += 1
214
215                     if not first_common:
216                         # print '&      first common',
217                         first_common = True
218                         if b1 != block:
219                             # print '& NOT current block => return
                                -1.'
220                             return -1 # error code
221                         else:
222                             # print '&      current block => continue
                                .'
223                             pass
224                     else:
225                         # print '& NOT first common => continue.'
226                         pass
227                 else:
228                     # print '=> skip.'
229                     pass
230
231         #print '            | return', common_blocks
232         return common_blocks
233
234     with timer('Graph'):
235         print >>err, 'CALCULATING total_weight, n_distinct_edges,
                average_weight'
236         total_weight = 0L
237         n_distinct_edges = 0L
238
239         for block, entities in sorted(blocks.items()):
240             for e1, e2 in all_combinations(blocks[block]):
241                 weight = get_weight(block, e1, e2)
242                 if weight != -1:
243                     total_weight += weight
244                     n_distinct_edges += 1
245
246         average_weight = float(total_weight) / n_distinct_edges
247         print >>err, ' - total_weight:    ', total_weight
248         print >>err, ' - n_distinct_edges:', n_distinct_edges
249         print >>err, ' - average_weight:  ', average_weight
250         print >>err
251         stats['Total Weight.N'] = total_weight
252         stats['Distinct Edges.N'] = n_distinct_edges
253         stats['Average Weight.N'] = average_weight
254
255     print >>err, c('# meta 3: re-iterate through all blocks and
            apply the pruning')
256     print >>err, c('#         criterion. create the output blocks.')
257     print >>err
```

```
258
259
260 with timer('Pruning'):
261     # print 'APPLY PRUNING CRITERION AND OUTPUT NEW BLOCKS'
262     new_blocks = hashtable(set)
263
264     for block, entities in sorted(blocks.items()):
265         #print ' - working on:', b(block)
266         for e1, e2 in all_combinations(blocks[block]):
267             weight = get_weight(block, e1, e2)
268             #print '           \- weight:', weight,
269             if weight < average_weight:
270                 #print '< ', average_weight, '=> skip.'
271                 pass
272             else:
273                 new_blocks[block].add((e1, e2))
274                 #print '>=', average_weight, '=> add to block.'
275                 #print
276         #print '    \- new block:', b(block), '[', ' '.join(map(
                e, new_blocks[block])), ']'
277         #print
278
279     #print 'NEW BLOCKS:'
280     #show(new_blocks, b, e)
281     #print
282
283 print >>err, c('# post 1: measure stuff')
284 print >>err, c('#          (this is not part of metablocking
        anymore.)')
285 print >>err
286
287 with timer('Scoring'):
288
289     ground_truth = set()
290
291     n_true_positive = 0L
292
293     for cluster, entities in clusters.items():
294         if len(entities) > 1:
295             ground_truth = ground_truth.union(sorted(
                    all_combinations(entities)))
296         #else:
297         #    # single record entity
298         #    n_true_positive += 1
299
300     print >>err, '# ground_truth:', len(ground_truth)
301     stats['Ground Truth Entity Pairs.N'] = len(ground_truth)
302
303     #all_comparisons = set()
```

```
304     #for block, comparisons in new_blocks.items():
305     #    all_comparisons = all_comparisons.union(comparisons)
306
307     all_comparisons = list(new_blocks.values())
308     while len(all_comparisons) > 1:
309         for _ in xrange(len(all_comparisons)/2):
310             tmp = all_comparisons.pop(0)
311             tmp = tmp.union(all_comparisons.pop(0))
312             all_comparisons.append(tmp)
313     all_comparisons = all_comparisons[0]
314     stats['Output Entity Pairs.N'] = len(all_comparisons)
315
316     #all_comparisons = list(new_blocks.values())
317     #chunks = lambda l, n: [l[x: x+n] for x in xrange(0, len(l),
            n)]
318
319     #while len(all_comparisons) > 1:
320     #    all_comparisons = map(_merge, chunks(all_comparisons,
            2))
321     #all_comparisons = all_comparisons[0]
322
323     print >>err, '# all_comparisons:', len(all_comparisons)
324     print >>err
325
326     n_true_positive += len(ground_truth.intersection(
            all_comparisons))
327     n_false_positive = len(all_comparisons - ground_truth)
328     n_false_negative = len(ground_truth - all_comparisons)
329
330     stats['True Positives.N'] = n_true_positive
331     stats['False Positives.N'] = n_false_positive
332     stats['False Negatives.N'] = n_false_negative
333
334     recall = float(n_true_positive) / (n_true_positive +
            n_false_negative)
335     precision = float(n_true_positive) / (n_true_positive +
            n_false_positive)
336     f_measure = 2 * precision * recall / (precision + recall)
337
338     print >>err, 'MEASURING QUALITY'
339     print >>err, ' - recall:   ', recall
340     print >>err, ' - precision:', precision
341     print >>err, ' - f-measure:', f_measure
342     print >>err
343
344     stats['Recall.Recall'] = recall
345     stats['Precision.Precision'] = precision
346     stats['F-Measure.F-Measure'] = f_measure
347
```

```
348 time_stopped = clock()
349 #stats['time_stopped'] = time_stopped
350 stats['Overall Runtime.Runtime'] = time_stopped - time_started
351
352 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')
```