Applying Meta-Blocking to Improve Efficiency in Entity Resolution

Tobias Ammann
tag@adnm.ch
08-922-106

May 11, 2014

Facharbeit Informatik (Inf 30)
written at

University of Zurich: Department of Informatics
Prof. Dr. Michael Böhlen

Supervisor: Pei Li PhD

Abstract

This report 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).

Contents

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

```
  Ē: Bag of edges (including duplicates).
  // Graph construction:
  Ē = all entity pairs of all blocks in B̄.
  sort Ē.
  N_distinct = 1
  pair_last = Ē_0
  for pair in Ē_{1..N} do
    if pair ≠ pair_last then
      N_distinct++
      pair_last = pair
    end if
  end for
  // Graph pruning:
  W_avg = |Ē| / N_distinct
  pair_last = Ē_0.
  W_pair = 1.
  for pair in Ē_{1..N} do
    if pair ≠ pair_last then
      if W_pair ≥ W_avg then
        add pair to B̄_output.
      end if
      W_pair = 0
      pair_last = pair
    end if
    W_pair++
  end for
  if W_pair ≥ W_avg then
    add pair to B̄_output.
  end if
  return B̄_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 $b_i \in \bar{R}_{pair_0}$ **do**
    for $b_j \in \bar{R}_{pair_1}$ **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: B̄_input
Output: B̄_output
  // Reversed Index:
  R̄: Reversed Index storing each entity's blocks.
  // Graph construction:
  W_total = 0
  N_distinct = 0
  for b̄ ∈ B̄_input in sorted order do
    for pair ∈ b̄ do
      W_pair = GetWeight(b, R̄, pair)
      if w ≠ −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 b̄ ∈ B̄_input in sorted order do
    for pair ∈ b̄ do
      W_pair = GetWeight(b, R, pair)
      if W_pair ≥ W_avg then
        add pair to B̄_output
      end if
    end for
  end for
  return  B̄_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.

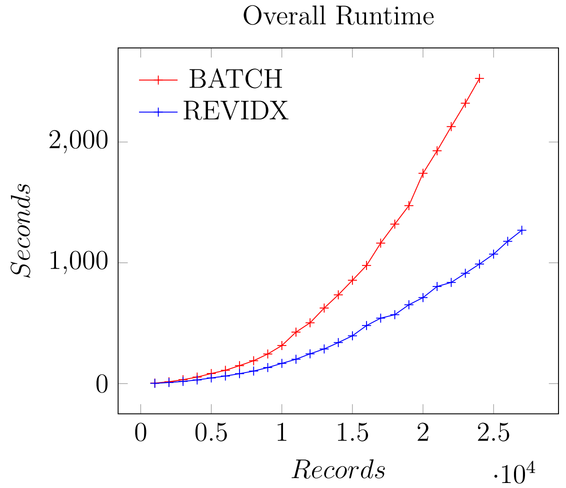
2. The decreasing average number of blocks per entity hints at a large number of sparse records. Given the number of fields in the dataset, we expect a lower bound of 6 blocks per entity for complete records.

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

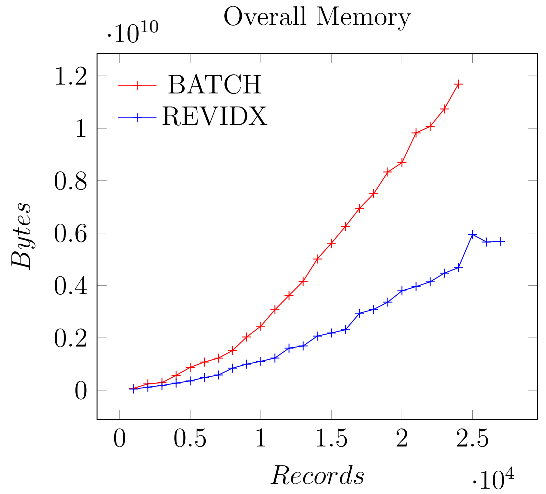
3.2.1 Comparison of Runtime

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.



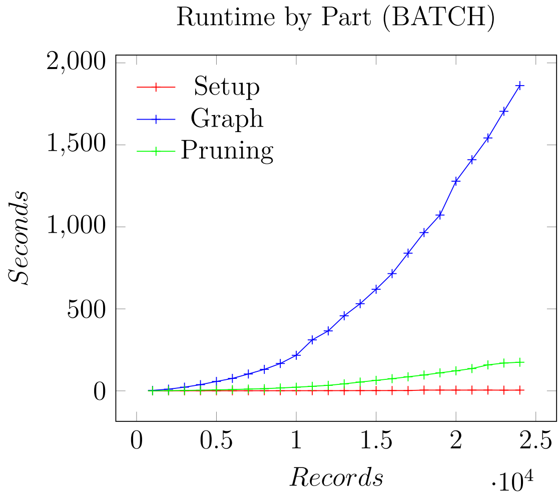
3.2.2 Comparison of Memory Usage

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

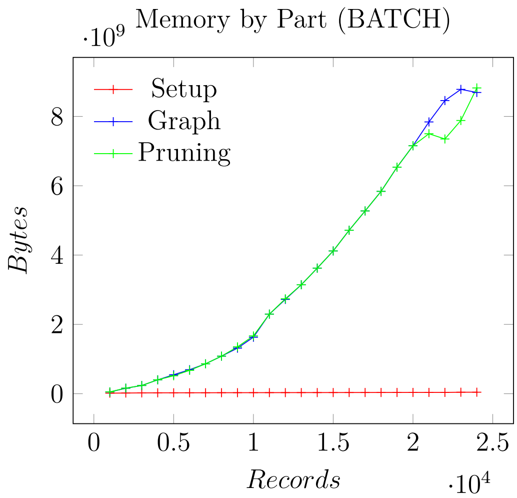


3.2.3 Detailed Analysis of BATCH

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*).

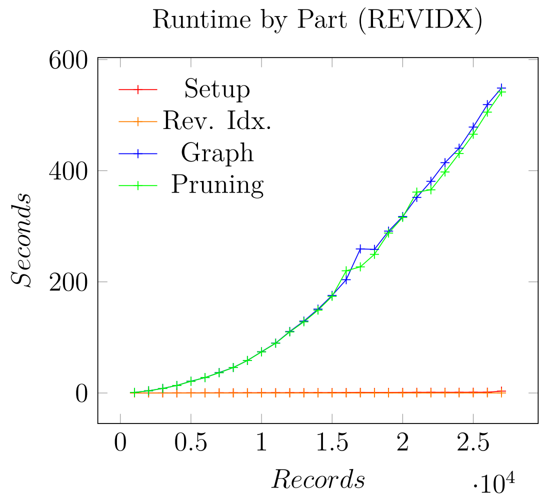


The memory consumption of BATCH is also dominated by the construction of the *Graph*. The later *Pruning* stage, only consumes marginally more memory for $\bar{B}_{output}$. The small reduction in memory consumption above 25 K records is an artefact of the implementation of $\bar{E}$.

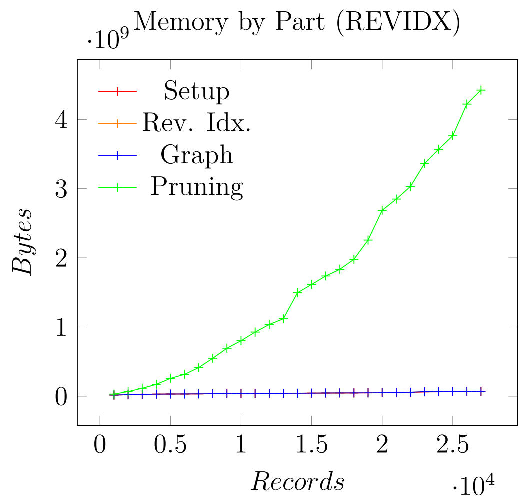


3.2.4 Detailed Analysis of REVIDX

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.

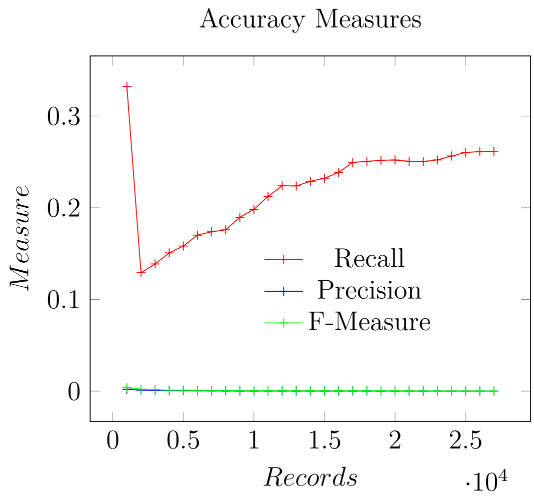


3.3 Accuracy of the method

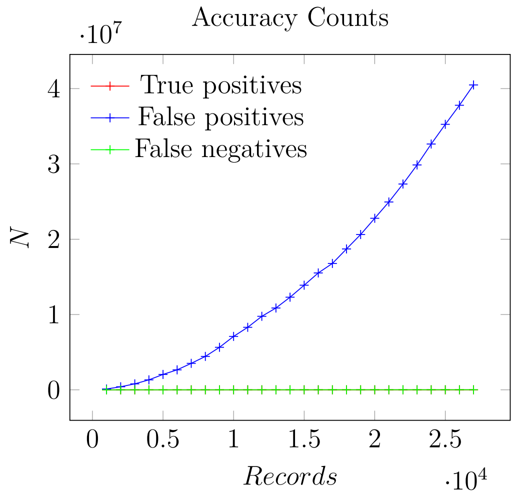
The quality of the output generated by meta-blocking was measured using *precision*, *recall*, and *f-measure*, by comparing $\bar{B}_{output}$ against a list of entity pairs generated using the *cluster* attribute of the dataset.

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

```

```
get_wdb = functools.partial(get_du, 'batch.sqlite3')
get_ldb = functools.partial(get_du, 'batch.leveldb')

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
        self.start_disk = 0L
    def __enter__(self):
        cput = P.cpu_times()
        memi = P.memory_info_ex()
        self.start_sys  = cput.system
        self.start_user = cput.user
        self.start_rss  = memi.rss
        self.start_disk = get_wdb() + get_ldb()
    def __exit__(self, *args):
        cput = P.cpu_times()
        memi = P.memory_info_ex()
        self.stop_sys  = cput.system
        self.stop_user = cput.user
        self.stop_rss  = memi.rss
        self.stop_disk = get_wdb() + get_ldb()
        t_elapsed_sys  = self.stop_sys - self.start_sys
        t_elapsed_user = self.stop_user - self.start_user
        t_elapsed = t_elapsed_sys + t_elapsed_user
        print >>sys.stderr, blessings.Terminal().yellow('timer: {} took {} (user: {}, sys: {}) seconds.'.format(self.name, t_elapsed, t_elapsed_user, t_elapsed_sys))
        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))
        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))
        print >>sys.stderr
        stats[self.name+'.Memory'] = self.stop_rss + self.stop_disk
        stats[self.name+'.Runtime'] = t_elapsed

def main():
    info('retry3.py started.')

    # opening connections to all databases and some necessary cleaning and setup.
    # - db_s: data source
```

```
    # - db_w: in memory working set
    # - db_e: on disk leveldb hashtable
    #

    with timer('Setup'):
        db_s = sqlite3.connect('cleaned.sqlite3')
        cu_s = db_s.cursor()
        #db_w = sqlite3.connect(':memory:')
        if os.path.exists('batch.sqlite3'):
            os.remove('batch.sqlite3')
        db_w = sqlite3.connect('batch.sqlite3')
        cu_w = db_w.cursor()

        cu_w.execute('''
            CREATE TABLE profile (id integer not null, cluster integer not null, block integer not null);
        ''')
        db_w.commit()

        if os.path.exists('batch.leveldb'):
            info('cleaning up old hashtable...')
            shutil.rmtree('batch.leveldb')

        megabyte = 1024**2
        db_e = leveldb.LevelDB('batch.leveldb', \
                               block_cache_size=128*megabyte, \
                               write_buffer_size=128*megabyte)

        info('databases ready.')

        block_keys = {}
        block_to_value = {}
        clusters = collections.defaultdict(set)


        ok_chars  = string.ascii_letters + string.digits + ' '

        sane_str = lambda c: c in ok_chars

        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)):
            _id = int(record[0])
            _cl = int(record[1])

            clusters[_cl].add(_id)

            for value in record[2:]:
```

```
                    if value:
                        value = unicode(value).strip()

                        if value:
                            values = u''.join(filter(sane_str, value
                                )).lower().split()

                            for value in values:

                                if value in bad_values:
                                    continue

                                block = block_keys.get(value, None)

                                if block == None:
                                    block = len(block_keys)
                                    block_keys[value] = block
                                    block_to_value[block] = value

                                cu_w.execute('INSERT INTO profile (
                                    id, cluster, block) VALUES
                                    (?,?,?);', (_id, _cl, block))

        cu_s.close()
        db_s.close()
        del cu_s, db_s

        cu_w.execute('CREATE INDEX iprofblock ON profile (block)
            ;')
        db_w.commit()


    with timer('Graph'):
        info('creating graph...')

        # add all edges of the graph by adding them to a
            hashtable.
        # use some hacks to keep the memory usage low.
        #
        packer = struct.Struct('>I').pack
        unpacker = struct.Struct('>I').unpack
        def pack(n):
            return packer(n)

        def unpack(s):
            return unpacker(s)[0]

        def add_edges(block, ids):
            if len(ids) < 2:
```

```
                return 0L

            #print 'adding:', block, ids

            b_block = pack(block)

            ids = list(set(ids))
            ids.sort()
            b_ids = map(pack, ids)

            n_edges = 0L
            wb = leveldb.WriteBatch()

            for edge in itertools.combinations(b_ids, 2):
                wb.Put(edge[0] + edge[1] + b_block, '')
                n_edges += 1

            db_e.Write(wb)

            return n_edges

        with timer('meta 2-insert'):

            n_edges = 0L
            last_block = None
            block_members = []

            for _id, block in cu_w.execute('SELECT id, block FROM profile ORDER BY block;'):
                if block != last_block:
                    n_edges += add_edges(last_block, block_members)
                    last_block = block
                    block_members = [_id]

                else:
                    block_members.append(_id)

            if block_members:
                n_edges += add_edges(last_block, block_members)

            info('edges inserted.', n_edges=n_edges)


        with timer('meta 2-counting'):

            info('calculate edge weights...')

```

```
        # scan through all edges and count them to calculate their edge weight.
        # calculate their average.
        #
        cu_w.execute('''
                CREATE TABLE edges (
                    n1 integer not null,
                    n2 integer not null,
                    weight integer
                );
            ''')
        db_w.commit()

        b_pre_edge = '\x00'*12
        b_post_edge = '\xff'*12

        last_edge = b_post_edge
        weight = 0L
        n_distinct_edges = 0L
        total_weight = 0L

        edges = db_e.RangeIter(key_from=b_pre_edge, key_to=b_post_edge, include_value=False)
        for edge in edges:
            if edge.startswith(last_edge):
                weight += 1
            else:
                if weight:
                    total_weight += weight
                    n1 = unpack(last_edge[:4])
                    n2 = unpack(last_edge[4:])
                    cu_w.execute('INSERT INTO edges (n1, n2, weight) VALUES (?,?,?);', (n1, n2, weight))
                weight = 1L
                n_distinct_edges += 1
                last_edge = edge[:8]

        if weight:
            n1 = unpack(last_edge[:4])
            n2 = unpack(last_edge[4:])
            total_weight += weight
            cu_w.execute('INSERT INTO edges (n1, n2, weight) VALUES (?,?,?);', (n1, n2, weight))

        db_w.commit()

        avg_weight = float(total_weight) / n_distinct_edges

```

```
            info('edges counted up.', n_distinct_edges=n_distinct_edges, total_weight=total_weight, avg_weight=avg_weight)

            stats['Distinct Edges.N'] = n_distinct_edges
            stats['Total Weight.N'] = total_weight
            stats['Average Weight.N'] = avg_weight

    with timer('Pruning'):
        info('pruning graph...')

        cu_w.execute('''
            DELETE FROM edges WHERE weight < ?;
        ''', (avg_weight,))
        db_w.commit()

        info('edges saved.')


    with timer('Scoring'):
        info('scoring metablocking run...')

        ground_truth = map(lambda entities: set(itertools.combinations(sorted(entities), 2)), clusters.values())
        while len(ground_truth) > 1:
            for _ in xrange(len(ground_truth)/2):
                tmp = ground_truth.pop(0)
                tmp = tmp.union(ground_truth.pop(0))
                ground_truth.append(tmp)
        ground_truth = ground_truth[0]

        print >>sys.stderr, '# ground_truth:', len(ground_truth)
        stats['Ground Truth Entity Pairs.N'] = len(ground_truth)

        meta_pairs = set(cu_w.execute('''
            SELECT n1, n2 FROM edges;
        ''').fetchall())
        stats['Output Entity Pairs.N'] = len(meta_pairs)

        n_cluster_pairs = len(ground_truth)
        n_meta_pairs = len(meta_pairs)

        # true positive: PAIR found in INPUT and OUTPUT blocks.
        n_true_positive = len(ground_truth.intersection(meta_pairs))
        # false positive: PAIR found in OUTPUT but not in INPUT.
        n_false_positive = len(meta_pairs - ground_truth)
        # true negative: PAIR found neither in INPUT nor OUTPUT.
```

```
        n_true_negative = '-------'
        # false negative: PAIR found in INPUT but not in OUTPUT.
        n_false_negative = len(ground_truth - meta_pairs)

        stats['True Positives.N'] = n_true_positive
        stats['False Positives.N'] = n_false_positive
        stats['False Negatives.N'] = n_false_negative


        # recall and precision:
        recall = float(n_true_positive) / (n_true_positive + n_false_negative)
        precision = float(n_true_positive) / (n_true_positive + n_false_positive)

        # f-measure
        f_measure = 2 * precision * recall / (precision + recall)

        stats['Recall.Recall'] = recall
        stats['Precision.Precision'] = precision
        stats['F-Measure.F-Measure'] = f_measure

    with timer('post 1-paperstats'):
        # PC
        cluster_pairs_sharing_block = set(cu_w.execute('''
            SELECT DISTINCT p1.id, p2.id FROM profile AS p1, profile AS p2
                WHERE p1.cluster = p2.cluster AND
                      p1.block = p2.block AND
                      p1.id < p2.id;
        ''').fetchall())

        Din = len(cluster_pairs_sharing_block)
        Dout = len(meta_pairs.intersection(cluster_pairs_sharing_block))
        PC = float(Dout) / Din

        # RR
        n_edges_remaining = cu_w.execute('SELECT count(*) FROM edges;').fetchone()[0]

        #RR_complete = 1.0 - float(n_edges_remaining) / (n_edges + n_edges_skipped)
        #RR_cheating = 1.0 - float(n_edges_remaining) / n_edges
        RR = 1.0 - float(n_edges_remaining) / n_edges

        # PQ
        PQ = float(n_true_positive) / n_edges
```

```

            stats['PC'] = PC
            stats['RR'] = RR
            stats['PQ'] = PQ

            cu_w.close()
            db_w.close()

            info('batch.py ended.')

main()

time_stopped = time.clock()
stats['Overall Runtime.Runtime'] = time_stopped - time_started

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
```

```
        self.start_disk = 0L
    def __enter__(self):
        cput = P.cpu_times()
        memi = P.memory_info_ex()
        self.start_sys  = cput.system
        self.start_user = cput.user
        self.start_rss  = memi.rss
        self.start_disk = 0L
    def __exit__(self, *args):
        cput = P.cpu_times()
        memi = P.memory_info_ex()
        self.stop_sys  = cput.system
        self.stop_user = cput.user
        self.stop_rss  = memi.rss
        self.stop_disk = 0L
        t_elapsed_sys  = self.stop_sys - self.start_sys
        t_elapsed_user = self.stop_user - self.start_user
        t_elapsed = t_elapsed_sys + t_elapsed_user
        print >>err, T().yellow('timer: {} took {} (user: {}, sys: {}) seconds.'.format(self.name, t_elapsed, t_elapsed_user, t_elapsed_sys))
        print >>err, T().yellow('timer: rss = {} MiB. (change: {} MiB).'.format(self.stop_rss/1048576.0, (self.stop_rss-self.start_rss)/1048576.0))
        print >>err, T().yellow('timer: disk = {} MiB. (change: {} MiB.'.format(self.stop_disk/1048576.0, (self.stop_disk-self.start_disk)/1048576.0))
        print >>err
        stats[self.name+'.Memory'] = self.stop_rss + self.stop_disk
        stats[self.name+'.Runtime'] = t_elapsed

def all_combinations(entities):
    return combinations(entities, 2)

c = lambda v: T().bold_bright_black(str(v))
b = lambda v: T().bold_bright_red(str(v))
e = lambda v: T().underline_white(str(v))

def show(d, f1, f2):
    for k, s in d.items():
        k = str(k)
        print ' +', f1(k), '.'*(20-len(k)), '[', ' '.join(map(f2, sorted(s))), ']'
    return

# load the table into memory

```

```python
print >>err, c('# step 0: reading the table into memory and encoding attributes')
print >>err, c('#          through numbers to increase performance')
print >>err, c('#          (this is not part of metablocking)')
print >>err

with timer('Setup'):
    block_keys = {}
    block_to_value = {}
    table = hashtable(set)
    clusters = hashtable(set)

    db = connect('cleaned.sqlite3')
    cu = db.cursor()

    ok_chars  = ascii_letters + digits + ' '

    sane_str = lambda c: c in ok_chars

    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)):
        _id = int(record[0])
        _cl = int(record[1])

        clusters[_cl].add(_id)

        for value in record[2:]:
            if value:
                value = unicode(value).strip()

                if value:
                    values = u''.join(filter(sane_str, value)).lower().split()

                    for value in values:

                        if value in bad_values:
                            continue

                        block = block_keys.get(value, None)

                        if block == None:
                            block = len(block_keys)
                            block_keys[value] = block
                            block_to_value[block] = value

```

```
                        table[_id].add(block)

    cu.close()
    db.close()
    del cu, db


    print >>err, c('# step 1: transform the table into a collection of blocks')
    print >>err, c('#          (this is not part of metablocking)')
    print >>err

    def extract_blocks(table):
        blocks = hashtable(set)
        for entity, attributes in table.items(): # do entities need to be sorted in block?
            for attribute in attributes:
                blocks[attribute].add(entity)
        for block, entities in blocks.items(): # yes they do!!!
            entities = list(entities)
            entities.sort()
            blocks[block] = entities
        return blocks

    blocks = extract_blocks(table)

    del table

print >>err, c('# meta 1: create the reverse index from all blocks')
print >>err, c('#          (this is where metablocking starts)')
print >>err, c('#')
print >>err, c('#          the blocks in the reverse index have to be')
print >>err, c('#          in the same order as we process the blocks')
print >>err, c('#          for the sum calculation to work.')
print >>err

def build_rev_idx(blocks):
    rev_idx = hashtable(list) # must be a hashtable of SORTED lists
    for block, entities in sorted(blocks.items()): # add blocks in SORTED order.
        for entity in entities:
            rev_idx[entity].append(block)
    return rev_idx

```

```
with timer('RevIdx'):
    rev_idx = build_rev_idx(blocks)

print >>err, c('# meta 2: calculate the "total_weight", "n_distinct_edges",')
print >>err, c('#          and "avg_weight" by iterating through all blocks')
print >>err, c('#          in sorted order.')
print >>err

def get_weight(block, e1, e2):
    blocks_e1 = rev_idx[e1]
    blocks_e2 = rev_idx[e2]

    common_blocks = 0L
    first_common = False
    for b1 in blocks_e1:
        for b2 in blocks_e2:
            if b1 == b2:
                common_blocks += 1

                if not first_common:
                    first_common = True
                    if b1 != block:
                        return -1 # error code
                    else:
                        pass
                else:
                    pass
            else:
                pass

    return common_blocks

with timer('Graph'):
    print >>err, 'CALCULATING total_weight, n_distinct_edges, average_weight'
    total_weight = 0L
    n_distinct_edges = 0L

    for block, entities in sorted(blocks.items()):
        for e1, e2 in all_combinations(blocks[block]):
            weight = get_weight(block, e1, e2)
            if weight != -1:
                total_weight += weight
                n_distinct_edges += 1

    average_weight = float(total_weight) / n_distinct_edges
    stats['Total Weight.N'] = total_weight
```

```
    stats['Distinct Edges.N'] = n_distinct_edges
    stats['Average Weight.N'] = average_weight

print >>err, c('# meta 3: re-iterate through all blocks and apply the pruning')
print >>err, c('#         criterion. create the output blocks.')
print >>err


with timer('Pruning'):
    # print 'APPLY PRUNING CRITERION AND OUTPUT NEW BLOCKS'
    new_blocks = hashtable(set)

    for block, entities in sorted(blocks.items()):
        for e1, e2 in all_combinations(blocks[block]):
            weight = get_weight(block, e1, e2)
            if weight < average_weight:
                pass
            else:
                new_blocks[block].add((e1, e2))

print >>err, c('# post 1: measure stuff')
print >>err, c('#         (this is not part of metablocking anymore.)')
print >>err

with timer('Scoring'):

    ground_truth = set()

    n_true_positive = 0L

    for cluster, entities in clusters.items():
        if len(entities) > 1:
            ground_truth = ground_truth.union(sorted(all_combinations(entities)))

    stats['Ground Truth Entity Pairs.N'] = len(ground_truth)

    all_comparisons = list(new_blocks.values())
    while len(all_comparisons) > 1:
        for _ in xrange(len(all_comparisons)/2):
            tmp = all_comparisons.pop(0)
            tmp = tmp.union(all_comparisons.pop(0))
            all_comparisons.append(tmp)
    all_comparisons = all_comparisons[0]
    stats['Output Entity Pairs.N'] = len(all_comparisons)

```

```
    n_true_positive += len(ground_truth.intersection(all_comparisons))
    n_false_positive = len(all_comparisons - ground_truth)
    n_false_negative = len(ground_truth - all_comparisons)

    stats['True Positives.N'] = n_true_positive
    stats['False Positives.N'] = n_false_positive
    stats['False Negatives.N'] = n_false_negative

    recall = float(n_true_positive) / (n_true_positive + n_false_negative)
    precision = float(n_true_positive) / (n_true_positive + n_false_positive)
    f_measure = 2 * precision * recall / (precision + recall)

    stats['Recall.Recall'] = recall
    stats['Precision.Precision'] = precision
    stats['F-Measure.F-Measure'] = f_measure

time_stopped = clock()
stats['Overall Runtime.Runtime'] = time_stopped - time_started

print 'REVIDX', stats
```

A.4 Description of Dataset

```
import sqlite3
import collections
import string
import tabulate

# helpers

ok_chars  = string.ascii_letters + string.digits + ' '
sane_str = lambda c: c in ok_chars
bad_values = set(string.ascii_letters + string.digits) # single letters/digits

query_string = '''
    SELECT id,
           cluster,
           name,
           sort_name,
           type,
           area,
           gender,
           comment,
           begin_year,
           end_year
      FROM artist_sample
  ORDER BY cluster, id;
'''

def extract_stats(ht):
    n_ht = len(ht)
    s_min =  999999999
    s_max = -999999999
    s_sum = 0L
    for k, s in ht.items():
        s_min = min(s_min, len(s))
        s_max = max(s_max, len(s))
        s_sum += len(s)
    s_avg = float(s_sum) / n_ht

    return n_ht, s_min, s_max, s_avg

step_size =  [1000, 2000, 5000, 10000, 20000, 30000]
stop_size = max(step_size)

# connect to database
db = sqlite3.connect('cleaned.sqlite3')
cu = db.cursor()

```

```
# value->bid and bid->value can stay the same across subsets
value_to_bid = {}
bid_to_value = {}

# output statistics / helpers
stats = collections.defaultdict(list)

def dpt(k, y):
    stats[k].append(y)

# associations... kept globally for incremental approach.
entity2block = collections.defaultdict(set)
block2entity = collections.defaultdict(set)
entity2clust = collections.defaultdict(set)
clust2entity = collections.defaultdict(set)
block2clustr = collections.defaultdict(set)
clustr2block = collections.defaultdict(set)

for record in cu.execute(query_string):
    _id = int(record[0])
    _cl = int(record[1])

    # add cluster-entity associations
    clust2entity[_cl].add(_id)
    entity2clust[_id].add(_cl)

    for value in record[2:]:
        if value:
            # value is not none

            value = unicode(value).strip()

            if value:
                # value is not an empty string

                values = u''.join(filter(sane_str, value)).lower
                    ().split()

                for value in values:

                    if value in bad_values:
                        continue

                    bid = value_to_bid.get(value, None)

                    if bid == None:
                        bid = len(value_to_bid)
                        value_to_bid[value] = bid
                        bid_to_value[bid] = value
```

```
                    # add entity-block, and cluster-block associations
                    entity2block[_id].add(bid)
                    block2entity[bid].add(_id)
                    clustr2block[_cl].add(bid)
                    block2clustr[bid].add(_cl)

    n_records = len(entity2block)

    if n_records in step_size:
        # calculate statistics
        print 'calculating statistics... n_records =', n_records

        #EC = extract_stats(entity2clust)
        CE = extract_stats(clust2entity)
        EB = extract_stats(entity2block)
        BE = extract_stats(block2entity)
        #CB = extract_stats(clustr2block)
        #BC = extract_stats(block2clustr)

        # 1. table of input blocks
        # ------------------------

        # - n_records
        dpt('n-records', n_records)

        # - n_blocks
        dpt('n-blocks', BE[0])

        # - n_clusters
        dpt('n-clusters', CE[0])

        # - block size: min, max, avg
        dpt('blocksize-min', BE[1])
        dpt('blocksize-max', BE[2])
        dpt('blocksize-avg', "{0:.2f}".format(BE[3]))

        # - n_sebs (single entity blocks)
        dpt('n-sebs', len(filter(lambda (k, v): len(v)==1,
            block2entity.items())))

        # - bpe: min, max, avg (blocks per entity)
        dpt('bpe-min', EB[1])
        dpt('bpe-max', EB[2])
        dpt('bpe-avg', "{0:.2f}".format(EB[3]))

    if n_records == stop_size:
        break
```

```

cu.close()
db.close()

# output data for rendering
n_records = stats['n-records']

for k in stats:
    with file('report/dataset-stats/'+k, 'w') as fh:
        for i, v in enumerate(stats[k]):
            print >>fh, n_records[i], v

table = []

headers = [
    'n-records',
    'n-clusters',
    'n-blocks',
    'n-sebs',
    'blocksize-min',
    'blocksize-max',
    'blocksize-avg',
    'bpe-min',
    'bpe-max',
    'bpe-avg',
]
print_header = [
    'Records',
    'Clusters',
    'Blocks',
    '1-E. Blocks.',
    'Min.',
    'Max.',
    'Avg.',
    'Min.',
    'Max.',
    'Avg.',
]

for i in xrange(len(n_records)):
    row = []
    for k in headers:
        row.append(stats[k][i])
    table.append(row)

with file('report/dataset-table.tex', 'w') as fh:
    print >>fh, tabulate.tabulate(table, headers=print_header, tablefmt='latex')
```