Dynamic Record Blocking: Efficient Linking of Massive Databases in MapReduce

W.P. McNeill
Intelius Data Research
Bellevue, WA 98004, USA
bmcneill@intelius.com

Hakan Kardes
Intelius Data Research
Bellevue, WA 98004, USA
hkardes@intelius.com

Andrew Borthwick
Intelius Data Research
Bellevue, WA 98004, USA
aborthwick@intelius.com

ABSTRACT

Record Linkage is the task of identifying which records in a database refer to the same entity. A standard machine learning approach to this problem is to train a model that assigns scores to pairs of records where pairs scoring above a threshold are said to represent the same entity. However, it is too expensive to make pairwise comparisons among all records in large databases. "Blocking" is the process of grouping similar-seeming records into blocks that a machine learning component then explores exhaustively. In many blocking approaches, records are grouped together into blocks by shared properties that are indicators of duplication. However, when dealing with very large data sources, it is nearly impossible to determine any fixed set of properties at training time that will be optimal for the Zipfian distribution of values for these properties that we will encounter at run time. In this paper, we propose a novel *Dynamic Blocking* algorithm which automatically chooses the blocking properties in a data-driven way at execution time to efficiently determine which pairs of records in a data set should be examined as potential duplicates without creating the same pair across blocks. We demonstrate the viability of this algorithm for large data sets. We have scaled this system up to work on billions of records on an 80-node Hadoop cluster.

1. INTRODUCTION

A challenge for builders of databases whose information is culled from multiple sources is the detection of duplicates, where a single real-world entity gives rise to multiple records (see [10] for an overview). Online citation indexes need to be able to navigate the different capitalization and abbreviation conventions that appear in bibliographic entries. Government agencies need to know whether a record for "Robert Smith" living on "Northwest First Street" refers to the same person as one for a "Bob Smith" living on "1st St. NW". A standard machine learning approach to this problem is to train a model that assigns scores to pairs of records where pairs scoring above a threshold are said to represent the same entity. Transitive closure is then performed on this same-entity relationship to find the sets of duplicate records. Comparing all pairs of records is quadratic in the number of records and so is intractable for large data sets. In practice only a subset of the possible pairs is referred to the machine learning component and others are assumed to represent different entities. So a "Robert Smith"-"Bob Smith" record pair may be scored while a "Robert Smith"-"Barack Obama" pair is dismissed. This risks a false negative error for the system if the "Robert Smith" and "Barack Obama" records do in fact refer to the same person, but in exchange for this the system runs faster. The term of art for this process is *blocking*, because it groups similar-seeming records into blocks that a pairwise comparator then explores exhaustively.

This paper describes the blocking strategy used to deploy a massive database of personal information for an online people search company. This database distills a heterogeneous collection of publicly available data about people into coherent searchable profiles. This distillation process can be framed as a duplicate detection task. We have developed a novel blocking procedure that in addition to the standard performance/recall tradeoff is tailored to 1) scale to very large data sets and 2) robustly handle novel data sources. The first of these is necessary because we map billions of input records to hundreds of millions of people in the real world. This is only possible with distributed computing, and the need to distribute the work informs the design. The second is necessary because we are acquiring new and diverse sources of information all the time, so the hand-crafting of the blocking procedure by experts can become a bottleneck. We call this technique *dynamic blocking* because the blocking criteria adjust at run-time in response to the composition of the data set. We want blocking to be a mechanical process, not an art.

2. CONTRIBUTIONS AND PRIOR WORK

Even though there are many different blocking approaches such as traditional [19], Canopy Clustering [14], sorted neighborhood [11], Q-gram based [3], string map [12], Suffix Array [2], and numerous other approaches [1],[8],[9],[15], [18],[20] that have been proposed in recent years, the basic idea is generally to define a set of key fields from the data to determine which records will be in which block(s).

In this paper, we propose a scalable dynamic blocking approach that builds on work presented in [5], but never previously described in any scholarly venue. That work contained the fundamental idea of dividing oversized blocks into

This article was presented at the 9th International Workshop on Quality in Databases (QDB) 2012.

smaller sub-blocks. The system described in this paper extends this algorithm in a number of ways, including adapting it to run in a scalable, distributed computing environment, introducing pair deduplication, and a distinction between "top-level blocks" and "sub-blocks".

In traditional blocking (described in [19], where it is used to link a database of 4 billion with a database of 100 million), each record is compared only against records that share the same blocking key value, for example, only comparing records with same date of birth. In this approach, blocking keys are chosen to be as general as possible to span all the records and to keep recall high, but as specific as possible to considerably reduce the number of pairwise comparisons. According to a comprehensive analysis of blocking algorithms in [7], traditional blocking is one of the fastest approaches, however, there are several drawbacks. First, it is not able to handle typos in the data. For example, if the first-name in a record is written as "Andriw", instead of "Andrew", this record will not be compared with other "Andrew" records when using first-name as a blocking key. Second, the hand-crafting of the blocking key selection becomes a challenge when working with heterogeneous data sources and the size of each block is dependent on the frequency distributions of the block key values. For instance, there might be just ten "Andrew Borthwick" records in the database while there are millions of "John Smith" records. There will be a lot of unnecessary comparisons in the "John Smith" block as there are thousands of "John Smith's" in real life. Third, the same pair of records might be compared multiple times, both from blocks that are duplicates or subsets of each other and from the pair appearing in multiple, partially overlapping blocks.

Similar to traditional blocking, the sliding window approach (first described in [11] and used in [17]) identifies sets of individuating record properties as "blocking keys" according to which individual records may be ordered. A sliding window of a fixed size is moved along this list and pairs lying within that window are sent along to the linkage component. We note that a fixed sliding window might be too large when a name that is rare in the U.S. like "Hakan Kardes" is used as a blocking key, while being too small for a more common name like "William McNeill", a problem avoided by Dynamic Blocking.

Canopy clustering [14] quickly calculates a distance between all record pairs that share a blocking key. These distances are used to group records into blocks called canopies, and distance thresholds may be tuned to allow some records to appear in multiple canopies. Our Dynamic Blocking approach shares with this technique the strategy of allowing sets of records to overlap. However, dynamic blocking can be applied to data sets like the one we have, where there is no obvious quickly-calculable distance metric between records and the number of records makes even a fast calculation for all pairs intractable for common blocking keys like "William McNeill". Note that [14] tests their method on a database of 2000 person records whereas this work is executed on datasets sized in the billions of records.

Dynamic Blocking contrasts with [16], first of all in scale where [16] reports results on only 140K records. Dynamic Blocking shares with [16] an objective of trading off recall with efficiency, but in contrast to [16], does not require training data. Labeled training data is impractical for our application because data distributions may vary widely across different runs of our system and labeled data may not be representative of the actual run-time process. We use the term "Dynamic" blocking because in contrast to prior work, the blocking criteria are adjusted at run time rather than being fixed at training time as in [4] and [16]. Similarly, [18] describes a novel approach of dynamically combining matched records, $\langle a, b \rangle$, detected in one block in all other blocks where $\langle a, b \rangle$ are found, but uses blocking criteria that are fixed for a given run.

Consequently, this paper (together with [5]) makes the following contributions: 1) We demonstrate a novel, highly accurate blocking algorithm that can be used on datasets sized in the billions, an order of magnitude previously only accessible to simplistic algorithms like traditional blocking and sliding window. 2) Dynamic Blocking is simultaneously sensitive to differential frequencies of blocking keys ("William McNeill" vs. "Hakan Kardes"), while not requiring training data. 3) Although [16] executes in MapReduce, our algorithm is tailored to use MapReduce on datasets four orders of magnitude larger than what they consider. 4) We present a novel algorithm for ensuring that each pair of records is compared only once, even though that pair might be present in multiple blocks.

3. DYNAMIC BLOCKING

3.1 System Overview

The process starts by collecting billions of personal records from three sources of U.S. personal records to power a major commercial People Search Engine. The first source is derived from US government records, such as marriage, divorce and death records. The second is derived from publicly available web profiles, such as professional and social network public profiles. The third type is derived from commercial sources, such as financial and property reports (e.g., information made public after buying a house). Example fields on these records might include name, address, birthday, phone number, (encrypted) social security number, job title, and university attended. Note that different records will include different subsets of these example fields.

After collection and categorization, the Record Linkage process should link together all records belonging to the same real-world person. That is, this process should turn billions of input records into a few hundred million clusters of records (or profiles), where each cluster is uniquely associated with a single real-world U.S. resident.

Our system follows the standard high-level structure of a record linkage pipeline [10] by being divided into four major components: 1) data cleaning 2) blocking 3) pairwise linkage and 4) clustering. First, all records go through a cleaning process that starts with the removal of bogus, junk and spam records. Then all records are normalized to an approximately common representation. Finally, all major noise types and inconsistencies are addressed, such as empty/bogus fields, field duplication, outlier values and encoding issues. At this point, all records are ready for subsequent stages of Record Linkage. The blocking step, which is the focus of this paper, groups records by shared properties to determine which pairs of records should be examined by the pairwise linker as potential duplicates. Next, the linkage step assigns a score to pairs of records inside each block using a high precision machine learning model whose implementation is described in detail in [6]. If a pair

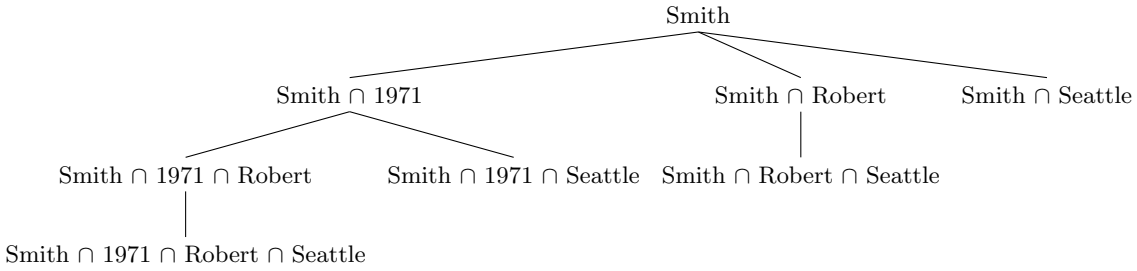

Figure 1: The root node of this tree represents an oversized block for the name Smith and the other nodes represent possible sub-blocks. The sub-blocking algorithm enumerates the tree breadth-first, stopping when it finds a correctly-sized sub-block.

scores above a user-defined threshold, the records are presumed to represent the same person. The clustering step first combines record pairs into connected components and then further partitions each connected component to remove inconsistent pair-wise links. Hence at the end of the entire record linkage process, the system has partitioned the input records into disjoint sets called profiles, where each profile corresponds to a single person.

The blocking step is described in detail below.

3.2 Blocks and Sub-blocks

How might we subdivide a huge number of records? We could start by grouping them into sets of the same first and last name. This would go a long way towards putting together records that represent the same person, but it would still be imperfect because people may have nicknames or change their names. To enhance this grouping we could consider a different kind of information like social security number. Fraud, error, and omission renders SSN imperfectly individuating; however, it will still help us put together records for, say, women who change their name when they get married. With a handful of keys like this we can build redundancy into our system to accommodate different types of error, omission, and natural variability. The blocks of records they produce may overlap, but this is desirable because it gives the clustering a chance to join records that blocking did not put together.

These blocks will vary widely in size. We may have a small set of "Barack Obama" records which can then be passed along immediately to the linkage component. However, we may have a set of millions of "Robert Smith" records which still needs to be cut down to size. One way to do this is to find other properties to further subdivide this set. The set of all Robert Smiths who have the same middle name is smaller than the set of all Robert Smiths, and intuitively records in this set will be more likely to represent the same person. Additionally we could block together all the Robert Smiths with the same phone number, or birthday, or who live in the same city. As with the original blocks, overlap between these sub-blocks is desirable. We do not have to be particularly artful in our choice of sub-blocking criteria: any property that seems like it might be individuating will do. As long as we have an efficient way to search the space, we can let the data dynamically choose different sub-blocking strategies for each oversize block.

More formally, this process can be understood in terms of operations on sets. In a set of N records there are $\frac{1}{2}N(N-1)$ unique pairs, so an enumeration over all of them is $O(N^2)$. The process of blocking divides this original set into k blocks, each of which contains at most a fixed maximum of M records. The exhaustive comparison of pairs from these sets is $O(k)$, and the constant factors are tractable if we choose a small enough M.

Call the elements in these sets *records*. An individual record can be thought of as a set of *properties*, where a property maps a *field* (e.g. City or First Name) to a *value* (e.g. Seattle or Robert).[1] It is possible to define a total ordering on properties. For instance we can alphabetize them. Define a *block* to be a set of records that share one or more properties in common and represent blocks as tuples of the form $\langle \textit{block key}, \textit{records} \rangle$ where *records* is the set of records in the block and *block key* is the set of properties those records have in common. A block key is a unique informative name of a set of records. Blocks whose keys contain multiple properties are the intersections of the blocks who have those individual properties as keys. If we have a total orderings of properties we can define a total ordering on block keys by sorting them lexicographically.

We select a small number of *top level properties* like name and social security number to do the initial blocking, and a broader set of *sub-blocking properties* which are used to subdivide oversized sets. The latter may be a superset of the former. Blocks whose keys consist of a single top-level property are called *top-level blocks* and blocks that are the result of further subdivision are called *sub-blocks*.

The algorithm that creates the blocks and sub-blocks takes as input a set of records and a maximum block size M. All the input records are grouped into blocks defined by the top-level properties. Those top-level blocks that are not above the maximum size are set aside. The remaining oversized blocks are partitioned into sub-blocks by sub-blocking properties that the records they contain share, and those properties are appended to the key. The process is continued recursively until all sub-blocks have been whittled down to an acceptable size.

The records in a given block will contain a set of sub-blocking properties, and a complete enumeration of the possible sub-blocks requires enumerating the power set of this property set—intersecting the same-birthday sets with the same-phone number sets and intersecting all those with the same-city sets and so forth. We need an efficient way to

[1] In practice it may be useful to think of properties as functions of the fields actually present in the record. For example, a record may contain separate entries for first and last name but we may prefer to work with a property that is a combination of the two. Or a record may contain a phone number, but we may prefer to work with a property that is just the first three digits.

conduct this exploration, bailing out as soon as possible. To this end, we use the ordering on block keys to define a binomial tree where each node contains a list of block keys and is the parent of nodes that have keys that come later in the ordering appended to the list. Figure 1 shows a tree for the oversize top-level set Smith with sub-block keys for birth year 1971 < first name Robert < city Seattle.

With each node of the tree we can associate a block whose key is the list of blocks keys in that node and whose records are the intersection of the records in those blocks, e.g. the Smith ∩ 1971 ∩ Seattle node represents all the records for people named Smith who were born in 1971 and live in Seattle. Because the cardinality of an intersected set is less than or equal to the cardinalities of the sets that were intersected, every block in the tree is larger than or equal to any of its children. We traverse the tree breadth-first and only recurse into nodes above the maximum block size. This allows us to explore the space of possible sub-blocks in cardinality order for a given branch, stopping as soon as we have a small enough sub-block.

Figure 2 shows an algorithm that performs blocking using this method.

The BLOCK function creates the top-level blocks and each call to the SUB-BLOCK function enumerates one level deeper into the tree. The OVERSIZE function determines if the block is over the maximum size, which in our system is expressed in terms of the maximum number of pairs a block may contain. There is also an additional condition to ignore blocks of size 1 since these will not contain any pairs. Note that we do not know all the sub-block keys that will be present in a given set of records *a priori*. Instead the algorithm discovers this set as it runs.

In the worst case, all the sub-blocks except the ones with the very longest keys are oversize. Then the sub-blocking algorithm will explore the powerset of all possible blocking keys and thus have exponential runtime. However, as the block keys get longer, the sets they represent get smaller and eventually fall beneath the maximum size. In practice these two countervailing motions work to keep this strategy tractable. As will be shown in the experiments, the bulk of the sub-blocks have key lengths of 3 or less.

```
BLOCK(records)
 1  // Do top-level blocking.
 2  blocks = ∅
 3  for record ∈ records
 4       for property ∈ TOP-LEVEL-PROPERTIES(record)
 5            ADD-TO-BLOCKS(blocks, property, record)
 6  // Do sub-blocking.
 7  correct = ∅
 8  for block ∈ blocks
 9       correct = correct ∪ SUB-BLOCK(block)
10  return correct

SUB-BLOCK(block)
 1  if |block.records| == 1
 2       return ∅
 3  if not OVERSIZE(block)
 4       return {block}
 5  blocks = ∅
 6  for record ∈ block.records
 7       for property ∈ SUB-BLOCK-PROPERTIES(record)
 8            if property > block.key
 9                 ADD-TO-BLOCKS(blocks, property, record)
10  correct = ∅
11  for block ∈ blocks
12       correct = correct ∪ SUB-BLOCK(block)
13  return correct

ADD-TO-BLOCKS(blocks, key, record)
1  // Let blocks be a table of blocks indexed by block key.
2  if blocks[key] == ∅
3       blocks[key] = blocks[key] ∪ ⟨key, ∅⟩
4  blocks[key].records = blocks[key].records ∪ record
```

Figure 2: Recursive blocking algorithm

3.3 Scalability

The volume of data involved requires that this system be distributed across a cluster of machines. We make heavy use of the Hadoop implementation of the MapReduce computing framework, and the blocking procedure described here is implemented as a series of Hadoop jobs written in Java. It is beyond the scope of this paper to fully describe this framework (see [13] for an overview), but we do discuss the ways its constraints inform our design.

MapReduce divides computing tasks into a map phase in which the job is split up among multiple machines to be worked on in parallel and a reduce phase in which the output of the map phase is put back together. In a MapReduce context, recursion becomes iteration. A single MapReduce iteration works down one level of the binomial tree of sub-block intersections. Figure 3 shows the MapReduce version of the sub-blocking algorithm.

The mapper functions parallelize the extraction of properties from records performed in the loops beginning in lines

```
TOP-LEVEL-MAPPER(no-blockkey, record)
1  for property ∈ TOP-LEVEL-PROPERTIES(record)
2       EMIT(property, record)

SUB-BLOCK-MAPPER(blockkey, record)
1  for property ∈ SUB-BLOCK-PROPERTIES(record)
2       if property > blockkey
3            EMIT(blockkey + property, record)

BLOCKING-REDUCER(blockkey, records)
 1  buffer = ∅
 2  destination = CORRECT
 3  for record ∈ records
 4       buffer = buffer ∪ records
 5       if OVERSIZE(buffer)
 6            destination = OVERSIZE
 7            break
 8  if |buffer| == 1
 9       return
10  // Dispatch the remaining incoming records.
11  for record ∈ records
12       DISPATCH(blockkey, record, destination)
13  // Dispatch the records in the buffer.
14  for record ∈ buffer
15       DISPATCH(blockkey, record, destination)
```

Figure 3: MapReduce blocking algorithm

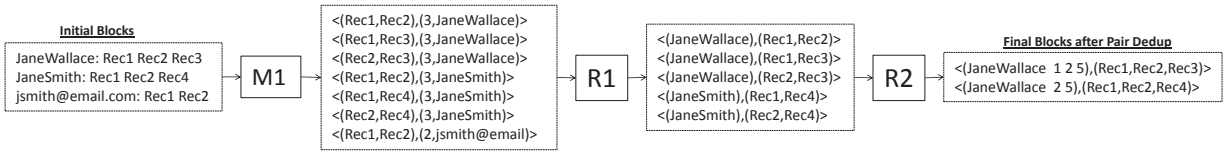

Figure 4: Pair Deduper MapReduce [2]

3 and 6 in the recursive blocking algorithm in Figure 2. The TOP-LEVEL-MAPPER is run on the first iteration and the SUB-BLOCK-MAPPER is run subsequently. The reducer function enumerates over all the records in a newly-created sub-block, counting them to determine whether or not the block is small enough or needs to be further subdivided. The blocks the reducer deems oversized become inputs to the next iteration. Care is taken that the memory requirements of the reducer function are constant in the size of a fixed buffer because otherwise the reducer runs out of memory on large blocks. Also note that the lexicographic ordering of the block keys allows separate mapper processes to work on different nodes in a level of the binomial tree without creating redundant sub-blocks (e.g. if one mapper creates a Robert ∩ Seattle block another mapper will not create a Seattle ∩ Robert one). This is necessary because individual MapReduce jobs run independently without shared memory or other runtime communication mechanisms.

4. PAIR DEDUPING

When dealing with deduplication of large databases in a parallel manner, many blocking approaches allow overlapping blocks in order to increase recall. However, overlapping blocks might cause redundant work for the linkage component by including the same pair across separate blocks as different blocking key combinations might create duplicate or subset blocks from the same database. For example, when we use 'First:Last' name combination and SSN as the blocking keys, some of the SSN blocks might be duplicate or subsets of some of the 'First:Last' blocks (e.g., if records with the same SSN also have the same first and last name)[3].

We deployed a pair deduping module as a phase running after blocking and before pairwise linkage to address this issue. This module is a chain of two MapReduce jobs. Figure 4 represents how this module works. The first job deduplicates all the pairs by retaining each pair with the largest block[4]. The second job, reconstructs the blocks from the retained pairs, and for each block, it generates an edge list representing which pairs in this block should be compared in the linkage component. For efficiency purposes, we represent each edge with an integer equal to $(i * n) + j$ where i and j are the index of the records to be compared and n is the block size.

In order to show the impact of pair deduping, we ran an experiment with 4.8B records. We blocked these records with Dynamic Blocking. Table 1 presents the number of blocks, the total number of records and pairs in these blocks. According to this table, 7.1B blocks are generated by blocking iterations. There are 38.1B records and 162.7B pairs(58.7B unique) inside these blocks. After applying the pair deduping module, all duplicate pairs across the blocks are removed, and this resulted in a 64% gain in run-time during the linkage component. Note that this pair deduping module took around 20 minutes while processing these duplicate pairs in linkage component would take 3 days. Additionally, the number of blocks is reduced to 1.5B blocks since all duplicate and subset blocks are removed. The number of records in all blocks decreased to 12.8B, so, the I/O gain with pair deduping module is around 66%.

	initial	**pair deduped**	**Gain**
#blocks	7,135,917,137	1,508,105,207	
#records	38,114,162,627	12,800,155,261	66% (I/O)
#pairs	162,722,621,924	58,668,501,433	64% (CPU)

Table 1: Effect of Pair Dedup

Note that we also experimented with a naive approach of exploding all blocks into their constituent record pairs and then deduping the pairs. This is trivial to do in MapReduce, but we found that the increased I/O costs from writing out all the pairs cancelled out any efficiency gained by eliminating redundant work in the linkage component. Note that given the scale of our data, we don't have efficient random access to the records during the pairwise linkage component, so the input for the linkage component is a record (not just a record id) array for each block. Exploding a block of size n into its constituent pairs results in n^2 I/O.

5. EXPERIMENTS

We used two different datasets for our experiments. The first one is a small subset of our production data. We used this dataset to make experiments by varying the maximum block size. We also used a second dataset which has around 5 billion public people records to demonstrate the viability of this algorithm for very large data sets.

First, we ran experiments on 5,680,599 records sampled from our full data set of approximately 5 billion. To build this sample we used all records that contained a first-last name pair that appeared on a list of 262,005 such pairs randomly chosen from our data set. We did name-based sampling instead of random sampling because the vast majority of record pairs in the full data set are not matches. By choosing a broadly individuating property like first and last name we choose a set of records that will have more matches, focusing these experiments on a region of the problem space where blocking is having an effect.

We ran blocking, linkage, and clustering on this sample of data for a range of maximum block size parameters. We used three top level properties: 1) first and last name, 2) social security number and 3) a tuple of last name, zip code,

[2]Since the mapper of the second job is an identity mapper which outputs the input, we excluded it from the figure.

[3]This might also happen when there are multiple values in each record for a specific blocking key. For example, a person might have multiple names and addresses which will cause this record to appear in different name or address blocks when using name or address as blocking key.

[4]If there are multiple blocks with same size, the pair is retained in the block whose key is lexicographically greater.

and house number. We used various sub-properties including various kinds of name and address information. The maximum block size ranged from 20 to 50. We also ran an experiment by using the traditional blocking approach with the same blocking keys. In traditional blocking, there is no limit on block size.

There are approximately 10^{12} record pairs in this dataset, making it infeasible to construct a hand-annotated reference set. However, when the linkage model is held constant, blocking does not change the score the linkage model assigns to any individual pair, only the number of pairs presented to it. In other words, blocking only affects recall, not precision. For these experiments, then, we treat the results returned by the linkage component as "truth". If the model determines that the pair should be matched, we treat the pair as a true positive, otherwise we treat it as a true negative. Although the pairwise model does, of course, make false positive errors, we treat this as the fault of the linkage component, not blocking whose job is solely to present the model with the record pairs it will tend to score the highest.

So, we used three different metrics for the evaluations, namely compression, pair quality, and total runtime. First, we define the *compression* metric as:

$$c = 1 - \frac{\text{output entities}}{\text{input records}} \tag{1}$$

We present compression as a percentage. The higher the compression the better recall we have. Compression does not take precision into account; however, as long as the model is doing something reasonable, compression is a good proxy for recall. Secondly, the pair quality metric is the ratio of true positive pairs to all unique pairs generated by blocking.

$$pq = \frac{\text{true positive pairs}}{\text{all unique pairs generated by blocking}} \tag{2}$$

Higher pair quality for a blocking approach shows that this approach is more efficient and generates mostly true matched candidate pairs while a lower pair quality value means a large number of non-matches are also generated [7]. Last but not the least, we used the total runtime for blocking and linkage components as a metric.

$$rt_{total} = rt_{blocking} + rt_{linkage} \tag{3}$$

Since the number of candidate pairs generated by blocking affects the runtime of the linkage component, we don't take into account just the runtime of blocking, and rather prefer looking at the overall runtime for blocking and linkage.

Table 2 presents the compression, pair quality, total runtime values for *Dynamic Blocking*, and *Traditional Blocking*. According to this table, *Traditional Blocking* yields the highest compression ratio. However, its pair quality is much lower than that of *Dynamic Record Blocking*. Additionally, the total runtime for this approach is around 4 times higher than the total runtime of *Dynamic Blocking* with a maximum block size 40. Moreover, while maximum block size increases for *Dynamic Record Blocking*, we have higher compression. This is to be expected because increasing the block size parameter increases the number of pairs that will be considered by the linkage component. But, the increase in compression is not directly proportional to the increase in total runtime. According to our experiments, the increase in total run time is more likely to be quadratic. Therefore,

	Block Size				Traditional
	20	**30**	**40**	**50**	**Blocking**
Compression	83.25	83.81	83.98	84.05	85.09
Pair Quality	0.65	0.59	0.52	0.44	0.13
Total Runtime	x	1.05x	1.21x	1.65x	4.83x

Table 2: Experimens with various Block Sizes

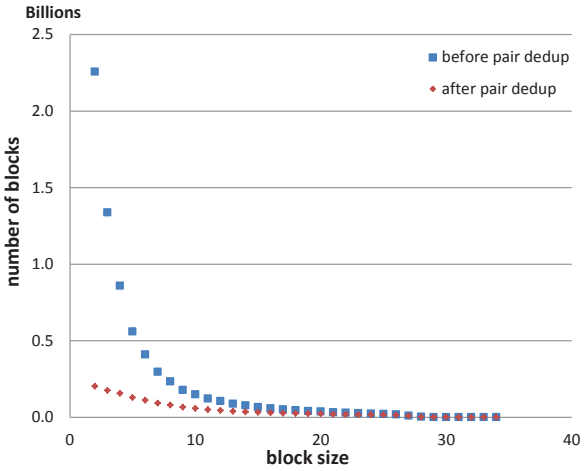

Figure 5: Block size distribution.

we need to keep the maximum block size at a certain level in order to complete all linkage processes in a reasonable amount of time.

We also used a dataset of around 5 billion public people records to demonstrate the viability of this algorithm for very large data sets. We set the maximum block size to 35 and blocked all 5 billion records with the *Dynamic Blocking* approach. The blocking step took around 1.5 days to complete. As shown in Table 1, around 1.5B blocks along with 58B candidate pairs were produced for the pairwise linkage component. Figure 5 shows the size distribution of blocks before and after pair deduping them. Since we keep the duplicate pairs that occur in the larger blocks, we remove mostly smaller blocks. After running the linkage component, which took around 5 days, we had around 84.7% compression. This compression was in line with the results on the smaller dataset.

6. CONCLUSION AND FUTURE WORK

We have developed a novel blocking technique for duplicate record detection that operates on the intuitive notion of grouping together records with similar properties and then subdividing the groups using other shared properties until they are all of tractable size. We have implemented this algorithm in the MapReduce framework so that it may scale to inputs in the billions of records.

One area of future work is in text normalization. Unlike other blocking systems we have not devoted much effort towards directly addressing errors in the original data, relying instead on multiple top-level keys to catch mis-blockings due to typos. However, we plan to investigate the effect of text normalization for our system. There is no reason why our framework could not incorporate additional record properties such as, for example, Soundex versions of text fields.

7. ACKNOWLEDGMENTS

The authors would like to express thanks to Xin Wang, Vitor Carvalho, Ken Goodhope, Mike D'Angelo, Søren Flexner, Jim Adler, Jean-Remy Facq, and the Data Team at Intelius for building the infrastructure that made this work possible. Particular thanks to Siddharth Agrawal and Chase Bradford for their contributions to blocking scalability.

8. REFERENCES

[1] N. Adly. Efficient record linkage using a double embedding scheme. In R. Stahlbock, S. F. Crone, and S. Lessmann, editors, *DMIN*, pages 274–281. CSREA Press, 2009.

[2] A. N. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39. IEEE Computer Society, 2005.

[3] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage, 2003.

[4] M. Bilenko and B. Kamath. Adaptive blocking: Learning to scale up record linkage. In *Data Mining, 2006. ICDM'*, number December, 2006.

[5] A. Borthwick, A. Goldberg, P. Cheung, and A. Winkel. Batch automated blocking and record matching, 2005. U.S. Patent #7899796.

[6] S. Chen, A. Borthwick, and V. R. Carvalho. The case for cost-sensitive and easy-to-interpret models in industrial record linkage. In *9th International Workshop on Quality in Databases*, August 2011.

[7] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2011.

[8] T. de Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 305–314, New York, NY, USA, 2009. ACM.

[9] T. de Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays and bloom filters. *ACM Trans. Knowl. Discov. Data*, 5(2):9:1–9:27, Feb. 2011.

[10] A. K. Elmagarmid, P. G. Iperirotis, and V. S. Verykios. Duplicate record detection: A survey. In *IEEE Transactions on Knowledge and Data Engineering*, pages 1–16, 2007.

[11] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty. data cleansing and the merge/purge problem. *Journal of Data Mining and Knowledge Discovery*, pages 1–39, 1998.

[12] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, DASFAA '03, pages 137–, Washington, DC, USA, 2003. IEEE Computer Society.

[13] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Langugage Technologies. Morgan & Claypool, 2010.

[14] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the ACM International Conference on Knowledge Discover and Data Mining*, pages 169–178, 2000.

[15] J. Nin, V. Muntes-Mulero, N. Martinez-Bazan, and J.-L. Larriba-Pey. On the use of semantic blocking techniques for data cleansing and integration. In *Proceedings of the 11th International Database Engineering and Applications Symposium*, IDEAS '07, pages 190–198, Washington, DC, USA, 2007. IEEE Computer Society.

[16] A. D. Sarma, A. Jain, and A. Machanavajjhala. CBLOCK: An Automatic Blocking Mechanism for Large-Scale De-duplication Tasks. Technical report, 2011.

[17] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. *Proc. VLDB Endow.*, 1(2):1253–1264, Aug. 2008.

[18] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 219–232, New York, NY, USA, 2009. ACM.

[19] W. Winkler. Overview of record linkage and current research directions. Technical report, U.S. Bureau of the Census, 2006.

[20] S. Yan, D. Lee, M.-Y. Kan, and L. C. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '07, pages 185–194, New York, NY, USA, 2007. ACM.