Scalable Blocking for Very Large Databases

Andrew Borthwick*, Stephen Ash*, Bin Pang, Shehzad Qureshi, and Timothy Jones

AWS AI Labs {andborth,ashstep,pangbin,shezq,timjons}@amazon.com

Abstract. In the field of database deduplication, the goal is to find approximately matching records within a database. *Blocking* is a typical stage in this process that involves cheaply finding candidate pairs of records that are potential matches for further processing.We present here *Hashed Dynamic Blocking*, a new approach to blocking designed to address datasets larger than those studied in most prior work. Hashed Dynamic Blocking (HDB) extends Dynamic Blocking, which leverages the insight that rare matching values and rare intersections of values are predictive of a matching relationship. We also present a novel use of Locality Sensitive Hashing (LSH) to build blocking key values for huge databases with a convenient configuration to control the trade-off between precision and recall. HDB achieves massive scale by minimizing data movement, using compact block representation, and greedily pruning ineffective candidate blocks using a Count-min Sketch approximate counting data structure. We benchmark the algorithm by focusing on real-world datasets in excess of one million rows, demonstrating that the algorithm displays linear time complexity scaling in this range. Furthermore, we execute HDB on a 530 million row industrial dataset, detecting 68 billion candidate pairs in less than three hours at a cost of $307 on a major cloud service.

Keywords: Duplicate detection · blocking · entity matching · record linkage

1 Introduction

Finding approximately matching records is an important and well-studied problem [12]. The challenge is to identify records which represent the same real-world entity (e.g. the same person, product, business, movie, etc.) despite the fact that the corresponding data records may differ due to various errors, omissions, or different ways of representing the same information.For many database deduplication/record linkage applications a common approach is to divide the problem into four stages [17]: Normalization [2], Blocking [7, 19, 24], Pairwise Matching [20, 6], and Graph Partitioning [14, 25].

This work focuses on the problem of blocking very large databases with record counts between 1M and 530M records. We focus on databases in this range due

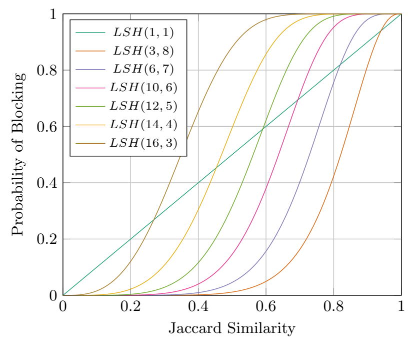
* Equal Contribution

to their importance in industrial settings. We make a case that different blocking algorithms will be successful in this range than are effective on databases with fewer than 1 million records. The experimental results that we show on real-world datasets with 50M or more records, in particular, are unusual in the literature.

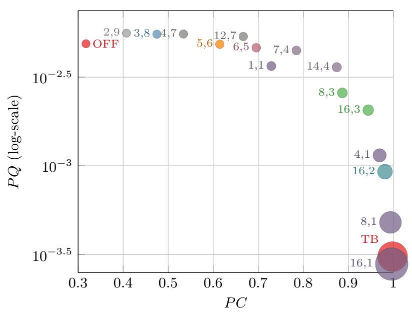
In contrast to prior work such as [3] which seeks to build an optimal set of fields on which to block records, the philosophy of the dynamic blocking family of algorithms [19] is to avoid selecting a rigid set of fields and instead dynamically pick particular *values* or combinations of values on which to block. As an example of blocking on a fixed, static set of blocking key fields, consider a system to deduplicate U.S. person records by simply proposing all pairs of persons who match on the field *last_name*. This would be prohibitively expensive due to the necessity of executing a pairwise matching algorithm on each of the $\binom{1,400,000}{2}$ pairs of people who share the surname "Jones". A pairwise scoring model averaging 50 μsecs would take ≈ 567 days to compare all "Jones" pairs.

On the other hand, suppose that we statically select the pair of fields *(first_name, last_name)* as a single blocking key. This solves the problem of too many "Jones" records to compare, but is an unfortunate choice for someone with the name "Laurence Fishburne" or "Shehzad Qureshi". Both of these surnames are rare in the U.S. A static blocking strategy which required both given name and surname to match would risk missing the pair ("laurence fishburne","larry fishburne") or ("shehzad qureshi","shezad qureshi"). Differentiating between common and less common field/value pairs in the blocking stage fits with the intuition that it is more likely that two records with the surname "Fishburne" or "Qureshi" represent the same real-world individual than is the case for two records with the surname "Jones", which is an intuition backed up by algorithms that weight matches on rare values more strongly than matches on common values [6, 17, 28].

This work makes the following contributions: (1) We describe a new algorithm called Hashed Dynamic Blocking (HDB) based on the same underlying principle as dynamic blocking [19], but achieves massive scale by minimizing data movement, using compact block representation, and greedily pruning ineffective candidate blocks. We provide benchmarks that show the advantages of this approach to blocking over competing approaches on huge real-world databases. (2) Our experimental evidence emphasizes very large real-world datasets in the range of 1M to 530M records. We highlight the computational complexity challenges that come with working at this scale and we demonstrate that some widely cited algorithms break down completely at the high end. (3) We describe a version of Locality Sensitive Hashing applied to *blocking* that is easily tunable for increased precision or increased recall. Our application of LSH can generate (possibly overlapping) blocking keys for multiple columns simultaneously, and we provide empirical evaluation of LSH versus Token Blocking to highlight the trade-offs and scaling properties of both approaches.



(a) Probability of blocking vs Jaccard of the record pair's attribute under various $LSH(b, w)$ settings



(b) PQ vs PC on SCHOLAR under various LSH(b, w) settings. The diameter of the point relates to the number of pairs produced.

Fig. 1: PQ and PC of various LSH settings on the SCHOLAR dataset

2 LSH and block building

Like most other blocking approaches such as Meta-blocking [21], dynamic blocking begins with a set of records and a *block building* step that computes a set of *top-level* blocks, which is a set of records that share a value computed by a block blocking process, t, where t is a function that returns a set of one or more *blocking keys* when applied to an attribute a_k of a single record, r. The core HDB algorithm described in Section 3 is agnostic to the approach to block building.

With structured records, one can use domain knowledge or algorithms to pick which block building process to apply to each attribute. We term *Identity Block Building* as the process of simply hashing the normalized (*e.g.* lower-casing, etc.) attribute value concatenated to the attribute id to produce a blocking key. Thus the string "foo" in two different attributes returns two different top-level blocking keys (*i.e.* hash values). For attributes where we wish to allow fuzzier matches to still block together, we propose *LSH Block Building* as described in the next section. Alternatively, *Token Blocking* [21] is a schema-agnostic block building process where every token for every attribute becomes a top-level blocking key. Note that, unlike Identity Blocking Building, the token "foo" in two different attributes will return just a single blocking key.

2.1 LSH block building

In this work we propose a new block building approach which incorporates Locality Sensitive Hashing (LSH) [15, 13] with configurable parameters to control the precision, recall trade-off per column. LSH block building creates multiple sets of keys that are designed to group similar attribute values into the same block. We leverage a version of the algorithm which looks for documents with high degrees of Jaccard similarity [18, 27] of tokens. Here a *token* could be defined as a *word*, a word n-gram, or as a character q-gram.

[18] describes a method in which, for each document d, we first apply a min-hash algorithm [5] to yield m minhashes and we then group these minhashes into b *bands* where each band consists of $w = m/b$ minhashes. In our approach each of these bands constitutes a blocking key. Now consider a function $LSH(b, w, j)$, in which b and w are the LSH parameters mentioned above and j is the Jaccard similarity of a pair of records, then $LSH(b, w, j)$ is the probability that the attributes of two records with Jaccard similarity of j will share at least one key and can be computed as: $LSH(b, w, j) = 1-(1 - j^w)^b$ $LSH(b, w, \cdot)$ has an attractive property in that the probability of sharing a key is very low for low Jaccard similarity and very high for high Jaccard similarity. Figure 1 graphs $LSH(b, w, \cdot)$ for various values of (b, w), which gives us a range of attractive trade-offs on the Pair-Quality (*i.e.* precision) versus Pair-Completeness (*i.e.* recall) curve by varying the two parameters for LSH, b and w.

2.2 Prior Work on Block Building

Our block building techniques are strongly distinguished from prior work in the field. For example, [9] makes an assumption that the block-building phase yields a strict partitioning of the database, although multiple distinct passes can be used to reduce false negatives [9, 12]. Our approach, by contrast leverages the fact that the block building strategies discussed in Section 2 yield blocks that are, by design, highly overlapping and all the blocks are processed in a single pass. Another distinction is that [9] generates a single minHash on a single attribute from which it builds blocks, which in terms of our approach would correspond to using a degenerate LSH with the parameters of $b = 1, w = 1$ on only a single column. Figure 1 includes this $LSH(1, 1)$ configuration, which highlights its particular point in the precision, recall curve for the SCHOLAR dataset. In this case, our $LSH(14, 4)$ improves recall by $\approx 20\%$ with only $\approx 1\%$ change in precision.

Use of LSH is fairly common in the literature. [23] has an extensive recent survey. [18] has a useful tutorial on LSH that uses deduplication as an example. However, our approach is new in that we do not apply LSH to the record as a whole, but rather LSH is applied selectively to columns where it makes sense (*e.g.* columns consisting of multi-token text), while columns consisting of scalar values generate top-level blocks through trivial identity block building.

Using the nomenclature of a recent survey on blocking [23], our block building strategy yields top-level blocks which are neither *redundancy-free*, where every entity is assigned to exactly one block, nor is it *redundancy-positive*, in which every entity is assigned to multiple blocks, such that the "more blocks two entities share, the more similar their profiles are likely to be". The latter is due to the fact that by construction LSH bands act as redundant blocking keys that do connote some similarity (*e.g.* higher jaccard) but less similarity than co-occurring non-LSH keys.

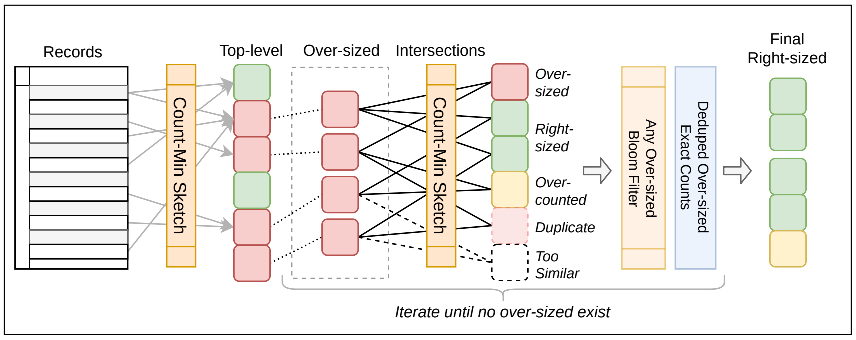


Fig. 2: Diagram illustrating how candidate blocks are processed in Hashed Dynamic Blocking

3 Hashed Dynamic Blocking

Dynamic blocking [19] takes the approach of *finding* overlapping subsets of records that share enough blocking key values in common to make the size of each subset small. We impose this subset size threshold as a way to balance the precision and recall of each emitted subset of records. At each iteration of the algorithm, we partition the blocking keys between those which are above and below this threshold: `MAX_BLOCK_SIZE`. Those below the max block size are deemed to be *accepted* or right-sized. For all accepted blocks b_i, the blocking phase is over. That is, later in the pairwise matching phase, we will compare all $\binom{|b_i|}{2}$ pairs of records from all right-sized blocks. However, for over-size blocks, we need to find additional co-occurring blocking key values. To do this we compute the logical intersections of all pairs of over-size blocks. For example, if the "Jones" block described in Section 1 had more than `MAX_BLOCK_SIZE` records, it could be intersected with other blocks generated by values that also had more than `MAX_BLOCK_SIZE` records such as the block for the first_name "Tim". The logical intersection of these two blocks may be under the threshold. We progressively intersect blocking key *values* in this way until we have no over-size blocks left. As described below, we use a few heuristics to guide this search for right-sized intersections to make this converge efficiently.

3.1 Algorithm detailed description

Algorithm 1 describes the high level algorithm of Hashed Dynamic Blocking (HDB). HDB is implemented in Apache Spark[1]. For clarity we show the pseudocode in an imperative style, but in the implementation everything is implemented as a sequence of lazy map and reduce operations. In particular, we use the keyword **parallel for** to indicate which loops are actually `map` operations (as in the MapReduce paradigm).

[1] https://spark.apache.org/

Algorithm 1 Hashed Dynamic Blocking

```
Input: $R$, a dataset of records, $r$, to be blocked, with each record having a long identifier, $rid$
Output: a *blocked* dataset of deduplicated record pairs
1: function HashedDynamicBlocking($R$)
2:     $K \leftarrow$ BlockOnKeys($R$)
3:     $(K_R, \tilde{K}_O) \leftarrow$ RoughOversizeDetection($K$)
4:     $(\hat{K}_R, K_O) \leftarrow$ ExactlyCountAndDedupe($\tilde{K}_O$)
5:     $K_R \leftarrow K_R \cup \hat{K}_R$
6:     while $K_O \neq \emptyset$ do
7:         $K' \leftarrow$ IntersectKeys($K_O$)
8:         $(K'_R, \tilde{K}'_O) \leftarrow$ RoughOversizeDetection($K'$)
9:         $(\hat{K}'_R, K'_O) \leftarrow$ ExactlyCountAndDedupe($\tilde{K}'_O$)
10:        $K_R \leftarrow K_R \cup \hat{K}'_R$
11:        $K_O \leftarrow K'_O$
12:    end while
13:    return RemoveDupePairs($K_R$)
14: end function
```

In HashedDynamicBlocking, first, the function BlockOnKeys applies the configured block building functions to the input dataset, R, as described in Section 2. The resulting dataset, K, is an inverted index of blocking keys for each record ID. It is important to note that we do *not* maintain or materialize the *flipped* view of block key to list of record IDs in that block during these iterations. Materializing this view is relatively expensive as it requires shuffling the data across the cluster, and thus we only do it once at the end after all of the right-sized blocks have been determined. HDB operates exclusively on 64-bit record IDs and a sequence of hashed blocking keys represented by 128-bit hash values derived from the top-level blocking attribute values. After top-level block building, the actual attributes of the records are no longer needed and do not flow through the algorithm.

Figure 2 visualizes the logical steps of how we identify right-sized and over-sized blocks. In each iteration, we lazily intersect the previously identified over-size blocking keys. We use fast approximate counting in RoughOversizeDetection (Algorithm 3) to quickly identify which of these are right-sized blocks, K_R, and possibly over-sized blocks, $\tilde{K}_O$.

Our probabilistic counting data structure *might* over-count, and therefore some of the identified blocks in $\tilde{K}_O$ may in fact be right-sized. The function ExactlyCountAndDedupe (Algorithm 4) post-processes $\tilde{K}_O$ to accurately identify any over-counted right-sized blocks, $\hat{K}_R$. This function also de-duplicates the over-sized blocks efficiently, resulting in the final true, unique set of over-sized blocks left over, K_O, for the next iteration. After all iterations, the set of resultant right-sized blocks may have some duplicate pairs. Thus in function RemoveDupePairs, we remove duplicate pairs as described in Section 3.1.

Algorithm 2 Intersecting Blocking Keys

```
Input: $K$, a dataset of $rid$ to blocks, $b_{0..n}$, where $b_i$ is a 2-tuple of the block key hash,
    $b.key_i$, and the count of records in this block $b.size_i$
Output: $R$, a dataset of $rid$ to intersected blocks, $b_{0..n}$, where $b_i$ is a 2-tuple of the
    block key hash, $b.key_i$, and the count of records in the parent block $b.psize_i$
 1: function IntersectKeys($K$)
 2:     $K \leftarrow \{(rid, b_{0..n}) \mid (rid, b_{0..n}) \in K \wedge n \leq \texttt{MAX\_KEYS}\}$
 3:     $R \leftarrow \emptyset$
 4:     parallel for $(rid, b_{0..n}) \in K$ do
    ▷ intersect all block keys, producing $\binom{n}{2}$ new block keys
 5:         $P \leftarrow \{b_i, b_j \mid b_i \in b, b_j \in b \wedge b_i < b_j\}$
 6:         parallel for $(b_i, b_j) \in P$ do
 7:             $x.key \leftarrow \textsc{Murmur3}(b.key_i, b.key_j)$
    ▷ the new block's size is unknown at this point, but we carry the smallest parent's
    size
 8:             $x.psize \leftarrow min(b.size_i, b.size_j)$
 9:             $R[rid] \mathrel{+}= x$
10:         end for
11:     end for
12:     return $R$
13: end function
```

Intersecting Keys Algorithm 2 shows how we compute what is semantically a pair-wise intersection of every over-sized block by local operations. The inverted index of blocking keys, K, accepted by IntersectKeys, is logically a map of record ID rid to the set of over-sized blocking keys $b_{0..n}$ for that record. The blocking keys here are represented as a 2-tuple of $(b.key_i, b.size_i)$, where $b.key_i$ is the blocking key hash value and $b.size_i$ is the number of records that share the blocking key $b.key_i$ (as computed by ExactlyCountAndDedupe).

We discard from further processing all records which have more than `MAX_KEYS` blocking keys (line 2.2) as a guard against a quadratic explosion of keys. The governing hypothesis of dynamic blocking is that, since all blocking keys cover a distinct set of record IDs, this quadratic increase in the number of keys will be counterbalanced by the fact that $|A \cap B| < |A| + |B|$ and thus the intersected blocks will tend to become right-sized. Furthermore, records which have a large number of keys on iteration i are likely to have participated in many right-sized blocks on iterations prior to i.

On lines 2.5 to 2.9, we replace the existing, over-sized block keys with $\binom{n}{2}$ new hashes computed by combining every pair of existing over-sized hashes for that record. There are some blocking key intersections which we do *not* want to produce. For example, if a dataset had 4 nearly identical over-sized blocks, then after the first intersection these 4 columns would intersect with each other to produce $\binom{4}{2} = 6$ columns, but since these were already over-sized and nearly identical, the intersected columns would be over-sized as well. This quadratic growth of over-sized blocking keys per record would not converge. To avoid this hazard, we apply a *progress heuristic* and only keep blocking key intersections

Algorithm 3 Rough Over-sized Block Detection

```
Input: $K$, a dataset of $rid$ to over-sized blocks, $b_{0..n}$ where $b_i$ is a 2-tuple of the block
    key hash, $b.key_i$, and the count of records in the parent block, $b.psize_i$
Output: $K_R$, a dataset of record to right-sized blocks
Output: $\tilde{K}_O$, a map of record to possibly over-sized blocks
 1: function RoughOversizeDetection($K$)
 2:     $cms \leftarrow$ ApproxCountBlockingKeys($K$)
 3:     $K_R \leftarrow \emptyset$
 4:     $\tilde{K}_O \leftarrow \emptyset$
 5:     parallel for $(rid, b_{0..n}) \in K$ do
 6:         for all $b_i \in b$ do
 7:             $s \leftarrow cms[b.key_i]$
 8:             $p \leftarrow b.psize_i$
 9:             if $s \leq$ MAX_BLOCK_SIZE then
10:                 $K_R[rid] \mathrel{+}= b_i$                        ▷ right-sized
11:             else if $(s/p) \leq$ MAX_SIMILARITY then
12:                 $\tilde{K}_O[rid] \mathrel{+}= b_i$                        ▷ over-sized
13:             end if
    ▷ We discard over-sized blocks that are too similar in size to parent
14:         end for
15:     end for
16:     return $K_R, \tilde{K}_O$
17: end function
```

that reduce the size of the resulting blocks by some fraction, MAX_SIMILARITY. This heuristic filter is applied in Algorithm 3, using the minimum *parent* block size which we propagate on line 2.8.

Rough Over-sized Block Detection It is critical that we can accurately count the size of each block. A naïve approach would be to pivot from our inverted index to a view of records per blocking key, at which point counting block sizes is trivial. This requires expensive global *shuffling* of all of the data across the cluster in each iteration of the blocking algorithm. Our approach makes novel use of a Count-Min Sketch (CMS) [10] data structure to compute an approximate count of the cardinality of every candidate blocking key (line 3.2). We compute one CMS per data partition and then efficiently merge them together. Due to the semantics of a CMS, the approximate count will never be *less* than the true count, and thus no truly over-sized blocks can be erroneously reported as right-sized. In this way, the CMS acts as a filter, dramatically reducing the number of candidate blocks that we need to focus on in each iteration.

Exactly Count and Deduplicate Algorithm 4 focuses on correcting the possibly over-sized blocks. The goal of this method is to partition the $\tilde{K}_O$ blocking keys in the inverted index into three sets, illustrated in Figure 2: (1) **right-sized blocks** that were erroneously over-counted by the Count-Min Sketch, $\hat{K}_R$, which

Algorithm 4 Correct over-counting and deduplicate blocks

Input: $\tilde{K}_O$, a dataset of rid to $b_{0..n}$ where b_i are block key hashes for r

Output: $\hat{K}_R$, a dataset of record to right-sized blocks that were erroneously over-counted by the Count-min Sketch

Output: K_O, a dataset of truly over-sized blocks, rid to $b_{0..n}$ where b_i is a 2-tuple of the block key hash, $b.key_i$, and the count of records in this block, $b.size_i$

```
 1: function ExactlyCountAndDedupe($\tilde{K}_O$)
 2:     $\hat{K}_R \leftarrow \emptyset$
 3:     $K_O \leftarrow \emptyset$
 4:     $H \leftarrow$ CountKeysAndXORRids($\tilde{K}_O$)
    ▷ $H$ is dataset of block key hash to tuple of (XOR, size)
 5:     $H_O \leftarrow \{h \mid h \in H \land h.size >$ MAX_BLOCK_SIZE$\}$
 6:     $H_U \leftarrow$ DropDuplicates($H_O$)                    ▷ based on XOR
    ▷ $counts$ is a broadcasted map of deduped, true over-sized counts
 7:     $counts \leftarrow$ BroadcastCounts($H_U$)
 8:     $bloom \leftarrow$ BuildBloomFilter($H_O$)
 9:     parallel for $(rid, b_{0..n}) \in \tilde{K}_O$ do
10:         for all $b_i \in b$ do
11:             if $b.key_i \notin bloom$ then
12:                 $\hat{K}_R[rid] \mathrel{+}= b_i$                 ▷ over-counted
13:             else if $b.key_i \in counts$ then
14:                 $x.key \leftarrow b.key_i$
15:                 $x.size \leftarrow counts[b.key_i]$
16:                 $K_O[rid] \mathrel{+}= x$                   ▷ over-sized
17:             end if
    ▷ keys in $bloom$ but not in $counts$ were duplicate over-sized blocks, which we discard
18:         end for
19:     end for
20:     return $\hat{K}_R, K_O$
21: end function
```

we subsequently union into this iteration's right-sized blocks (line 1.10); (2) **duplicate** over-sized blocks, which we discard; (3) **surviving, deduplicated** over-sized blocks, K_O, with precise counts of how many records are in each, which are then further intersected in the next iteration.

Block A duplicates block B if block A's record IDs are equal to block B's. We arbitrarily discard duplicate blocks, leaving only a single surviving block from the group of duplicates, in order to avoid wasting resources on identical blocks that would only continue to intersect with each other, but produce no new pair-wise comparisons. We do this exact count and dedup in parallel in one map-reduce style operation. To *deduplicate* the blocks we build a *block membership* hash key by hashing each record ID in the candidate block and bit-wise `XOR`ing them together. Since `XOR` is commutative, the final block membership hash key is then formed (reduced) by `XOR`ing the partial membership hash keys.

On line 4.6, we discard duplicate copies of blocking keys that have the same block membership hash key. From these deduplicated blocking keys, H_U, we create a string multiset, *counts*, to precisely count the over-sized blocking keys. Even in our largest dataset of over 1 billion records, the largest count of oversized blocks in a particular iteration after deduplication is ≈2.6M which easily fits into memory, but if this memory pressure became a scaling concern in the future, we could use another Count-Min Sketch here.

Lastly, we need to distinguish the erroneously over-counted blocks which are actually right-sized, $\hat{K}_R$, from the surviving, deduplicated blocks, H_U. On line 4.8 we build a Bloom filter [4] over *all* of the over-sized blocking keys, H_O, which contains both duplicate and surviving over-sized blocks as determined by precise counting. Therefore, the Bloom filter answers the set membership question: is this blocking key possibly over-sized? In this way, we use this filter as a mechanism to detect right-sized blocks that were erroneously over counted. We build the Bloom filter using a large enough bit array to ensure a low expected false positive rate of 1e−8. Even in our largest dataset, the biggest Bloom filter that we have needed is less than ≈100MB.

Pair Deduplication The final set of right-sized blocks determined after k iterations of Hashed Dynamic Blocking will likely contain blocks that overlap or are entirely subsumed by other blocks. We use a map-reduce sequence to compute all distinct pairs similar to the pair deduplication algorithm presented in [19], retaining only the pair from the *largest* block in the case of duplicates. This results in tuples $(rid_1, rid_2, b.key_i)$ where $b.key_i$ is the identifier for the *largest* block that produced the pair (rid_1, rid_2). We then group the tuples by $b.key_i$ to reconstruct the blocks. For each block, we now have an edgelist of $[1, \binom{n}{2}]$ pairs and have the complete set of n resulting record IDs. We build a bitmap of $\binom{n}{2}$ bits with each representing one pair for pairwise matching. The bit index $b_{i,j}$ for a pair of record IDs rec_a, rec_b is computed by: $b_{i,j} = i*(n-1)-(i-1)*i/2+j-i-1$, where i, j are the zero-based *indexes* of rec_a, rec_b in the block of n records, ordered by the record IDs natural order and $i < j$. This is simply a sequential encoding of the strictly upper triangular matrix describing all $\binom{n}{2}$ pairs in the block. In the common case where none of the $\binom{n}{2}$ pairs are filtered out, we omit the bitmap and just score all pairs from the block during pairwise matching.

4 Prior Work

4.1 Prior work on Dynamic Blocking

The need for Hashed Dynamic Blocking may be unclear since its semantics (the pairs produced after pair deduplication) are essentially the same as that of [19] for scalar-valued attributes. Relative to [19], this work offers the following advantages: (1) [19] had a substantial memory and I/O footprint since the content of the records being blocked had to be carried through each iteration of the algorithm. (2) LSH would have been challenging to implement in the Dynamic Blocking algorithm of [19] as it did not contemplate blocking on array-valued columns.

Table 1: Datasets used for experiments where BB indicates (L)SH or (T)oken block building strategy and positive labels marked with † are complete ground truth. Datasets marked C are Commercial datasets.

Moniker	Records	+Labels	Cols	BB	Src
VAR1M	1.03M	818	60	L	C
VAR10M	10.36M	8,890	60	L	C
VAR25M	25.09M	20,797	60	L	C
VAR50M	50.02M	40,448	60	L	C
VAR107M	107.58M	80,068	60	L	C
VAR530M	530.73M	76,316	60	L	C

Moniker	Records	+Labels	Cols	BB	Src
VOTER	4.50M	53,653†	108	L	[1]
SCHOLAR	64,263	7,852†	5	L	[16]
CITESR	4.33M	558k†	7	L	[26]
DBPEDIA	3.33M	891k†	—	T	[11]
FREEB	7.11M	1.31M†	—	T	[11]

4.2 Meta-Blocking based approaches

Meta-Blocking [21], like dynamic blocking, starts from an input collection of blocks and is independent of the scheme for generating these blocks. It encompasses a broad variety of techniques, but at its most basic, it builds a graph in which a node corresponds to a record and an edge (e_1, e_2) indicates that at least one block contains both e_1 and e_2.

A shortcoming of the Meta-Blocking family of algorithms is that it is linear in the number of comparisons in the input block collection [11], which would be equivalent to the total number of comparisons implied by all blocks (both *over-sized* and *right-sized*) in the input block collection. Meta-Blocking approaches generally mitigate this linearity by purging the very largest blocks [11] and by Block Filtering [22] which, for each entity, trims the entity from the largest blocks in which it participates. However, [11] reports only an "at least 50%" reduction in the number of pairwise comparisons using Block Filtering, leaving the algorithm still linear in the comparisons of the input block collection.

Our approach, by contrast, aims to leverage rather than trim large blocks by intersecting them with other large blocks. This is better than either discarding or trimming the large blocks, which may sacrifice recall, or attempting to do even a minimal amount of pairwise processing on the large blocks, which would impact performance.

BLAST [26] is a schema-aware meta-blocking approach. One innovation of BLAST is making records that share high entropy attributes like *name* more likely to be pairwise-compared than low entropy attributes like *year of birth*. HDB, as noted above, takes this approach one step further by making rare values (*e.g.* surname *Fishburne*) more likely to create pairs than common values (*e.g.* surname *Jones*).

5 Experimental Results

We present experimental results to explore a few different aspects of Hashed Dynamic Blocking: (1) we present metrics illustrating the overall performance of HDB on a diverse collection of datasets compared to two different baselines: (a)

Threshold Blocking (THR) and (b) Parallel Meta-blocking[2] (PMB) [11]. Threshold Blocking refers to blocking based on field values (as in HDB and PMB), but if a block is too large (records > 500) then the block is discarded entirely. This simple baseline is useful in illustrating the value of *dynamic* blocking as a means to discover co-occurring values that are discriminating enough to warrant all pairs comparison. (2) we demonstrate the impact of using LSH-based block building with varying parameter values of b bands and w minhashes per band. Unless mentioned otherwise, we use hyper-parameters `MAX_BLOCK_SIZE` $= 500$, `MAX_KEYS` $= 80$, and `MAX_SIMILARITY` $= 0.9$.

We run all of our experiments using AWS ElasticMapReduce (EMR) using Spark 2.4.3 and 100 m4.4xlarge core nodes with 20GB of executor memory and 16 cores per executor. At the time of writing m4.4xlarge instances on AWS cost \$1.04/hour (on EMR). Our largest dataset of 530M records takes 169 minutes to complete blocking costing ≈\$307.

5.1 Datasets

In order to evaluate the effectiveness of Hashed Dynamic Blocking, we evaluate against a diverse collection of datasets. Table 1 lists summary information for each dataset used for evaluation. The **VARxx** datasets are product variations datasets that come from a subset of a large product catalog from a large E-commerce retailer. Each **VARxx** record contains sparsely populated fields such as product name, description, manufacturer, and keywords. We include the bibliographic citation dataset *DBLP-Scholar*, called **SCHOLAR**, from [16] as it is small enough to practically illustrate the differences between LSH-based block building with many different configurations. We include the DBLP-Citeseer bibliographic citation dataset, **CITESR**, from [26]. We also include two token blocking-based datasets, **DBPEDIA** and **FREEB** (Freebase), published in [11]. For these two datasets, instead of using our LSH-based block building method, we use the exact token-blocking input published by the authors in [11].

Finally, we introduce a large, labeled, public-domain dataset: the Ohio Voter dataset, called **VOTER**. We built this by downloading two snapshots of Ohio registrations [1] from different points in time and treating as duplicates records with the same voter ID but different demographic information. Similar work was done previously on North Carolina voter registration data [8], but the Ohio dataset is richer in that it contains 108 columns, including birthday and voter registration.

5.2 Metrics

We present a few different metrics in order to evaluate performance. We use the established *PQ*, Pair Quality, (analogous to precision) and *PC*, Pair Completeness, (analogous to recall) metrics [7]. Every dataset described in Table 1 has labeled pair-wise training data. We report the number of positively labeled

[2] https://github.com/vefthym/ParallelMetablocking

Table 2: Comparing Hash Dynamic Blocking (HDB) to other methods based on Pair Quality (PQ), Pair Completeness (PC), and Elapsed Time (T) in minutes

	Threshold (THR)			Parallel Meta (PMB)			Hash Dynamic (HDB)		
Dataset	*PQ*	*PC*	*T*	*PQ*	*PC*	*T*	*PQ*	*PC*	*T*
VAR1	**0.3003**	**0.9450**	2.6	0.1825	0.5428	13.9	0.2612	**0.9450**	4.9
VAR10	**0.3336**	0.9444	4.8	0.1174	0.7773	16.3	0.3083	**0.9445**	10.6
VAR25	**0.3445**	0.9251	7.8	0.1213	0.7620	23.8	0.2739	**0.9315**	20.7
VAR50	**0.3355**	0.9240	13.9				0.2394	**0.9343**	30.5
VAR107	**0.3227**	0.9102	23.8				0.2168	**0.9277**	54.6
VAR530	0.4787	0.8341	110.3				**0.4834**	**0.8588**	169.2
VOTER	**6.96e-4**	**1.0000**	2.6	2.74e-4	0.9986	24.2	5.19e-4	**1.0000**	5.4
SCHOLAR	**5.52e-3**	**0.4749**	0.8	1.71e-3	0.3583	10.5	5.52e-3	**0.4749**	1.2
CITESR	**1.32e-2**	0.9544	2.2	5.02e-4	0.4808	15.5	5.58e-3	**0.9545**	3.6
DBPEDIA	**7.99e-4**	0.9376	3.6	2.60e-4	0.9742	14.5	2.38e-4	**0.9921**	22.1
FREEB	**2.37e-4**	0.7340	6.8	1.50e-4	0.8303	23.8	1.47e-4	**0.8497**	25.9

pairs as *+Labels*. However, as is a common problem in record linkage evaluation, the number of labeled training pairs is usually incomplete and significantly smaller than the possible pairs in the input record set. For these incompletely labeled datasets, we present PC with respect to the labeled pairs, defined as: $|P \cap L^+|/|L^+|$, where P is the set of pairs produced by the blocker and L^+ is the set of positively labeled pairs in the training data.

To measure PQ for datasets without complete ground truth, we follow a similar practice as described in [19] where we employ an *Oracle* pair-wise model previously trained on the complete labeled dataset. We use the same oracle per dataset for all experiments and thus the numbers are relatively comparable, despite containing some error introduced by the imperfect Oracle.

5.3 Comparing Hashed Dynamic Blocking to other methods

Table 2 shows the performance of Hashed Dynamic Blocking (HDB). Threshold Blocking is a simple strategy, but comparing the results to HDB shows that in all large datasets, recall is hurt when we simply discard over-sized blocks. Table 3 shows the number of pairs produced in each of our experiment setups. Since PMB and HDB have different hyper-parameters that affect the operating point, we configured both through trial and error to produce a similar number of pairs to relatively evaluate PC and PQ at the same operating point. *Naive* here represents the number of pairs produced by simple blocking on the blocking key values; that is comparing all pairs of records that share any blocking key value and not discarding any blocks due to size (as we do in THR). The VAR530 dataset with Naive blocking produces 120 quadrillion pairs, which on our cluster would take over $7,200$ years to score, highlighting the need for more sophisticated blocking approaches at massive scale.

We have not been able to successfully execute PMB on some of the larger sets (VAR50, VAR107, VAR530); it fails with out-of-memory errors and we have

Table 3: Comparing the number of pairs produced by different blocking algorithms

	Naive	THR	PMB	HDB
Dataset	$\|\|B\|\|$	$\|\|B\|\|$	$\|\|B\|\|$	$\|\|B\|\|$
VAR1	4.5e11	1.1e8	1.8e8	1.6e8
VAR10	4.4e13	1.1e9	1.7e9	1.4e9
VAR25	2.6e14	3.2e9	5.4e9	5.3e9
VAR50	1.1e15	6.4e9		1.3e10
VAR107	5.2e15	1.4e10		3.0e10
VAR530	1.2e17	4.5e10		6.8e10

	Naive	THR	PMB	HDB
Dataset	$\|\|B\|\|$	$\|\|B\|\|$	$\|\|B\|\|$	$\|\|B\|\|$
VOTER	3.5e11	9.3e8	2.3e9	1.3e9
SCHOLAR	2.4e7	2.0e6	4.7e6	2.0e6
CITESR	2.3e11	4.5e7	6.2e8	1.1e8
DBPEDIA	8.0e10	1.0e9	3.2e9	3.7e9
FREEB	2.2e11	4.1e9	7.0e9	7.6e9

been unable to get it to complete. We ran into similar issues when running BLAST [26] on our huge datasets, which we expected given that they broadcast hash maps of record ID $\rightarrow$ blocking keys to every node. For our large datasets, this single broadcast map would be multiple TBs of memory.

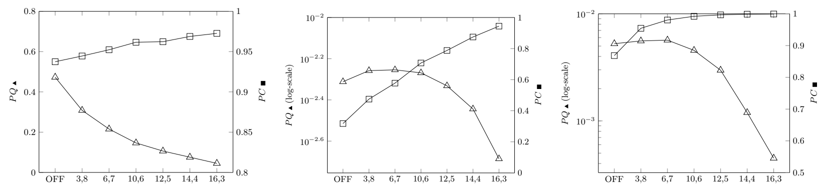
We note that HDB demonstrates improved recall over PMB despite PMB producing more pairs to evaluate. We believe this may be a consequence of the heuristic of meta-blocking weighting pairs that occur in multiple blocks. In the case of LSH-based blocking keys where there are many highly overlapping blocks, this may result in PMB picking many redundant pairs that don't improve compression. HDB by contrast, prefers to focus on the blocks that are small enough to thoroughly evaluate and find intersections of over-sized blocks. This may produce more diversity in the pairs emitted by HDB compared to PMB.

5.4 Comparing LSH Configurations

To illustrate the impact of including Locality Sensitive Hashing (LSH) with HDB we present numbers showing how PQ and PC are affected by various LSH configurations. Figure 3 shows the results with varying the number of *bands* b between 3 and 16 and varying the number of minhashes per band w from 8 to 3. As expected, LSH improves recall for datasets that have multi-token text fields, which is most of the datasets evaluated. In some instances, adding LSH dramatically improves recall. As expected, for most datasets the precision decreases as LSH becomes more liberal. Figure 1 shows a scatter plot of many different LSH configurations on the SCHOLAR dataset and includes the Token Blocking ($HDBTB = 0.5$) result for comparison to highlight the differences in PQ and PC. The diameter of each data point in the scatter plot is a linear scaling of the number of pairs produced.

6 Conclusions

We have shown Hashed Dynamic Blocking being applied to different large datasets up to 530M records. We also introduced the LSH-based block building technique, and illustrated its usefulness in blocking huge datasets. The Hashed Dynamic Blocking algorithm leverages a fortunate convergence in the requirements for



(a) VAR10 (b) SCHOLAR (c) CITESR

Fig. 3: PQ and PC of various LSH(b, w) settings on three datasets with text fields

efficiency and accuracy. HDB accomplishes this through a new algorithm which iteratively intersects and counts sets of record IDs using an inverted index and approximate counting and membership data structures. This efficient implementation is fast, robust, cross-domain, and schema-independent, thus making it an attractive option for blocking large complex databases.

References

1. Ohio voter registration and election history statewide data. `https://www6.ohiosos.gov/ords/f?p=VOTERFTP:STWD:::#stwdVtrFiles`, accessed: 2019-12-21 and 2020-02-08. These two snapshots are available for research purposes from the authors.
2. Ash, S.M., Ip-Lin, K.: Embracing the sparse, noisy, and interrelated aspects of patient demographics for use in clinical medical record linkage. In: AMIA Summits on Translational Science Proceedings. p. 425. AMIA (2015)
3. Bilenko, M., Kamath, B., Mooney, R.J.: Adaptive blocking: Learning to scale up record linkage. In: IEEE International Conference on Data Mining, ICDM. pp. 87–96 (2006)
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
5. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. Journal of Computer and System Sciences **60**(3), 630–659 (2000)
6. Chen, S., Borthwick, A., Carvalho, V.R.: The Case for Cost-Sensitive and Easy-To-Interpret Models in Industrial Record Linkage. In: 9th International Workshop on Quality in Databases (2011)
7. Christen, P.: A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. IEEE Transactions on Knowledge and Data Engineering pp. 1–20 (2011)
8. Christen, P.: Preparation of a real temporal voter data set for record linkage and duplicate detection research. `http://cs.anu.edu.au/~./Peter.Christen/publications/ncvoter-report-29june2014.pdf` (2013)
9. Chu, X., Ilyas, I.F., Koutris, P.: Distributed Data Deduplication. Proceedings of the VLDB Endowment **9**(11), 864–875 (2016)
10. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms **55**(1), 58–75 (2005)

11. Efthymiou, V., Papadakis, G., Papastefanatos, G., Stefanidis, K., Palpanas, T.: Parallel meta-blocking for scaling entity resolution over big heterogeneous data. Information Systems **65**, 137–157 (2017)
12. Elmagarmid, A., Ipeirotis, P., Verykios, V.: Duplicate Record Detection: A Survey. IEEE Transactions on Knowledge and Data Engineering **19**(1), 1–16 (2007)
13. Gionis, A., Indyk, P., Motwani, R.: Similarity Search in High Dimensions via Hashing. Proceedings of the 25th International Conference on Very Large Data Bases pp. 518–529 (1999)
14. Hassanzadeh, O., Chiang, F., Lee, H.C., Miller, R.J.: Framework for evaluating clustering algorithms in duplicate detection. Proceedings of the VLDB Endowment **2**, 1282–1293 (2009)
15. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: 30th Annual ACM Symposium on Theory of Computing. pp. 604–613. ACM (1998)
16. Köpcke, H., Thor, A., Rahm, E.: Evaluation of entity resolution approaches on real-world match problems. Proceedings of the VLDB Endowment **3**(1-2), 484–493 (2010)
17. Koudas, N., Sarawagi, S., Srivastava, D.: Record linkage: Similarity measures and algorithms. In: ACM SIGMOD International Conference on Management of Data. pp. 802–803. ACM (2006)
18. Leskovec, J., Rajaraman, A., Ullman, J.D.: Finding Similar Items. In: Mining of Masive Datasets, pp. 72–130. 2nd edn. (2014)
19. McNeill, W.P., Kardes, H., Borthwick, A.: Dynamic Record Blocking: Efficient Linking of Massive Databases in MapReduce. In: Quality in Databases (2012)
20. Mudgal, S., Li, H., Rekatsinas, T., Doan, A., Park, Y., Krishnan, G., Deep, R., Arcaute, E., Raghavendra, V.: Deep Learning for Entity Matching: A Design Space Exploration. In: 2018 International Conference on Management of Data. pp. 19–34 (2018)
21. Papadakis, G., Koutrika, G., Palpanas, T., Nejdl, W.: Meta-Blocking: Taking Entity Resolution to the Next Level. IEEE Transactions on Knowledge and Data Engineering **26**(8), 1946–1960 (2014)
22. Papadakis, G., Papastefanatos, G., Palpanas, T., Koubarakis, M.: Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking. EBDT (February) (2016)
23. Papadakis, G., Skoutas, D., Thanos, E., Palpanas, T.: A Survey of Blocking and Filtering Techniques for Entity Resolution. arXiv e-prints arXiv:1905.06167 (May 2019)
24. Papadakis, G., Svirsky, J., Gal, A., Palpanas, T.: Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. Proceedings of the VLDB Endowment **9**(9), 684–695 (2016)
25. Reas, R., Ash, S., Barton, R., Borthwick, A.: SuperPart : Supervised graph partitioning for record linkage. In: IEEE International Conference on Data Mining (2018)
26. Simonini, G., Gagliardelli, L., Bergamaschi, S., Jagadish, H.: Scaling entity resolution: A loosely schema-aware approach. Information Systems **83**, 145–165 (2019)
27. Van Dam, I., van Ginkel, G., Kuipers, W., Nijenhuis, N., Vandic, D., Frasincar, F.: Duplicate detection in web shops using LSH to reduce the number of computations. In: 31st Annual ACM Symposium on Applied Computing. pp. 772–779 (2016)
28. Wang, X., Sun, A., Kardes, H., Agrawal, S., Chen, L., Borthwick, A.: Probabilistic estimates of attribute statistics and match likelihood for people entity resolution. In: 2014 IEEE International Conference on Big Data. pp. 92–99. IEEE (2014)