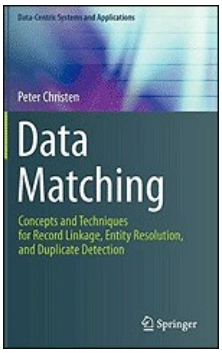


Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection

by Peter Christen

Chapter 4: Indexing

4.1 Why Indexing?

The simple example given in Chap. 2, specifically Figs. 2.4, 2.5 and 2.6 on pp. 29 and 31, helps to illustrate that even when matching small databases the majority of comparisons between records will correspond to non-matches. These are comparisons between two records that each refers to a different entity. As will be covered in Chap. 5, the detailed comparison of records can be a computationally expensive undertaking, with some comparison functions having a computation complexity that is quadratic in the lengths of the attribute values (that most commonly are strings) that are compared. The comparison step is generally the computationally most expensive step in the data matching process.

The aim of indexing in data matching is to reduce the number of record pairs that are compared in detail as much as possible, by removing pairs that unlikely correspond to true matches. At the same time, all record pairs that possibly correspond to true matches (i.e. where the two records of a pair refer to the same entity) need to be kept for detailed comparison. Without indexing, the matching of two databases that contain m and n records, respectively, would result in $m \times n$ detailed record pair comparisons. For large databases, this is clearly not feasible.

Indexing can be seen as a filtering or searching step. Because it is mostly based on some form of index data structure that brings 'similar' values together (what is similar will be discussed below), the term *indexing* is commonly used to name this step of the data matching process [64].

The general approach of indexing techniques is to process all records of the databases to be matched and to either insert each record into one or several blocks, lists or clusters, according to some criteria, or to sort the databases such that similar records are moved closely together. The criteria used is commonly called a 'blocking key' (the term used in this book) or 'sorting key'. The blocking or sorting key values are generated based on the values of either a single or from several attributes. These values are often encoded, as will be discussed below. As an example blocking key, a postcode (or zipcode) attribute could be used, such that all records that have the same postcode value will be inserted into the same block or index list. For a sorting-based indexing technique, using this sorting key the databases will be sorted according to postcode values. This sorting will lead to records that have the same value in the postcode attribute being next to each other.

A critical aspect for any indexing technique is the definition of the blocking key or keys used. As will be further discussed in the following section, a major consideration is the quality of the values in the attributes used as blocking keys, especially their completeness (how many records have values in an attribute), and the frequency distribution of the values in these attributes. Both these characteristics affect the number of candidate record pairs that are generated, and their quality (i.e. how many refer to true matches or not). This in turn will affect the overall accuracy and completeness results of a data matching exercise.

Indexing is not just important for data matching, it also needs to be applied for the deduplication of a single database. Without indexing, in a deduplication project each record in a database would be compared with all others, resulting in a total of $n \times (n - 1)/2$ record pair comparisons for a database that contains n records. Because the comparison of two records is symmetric, each pair only needs to be compared once. The indexing techniques used for the matching of two databases can also be applied for the deduplication of a single database.

The question of how to evaluate indexing techniques will be discussed in detail in Sect. 7.3. Three measures are generally used [71]. The first measure, known as reduction ratio, calculates how many candidate record pairs are generated by an indexing technique compared to all possible record pairs (full naive pair-wise comparison of all pairs). The second measure, called pairs completeness, calculates how many record pairs that refer to known true matches are included in the candidate record pairs (this measure corresponds to recall as used in information retrieval [288]). The third measure, pairs quality, calculates how many of the candidate record pairs generated correspond to true matches (this measure corresponds to precision as used in information retrieval). The last two measures require knowledge about the true match status of all record pairs, which is commonly not available in real-world matching or deduplication situations (a topic which will be covered in detail in Chap. 7).

4.2 Defining Blocking Keys

As a recent experimental survey highlighted [64], one of the most important aspects of the indexing step is not which indexing technique is employed in a data matching project, but the definition of the blocking key(s) that results in similar records being successfully grouped into the same block(s).

At this point a discussion about what constitutes 'similar records' is warranted. Depending upon the data to be matched, similarity between attribute values can refer to phonetic similarity (for example how similar two names sound), character shape

similarity (for example how similar two written names look) or numerical similarity (for example how close two age or date values are to each other). If it is known how the data in the databases to be matched or deduplicated were recorded or entered, then an appropriate encoding function can be applied on attribute values when the blocking key values (BKVs) are generated from them. The aim of such an encoding is to bring similar values together such that they are inserted into the same block.

The most common forms of data entry are manually typed values (from hand-written or typed forms), values scanned and automatically recognised using OCR technology, or values dictated and transcribed using an automatic speech recognition system [72]. A widely used approach in indexing to convert 'similar' values into the same BKV is to employ a phonetic encoding function, such as Soundex, NYSIIS or Double-Metaphone [57]. These functions replace a (name) string with a code that reflects how a name would sound if it is spoken. Names that sounds similar are converted into the same code. The phonetic encoding functions most commonly used for data matching are presented in the following section.

When blocking keys are defined based on the attributes available in the databases to be matched or deduplicated, then several issues need to be considered.

- *Attribute data quality*: the quality of the values in an attribute will influence the quality of the BKVs generated. If an attribute has a missing or empty value in a large portion of records, then as a result many records will be added into a block where the BKV is an empty value. This raises the question of whether having an empty value in an attribute means that the records that have an empty value are similar to each other or not.

  Ideally, an attribute used to generate BKVs should be as complete as possible, i.e. all records in a database should contain a value in this attribute. These values should also be of high quality, because any error in a value that results in a different BKV will mean that a corresponding record is inserted into the wrong block [71]. For example, if a postcode value is recorded wrongly as '2130' rather than '2730' (possibly an OCR mistake), then the corresponding record might be inserted into the block with BKV '2130' rather than '2730', and therefore it would not be compared with the records that it potentially matches with.

- *Attribute value frequencies*: the frequency distribution of the values in an attribute used as part of a blocking key will influence the number of candidate record pairs that are generated. If the frequency distribution is skewed such that some values are very frequent, then these most frequent values will dominate the number of candidate pairs that are generated.

  For example, if an attribute that contains surnames is used to generate BKVs, then the two blocks generated from the surname values 'smith' and 'miller' (two of the most common surnames in many English speaking countries) will be large. If for example two databases where each contains 1 million records are being matched, and only one percent (10, 000 records) contains the surname 'smith', then this single block will generate 10, 000 × 10, 000 = 100, 000, 000 candidate record pairs that are to be compared in detail. This clearly would not make sense, and the information contained in other attributes of these records (such as given name, age or postcode values) would not only reduce the number of record pair comparisons, but also increase the likelihood that the compared pairs do refer to the same person. The application of an encoding function, such as Soundex for example, can make the problem of large blocks even worse, because several attribute values are mapped into the same encoding value and therefore into the same block. This can result in more records being inserted into the same block.

  It is therefore of advantage to select attributes as blocking keys where attribute values have a frequency distribution close to the uniform distribution, resulting in blocks that are of equal sizes.

- *Trade-off between number and size of blocks*: the third issue that needs to be considered is the trade-off between the number of BKVs (and thus the number of blocks) and the size of the blocks generated (and thus the number of candidate record pairs generated) [20, 64]. On the one hand, a small number of large blocks will result in a larger number of candidate record pairs that likely contain more of the true matching record pairs. On the other hand, a large number of small blocks will lead to a smaller number of candidate record pairs (and thus a reduced run time) at the cost of potentially missing more of the true matching pairs.

  The more specific a blocking key definition is, the smaller the resulting blocks will become and therefore less record pair comparisons need to be conducted. A more specific blocking key definition can be achieved by concatenating values from several attributes, possibly encoded first, as illustrated in Fig. 4.1.

  As will be discussed later in this chapter, some of the presented indexing techniques are more sensitive to the choice of blocking keys than others. Having a more specific blocking key that leads to a larger number of smaller blocks is also of advantage for indexing techniques that sort the databases according to the blocking key (or sorting key), because a larger number of BKVs allows a more fine-grained sorting of the records in a database. This will be described in detail in the relevant sections later in this chapter.

RecID	GivenName	Surname	Postcode	Suburb
r1	peter	christen	2010	north sydney
r2	paul	smith	2600	canberra
r3	pedro	kristen	2000	sydeny
r4	pablo	smyth	2700	canberra sth

RecID	PC+Sndx(GiN)	Fi2D(PC)+DMe(SurN)	La2D(PC)+Sndx(SubN)
r1	2010-p360	**20-krst**	10-n632
r2	2600-p400	26-sm0	**00-c516**
r3	2000-p360	**20-krst**	00-s530
r4	2700-p140	27-sm0	**00-c516**

Figure 4.1: Example records in the upper table and their blocking key values (BKVs) in the lower table, adapted from [64]. The first blocking key definition concatenates postcode (PC) values with Soundex (Sndx) encoded given name (GiN) values, the second blocking key definition concatenates the first two digits (Fi2D) of postcode values with Double-Metaphone (DMe) encoded surname (SurN) values and the third concatenates the last two digits (La2D) of postcodes with Soundex encoded suburb name (SubN) values. The hyphens ('-') in the BKVs are only shown for illustration, in realworld applications they would not be inserted. The two bold highlighted pairs show that records r1 and r3 would be inserted into the block with key '20-krst', and records r2 and r4 into the block with key '00-c516'

As was discussed in the previous chapter, real-world data are commonly dirty [140], and therefore the approach used to generate the BKVs must be able to deal with data that contain errors and variations and still achieve the aims of the indexing step, namely to put similar records into the same block, or closely together in the sorted databases.

As the example given under the attribute quality issue described above illustrated, an error in an attribute value used as a blocking key will lead to a record potentially being inserted into a different block. A commonly used way to overcome this problem is to define several different blocking keys, ideally based on different attributes, rather than having one blocking key only. The union of all candidate record pairs generated by each of the blocking key definitions is used in the comparison step to perform the detailed comparisons between records. Figure 4.1 illustrates a set of four example records and three blocking key definitions applied on them.

An alternative approach is to run the indexing step several times using different blocking key definitions (sometimes called 'blocking passes' [287]), and to compare and classify the generated candidate record pairs, with pairs classified as matches being removed from the input databases (or their records flagged as being matched). This approach also allows that different comparison and classification functions can be used in the different blocking passes.

The objective of using several blocking key definitions is that in at least one of them no errors or variations occur in the BKV, and thus a record is inserted into the 'correct' block and compared with those records that likely match with it. A second advantage of this approach is also that more selective blocking key definitions can be used, as was discussed above in the description of the trade-off between block numbers and their sizes. These will result in smaller blocks that are more specific and group more similar records together. And because several blocking key definitions are used, the likelihood increases that a pair of records that refers to a match has at least one BKV in common, as illustrated in Fig. 4.1.

All the indexing techniques that will be discussed in the remainder of this chapter do require the definition of a blocking or sorting key. An optimal definition of a blocking key would result in (1) all true matches being included in the candidate record pairs generated while (2) the total number of candidate record pairs generated is kept as small as possible. Blocking keys should generally be defined by keeping in mind the indexing technique that will be employed.

While traditionally blocking keys were defined manually by somebody who ideally has expertise in both data matching techniques and the domain of the data that are to be matched or deduplicated (especially the quality and characteristics of the data), several techniques have recently been proposed that allow the learning of optimal blocking keys from training data [34, 188]. These techniques are based on supervised machine learning algorithms and require training data in the form of pairs of records that are known to refer to true matches or true non-matches. Having such training data that need to be of high quality and diverse enough to cover as many true matches as possible, are however hard to get in many practical data matching applications. Therefore, the manual definition of blocking keys is still a widespread undertaking. These learning-based techniques will be described in more detail in Sect. 4.12.

4.3 (Phonetic) Encoding Functions

Functions to (phonetically) encode attribute values before they are used as blocking or sorting key values are commonly used

in the indexing step of data matching and deduplication to bring similar sounding string values, that are often assumed to refer to names, into the same blocks. The records that contain these similar sounding names will then be compared in detail in the comparison step.

Phonetic encoding functions can however also be used in the comparison step to calculate the similarity between similar sounding string values, as will be discussed further in Sect. 5.2.

The common idea behind all phonetic encoding functions is that they attempt to convert a string, commonly assumed to refer to a name, into a code according to how a name is pronounced, i.e. how a name would be spoken [57]. This encoding process is often language dependent. Most techniques that have been developed (including all techniques presented in this chapter), are based on the assumption that names originate from the English language. Some of these techniques have been adapted for other languages [175, 238]. Other techniques, such as Double-Metaphone discussed below, can generate two encodings of a single name, depending upon whether there are variations in the spelling of a name.

4.3.1 Soundex

The Soundex [145, 175, 302] algorithm is one of the oldest approaches. It was developed and patented by Russell and Odell in 1918 [201], and is one of the best known and most widely used phonetic encoding algorithm. Based on American-English language pronunciation, it encodes name strings by keeping the first letter in a string and converting the remaining characters of the string into numbers according to the transformation table given in Fig. 4.2.

a, e, h, i, o, u, w, y	→ 0
b, f, p, v	→ 1
c, g, j, k, q, s, x, z	→ 2
d, t	→ 3
l	→ 4
m, n	→ 5
r	→ 6

Figure 4.2: Soundex encoding transformation table

After transforming a string into digits, all zeros (which correspond to vowels and 'h', 'w' and 'y') are removed from the encoded string, and all repetitions of the same number are also removed. For example, an initial transformed encoding of 'p0330111' is converted into 'p31', and the initial encoding 's550144042' is converted into 's5142'. If an encoding contains less than three digits (as the first example), then the code is extended with zeros to a total length of three digits (so 'p31' becomes 'p310'), while codes that contain more than three digits are truncated to three digits only (thus the encoding 's5142' becomes 's514').

The advantages of Soundex are its simplicity and computational efficiency. A first major drawback of Soundex is that a difference in the first letter of two name strings results in different Soundex codes for these names, as can be seen in Table 4.1 for the name strings 'christina' and 'kristina' with their corresponding Soundex codes 'c623' and 'k623', respectively. A second drawback is that Soundex codes are mostly representing the beginning of name strings, and differences that appear towards the end of two names are often not represented properly because they are pruned away if the codes are too long. A commonly applied solution to both these drawbacks is to not only generate the Soundex encodings of name strings, but also the encodings of the reversed name strings. A name pair is then seen to be similar if either of the two calculated encodings are the same.

Table 4.1: Example name strings and their phonetic encodings. Variations of the same name are grouped together

String	Soundex	Phonex	Phonix	NYSIIS	Double Metaphone	Fuzzy Soundex
peter	p360	b360	p300	pata	ptr	p360
pete	p300	b300	p300	pat	pt	p300
pedro	p360	b360	p360	padr	ptr	p360
Stephen	s315	s315	s375	staf	stfn	s315
steve	s310	s310	s370	staf	stf	s310
smith	s530	s530	s530	snat	sm0, xmt	s530
smythe	s530	s530	s530	snat	sm0, xmt	s530
gail	g400	g400	g400	gal	kl	g400
gayle	g400	g400	g400	gal	kl	g400

Table 4.1: Example name strings and their phonetic encodings. Variations of the same name are grouped together

String	Soundex	Phonex	Phonix	NYSIIS	Double Metaphone	Fuzzy Soundex
christine	c623	c623	k683	chra	krst	k693
christina	c623	c623	k683	chra	krst	k693
kristina	k623	c623	k683	cras	krst	k693

4.3.2 Phonex

This encoding algorithm is a variation of the original Soundex approach [175]. It aims to improve the quality of the calculated encodings through a pre-processing step where name strings are modified according to their English pronunciation before Soundex-like encodings are generated. The following modifications are applied to a name string in the pre-processing step:

- All 's' characters at the end are removed.
- A 'kn' character sequence at the beginning is replaced with a single 'n' character.
- A 'ph' character sequence at the beginning is replaced with a single 'f' character.
- A 'wr' character sequence at the beginning is replaced with a single 'r' character.
- An 'h' character at the beginning of a name string is removed.
- If the first character is a vowel (including 'y') then it is replaced with an 'a'.
- If the first character is a 'p' then it is replaced by a 'b'.
- If the first character is a 'v' then it is replaced by an 'f'.
- If the first character is a 'k' or a 'q' then it is replaced by a 'c'.
- If the first character is a 'j' then it is replaced by a 'g'.
- If the first character is a 'z' then it is replaced by an 's'.

After this initial pre-processing step, the processed name string is encoded similar as with Soundex into a code made of the initial character followed by three digits. The transformation from letters into digits is somewhat different from the original Soundex transformation, because several transformation rules are taken into account. Similar to the initial pre-processing done, these rules take character sequences into account when converting letters into digits [175].

4.3.3 Phonix

The Phonix algorithm extends the idea of Phonex pre-processing of name strings even further by applying more than a hundred transformation rules. These rules are applied not only on a single character, but also on sequences of several characters. While most rules are applied anywhere in a name string, 19 rules are only applied if the character(s) appear(s) at the beginning of a string, 12 rules are applied only to the middle of a string, and 28 rules only to the end of a string.

Similar to Soundex and Phonex, the transformed name string is encoded into a code consisting of a starting letter followed by three digits (again removing zeros and duplicate numbers). The transformation table is different from the one used in Soundex, as can be seen from Fig. 4.3.

---

a, e, h, i, o, u, w, y → 0
b, p → 1
c, g, j, k, q → 2
d, t → 3
l → 4
m, n → 5
r → 6
f, v → 7
s, x, z → 8

Figure 4.3: Phonix encoding transformation table

The larger number of transformation rules means that the Phonix algorithm is more complex and thus slower than the Soundex and Phonex algorithms. An experimental evaluation has shown that Phonix is around ten times slower than Soundex on different data sets that contained several thousand name strings each [57].

4.3.4 NYSIIS

The *New York State Identification and Intelligence System* (NYSIIS) phonetic encoding algorithm departs from the one-letter three-digit code and only returns an encoding made of letters [40]. Similar to Phonex and Phonix, it applies various rules to the input name string. These rules are:

- Transform various beginnings of the name string: 'mac' becomes 'mcc', 'kn' becomes 'n', 'k' is replaced with 'c', 'ph' and 'pf' are replaced with 'ff', and 'sch' with 'sss'.
- Transform various endings of the name string: 'ee' and 'ie' are replaced with 'y', while 'dt', 'rt', 'rd', 'nt' and 'nd' are all replaced with 'd' only.
- The first letter of the transformed name string now becomes the first letter of the NYSIIS encoding.
- The remaining letters of the transformed name string are further transformed using one of the following rules, applied starting from the beginning of the string:
    1. 'ev' is replaced by 'af'.
    2. 'e', 'i', 'o' and 'u' are replaced with 'a'.
    3. 'q' is replaced by 'g', 'z' is replaced by 's' and 'm' is replaced by 'n'.
    4. 'kn' is replaced by 'n' and 'k' by 'c'.
    5. 'sch' is replaced by 'sss' and 'ph' by 'ff'.
    6. If the letter before or after an 'h' is not a vowel then the 'h' is replaced by the letter before it.
    7. If the letter before a 'w' is a vowel then the 'w' is replaced with 'a'.
    8. Only add the current processed letter to the NYSIIS encoding if it is different from the previous letter in the encoding.
- Several rules are then applied to the end of the encoding:
    1. If the last letter in the encoding is an 'a' or 's' then remove it.
    2. If the encoding ends with 'ay' replace it by 'y'.
- Finally, if the length of the encoding is longer than six letters then truncate it to the first six letters only.

Besides Soundex, the NYSIIS algorithm is the second most popular phonetic encoding algorithm employed for data matching and deduplication, as well as other applications that require the grouping of similar sounding names strings.

4.3.5 Oxford Name Compression Algorithm

The Oxford Name Compression Algorithm (ONCA) combines the NYSIIS and Soundex algorithms [118, 119]. The ONCA has been used in the Oxford Record Linkage System. In a first step, name strings are processed using a version of the NYSIIS algorithm that was adapted for Anglo-Saxon and European names. The resulting phonetic codes are further processed by applying the standard Soundex algorithm on them. Table 4.2 shows several examples of the ONCA approach.

Table 4.2: Examples of the ONCA phonetic encoding algorithm, adapted from [119]

Original string	NYSIIS encoding	ONCA encoding
andersen, anderson	andar	a536
brian, brown, brun	bran	b650
capp, cope, copp, kipp	cap	c100

Table 4.2: Examples of the ONCA phonetic encoding algorithm, adapted from [119]

Original string	NYSIIS encoding	ONCA encoding
dane, dean, dent, dionne	dan	d500
smith, schmit, schmidt	snat	s530
truman, trueman	tranan	t655

4.3.6 Double-Metaphone

A major drawback of the four phonetic encoding algorithms discussed so far is that they are specifically aimed at English names, and are therefore not suitable for databases that contain names from different languages. Many countries have an increasingly multi-cultural population, and therefore non-English names appear more frequently in many databases that contain detailed information about people. It is therefore important that a phonetic encoding algorithm can accommodate non-English names.

The Double-Metaphone algorithm attempts to accomplish this by better accounting for European and Asian names [211]. Similar to the Phonix and NYSIIS algorithms, a large number of transformation rules are applied to a name string. These rules take the position within a name string into account, and some rules also consider the previous and following letters. In line with the NYSIIS encoding, Double-Metaphone returns an encoding made of letters only. Different from the NYSIIS algorithm, however, is that for certain name strings not only one phonetic encoding is calculated but two. These two codes are based on the application of different phonetic transformation rules. For example, the Polish name 'kuczewski' will be encoded as 'kssk' and 'kxfsk', accounting for different spelling variations of this name. In general, Double-Metaphone seems to be generating encodings that are closer to the correct pronunciation of names than NYSIIS.

4.3.7 Fuzzy Soundex

This algorithm combines a q-gram based pre-processing step with a Soundex like transformation table [145]. Q-grams are substrings of length q. In the fuzzy Soundex algorithm, q-grams of length 2 (bigrams) and 3 (trigrams) are applied similar to the substitutions applied in Phonix, NYSIIS or Double-Metaphone. Some of these substitutions are only applied at the beginning of a name string, while others are applied anywhere. The pre-processed name string is then converted into a one-letter three-digit encoding using the transformation table shown in Fig. 4.4.

Figure 4.4: Fuzzy Soundex encoding transformation table

Fuzzy Soundex was developed within the information retrieval community with the aim to improve the quality of Soundex-based retrieval [145]. Combined with a q-gram based pattern matching algorithm, it achieved better retrieval results on a database of over 30,000 names than the basic Soundex algorithm [210].

4.3.8 Other Encoding Functions

The encoding functions discussed so far are all aimed at the phonetic encoding of strings that are assumed to be names, such as personal or address names. As different types of data are commonly being used in data matching, some forms of encoding functions (possibly not phonetic) need to be available for data that do not correspond to names.

Recall that the objective of an encoding function is to bring 'similar' values together. For data that are not name strings, a 'binning' type of encoding function can be employed. Commonly used to smooth noisy data [135], binning puts numerical values that are similar to each other into the same bin, an approach related to blocking.

For example, numerical age values (as years) can be binned by having one bin (block) per age decade, thus inserting all records that have an age value from 0–9 into one bin, those with an age value of 10–19 into a second bin and so on. For

postcode values, as illustrated in Fig. 4.1, blocking can be achieved by only taking a subset of the available digits, such as only the first two or only the last two out of four postcode digits. This leads to a maximum of 100 bins. During the indexing process, all records that have the same first two (or last two) digits in common are inserted into the same block. If these blocks become too big, then taking the first or last three digits (leading to maximum 1000 bins and blocks) is an alternative.

For date values, depending upon the distribution and spread of date values in a database (i.e. the difference between the first and last date), either year values only, or month and year values combined can be used as the encoding function, resulting for example in blocking key values such as 'jan2011', 'feb2011' and so on.

4.4 Standard Blocking

This traditional indexing approach has been used in data matching and deduplication for several decades [108]. The uniqueness of this approach is that the identifier of each record is inserted into one block only. All other indexing techniques presented in this chapter potentially insert a single record into several blocks.

Assuming a single blocking key has been defined, one blocking key value (BKV) will be generated for each record in the input database(s). This BKV determines into which block a record is inserted. All records that have the same BKV are inserted into the same block. For the matching of two databases, pairs of candidate records are generated from all records that have the same BKV across both databases. If a BKV occurs in records from one database only, then no record pairs will be formed from this block, as there are no records in the corresponding block in the other database.

For a deduplication, pairs of candidate records are generated from all unique pairs of record identifiers within a block. Because the comparison of two records is symmetric, each unique record pair only needs to be compared once. For example, with a block that contains the three record identifiers 'r1', 'r2' and 'r3', the generated record pairs for a deduplication would be (r1,r2), (r1,r3) and (r2,r3), but not (r2,r1), (r3,r1) or (r3,r2).

An efficient way to implement standard blocking is to build an inverted index data structure [288, 303], where each BKV becomes the key of an index list, and the identifiers of all records that are in the same block are inserted into the same inverted index list. Figure 4.5 illustrates such inverted index lists for two small example databases.

Database A

RecID	GivenName	Surname	Sndx(GiN)	Sndx(SurN)
a1	peter	myler	p360	**m460**
a2	pedro	smith	p360	**s530**
a3	steve	peters	**s315**	p362
a4	gail	smythe	g400	**s530**
a5	christine	miller	c623	**m460**

Database B

RecID	GivenName	Surname	Sndx(GiN)	Sndx(SurN)
b1	kristina	miller	k623	**m460**
b2	stephen	peter	**s315**	p360
b3	kylie	smith	k400	**s530**
b4	pete	myler	p300	**m460**
b5	kellie	roberts	k400	r163

Candidate record pairs from GivenName

BKV	Candidate record pairs
s315	(a3,b3)

Candidate record pairs from Surname

BKV	Candidate record pairs
m460	(a1,b1), (a1,b4), (a5,b1), (a5,b4)
s530	(a2,b3), (a4,b3)

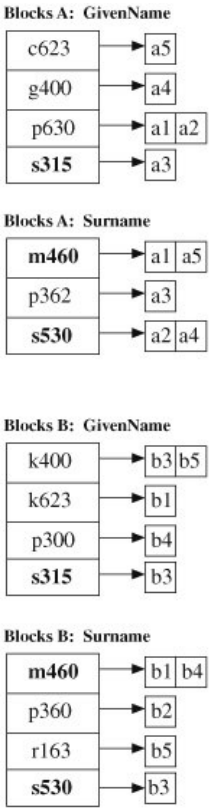


Figure 4.5: Two example databases with their blocking key values (BKVs) based on the Soundex (Sndx) encoded given name (GiN) and surname (SurN) values, the blocks generated using the standard blocking approach, and the resulting candidate record pairs. The BKVs that are generated from both databases are highlighted in boldface

As discussed in Sect. 4.2 before, several blocking keys are generally defined (often on different attributes), and for each a separate index data structure is built. Candidate record pairs are then generated independently when blocks are processed from each index data structure. However, even if a candidate record pair is generated several times, the corresponding record pair will only be compared once in the comparison step.

The number of candidate record pairs that are generated with standard blocking depends upon the frequency distribution of the BKVs [64]. The most frequent BKVs will generate the most candidate record pairs. For a simplified estimate, a uniform frequency distribution of BKVs can be assumed. If the number of records in the two databases to be matched is denoted with m and n, respectively, and the number of BKVs in common with b, then each block contains m/b or n/b records, respectively. The total number of candidate record pairs generated, c, then is

$$c = b\left(\frac{m}{b} \cdot \frac{n}{b}\right) = \frac{m \cdot n}{b}. \tag{4.1}$$

For the deduplication of a single database that contains n records (and b BKVs), the number of candidate record pairs generated, c, is

$$c = b\left(\frac{n}{b} \cdot \frac{n-1}{b}\right)/2 = \frac{n \cdot (n-1)}{2b}. \tag{4.2}$$

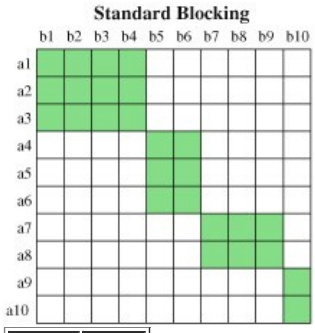
A more detailed complexity analysis for other frequency distributions of the BKVs can be found in the recent survey by Christen [64].

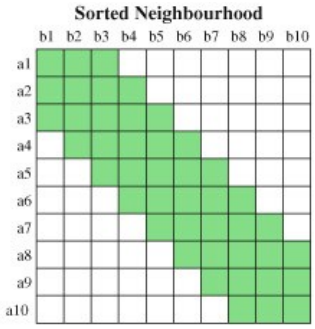
4.5 Sorted Neighbourhood Approach

The first alternative indexing technique to standard blocking was developed in the mid-1990s by Hernandez and Stolfo [140, 141]. Rather than generating blocks according to the BKVs, this approach sorts the databases to be matched according to a 'sorting key' (which is generated in a similar way as a blocking key). A sliding window of fixed size w (with $w > 1$) is then moved over the sorted databases, and candidate record pairs are generated from the records that are in the window in any given step.

Figure 4.6 illustrates both the standard blocking and the sorted neighbourhood approach. The first position of the sliding

window, with $w = 3$, contains the records 'a1' to 'a3' and 'b1' to 'b3', and therefore the nine candidate record pairs (a1,b1), (a1,b2), (a1,b3), (a2,b1), (a2,b2), (a2,b3), (a3,b1), (a3,b2) and (a3,b3) will be generated. In window position 2 (which contains records 'a2' to 'a4' and 'b2' to 'b4'), the four candidate record pairs (a2,b2), (a2,b3), (a3,b2) and (a3,b3) will again be generated. However, each unique candidate record pair will only be compared once in the comparison step. The new candidate record pairs generated in the second window position will be (a2,b4), (a3,b4), (a4,b2), (a4,b3) and (a4,b4).





RecID	BKV
a1	b400
a2	b400
a3	b400
a4	m460
a5	m460
a6	m460
a7	p362
a8	p362
a9	s530
a10	s530

RecID	BKV
b1	b400
b2	b400
b3	b400
b4	b400
b5	m460
b6	m460
b7	p362
b8	p362
b9	p362
b10	s530

RecID	SKV
a1	bail-mary
a2	bayl-pete
a3	bell-robe
a4	mell-jame
a5	mill-anne
a6	moli-fran
a7	patr-henr
a8	patt-jule
a9	smit-jonn
a10	smit-marg

RecID	SKV
b1	bayl-pete
b2	bell-gary
b3	bell-robe
b4	boyl-jame

RecID	SKV
b5	mill-anna
b6	mill-thom
b7	patt-juli
b8	patt-sall
b9	pete-brig
b10	smit-john

Figure 4.6: Comparison of the candidate record pairs that are generated (shaded squares) with standard blocking and the sorted neighbourhood approach. For standard blocking it is assumed the records are sorted according to the BKVs to better illustrate the blocks generated. The window size for the sorted neighbourhood approach is set to $w = 3$. Note that a different blocking key and sorting key definition is used for the two approaches. For standard blocking, Soundex codes on surnames are used, while for the sorted neighbourhood approach the sorting key values (SKVs) are the first four letters of surnames concatenated with the first four letters of given names. The hyphen ('-') is only used for illustration, in a practical implementation values would be concatenated directly

In the approach originally proposed by Hernandez and Stolfo [140], both databases are merged before they are sorted according to their sorting key values (SKVs), and then the sliding window is moved over the combined SKVs. In this way, similar records from both databases are moved closer together in the hope that they are included in the same window.

The criteria used to define a good sorting key is different from the criteria used to define a good blocking key. While a blocking key has to balance the quality of the candidate record pairs generated (how many true matches are included) with the size of the blocks generated (and thus the number of resulting candidate record pairs), the definition of a sorting key needs to be more specific and fine-grained, such that sorting the databases brings similar records closer together.

An important consideration when defining a sorting key is that the sorting of the databases is very sensitive to the beginning of the SKVs, especially the first character. For example, the two similar given name values 'christine' and 'kristina' will not be close to each other if the sorting is based on given names. Similar to the multiple blocking key definitions commonly used for standard blocking, it is of advantage to run several iterations of the sorted neighbourhood approach using different sorting key definitions [140].

The number of candidate record pairs that will be generated with the sorted neighbourhood approach does not depend upon the frequency distribution of the SKVs. Assuming both databases to be matched contain n records, and the window size is set to w, then there will be $(n - w + 1)$ window positions. In the first position, the number of candidate record pairs that are generated is $w \cdot w = w^2$, while in each of the following positions $w + (w - 1) = 2w - 1$ new unique candidate pairs are added to the set of candidate pairs (all other pairs in a certain window position have already been generated in previous window positions). Overall, the total number of unique candidate records pairs will be

$$c = w^2 + (n - w)(2w - 1) = 2nw - 2w^2 - n. \quad (4.3)$$

For large databases, it holds that $w \ll n$, and therefore this process has a computation complexity of $O(n)$, i.e. it is linear in the size of the databases to be matched. The same holds for the deduplication of a single database. However, the sorting of the SKVs must also be considered. This can be accomplished in $O(n \log n)$ steps.

An alternative of how the sorted neighbourhood approach can be implemented was recently proposed [64]. Depending upon how the SKVs are defined, there might be many SKVs that are the same, and therefore the sliding window will not cover all records with the same SKV. For example, in a large database there might be many records with surname 'smith' and given name 'john', resulting in many records with SKV 'smith-john' (similar to the example shown in Fig. 4.6). The proposed alternative is to generate an inverted index data structure (as can be done with standard blocking), where the index keys are the unique SKV values. The index keys are then sorted, and a sliding window is moved over the index key values rather than the SKVs directly. Each unique SKV only appears once in the sorted index key values, while with the original sorted neighbourhood approach an SKV occurred as many times as a record in the database(s) contained the SKV. At any window position, the union of the record identifiers from all inverted index lists in the current window will be used to generate the candidate record pairs for that window position. Because the inverted index lists can have different lengths, the number of candidate record pairs that are generated depends upon the distribution of the attribute values used in the SKVs rather than the window size only. Experimental results have shown that this alternative approach can lead to more true matches being included into the set of candidate record pairs at the cost of a larger number of generated candidate pairs [64].

A major drawback of the original sorted neighbourhood approach is that the fixed window size can result in missed true matches if similar records are not close enough in the sorted SKV array to be in the same window. This drawback has been

addressed by a recent approach that dynamically changes the window size *w* according to the values of the SKVs [292]. The window size is increased as long as SKVs in the sorted array are similar to each other according to an approximate string comparison function (as will be discussed in Chap. 5). A window will cover a sequence of SKVs (and their records) that have a similarity between each other above a certain similarity threshold. A new window will be started at a 'boundary pair' where two consecutive SKVs have an approximate string similarity below a certain similarity threshold.

Other recent work has generalised the standard blocking and sorted neighbourhood approach and shown that they can be two ends of the same approach [94]. Standard blocking can be seen as the sorted neighbourhood approach where the window moves *w* positions forward rather than only 1, leading to non-overlapping blocks. A novel indexing technique based on this approach has been proposed that allows the specification of a desired overlap as well as window size [94]. An experimental evaluation of this technique showed that the sorted neighbourhood approach outperformed standard blocking, especially when blocks were set to a small size [94].

4.6 Q-Gram Based Indexing

For data that are dirty and contain large amounts of errors and variations, both standard blocking and the sorted neighbourhood approach might not be able to insert records into the same blocks, for example if the beginning of a sorting key value is different for two name variations. Q-gram based indexing aims to overcome this drawback by generating variations of each BKV, and to use these variations as the actual index keys for a standard blocking-based indexing approach. Each record is inserted into several blocks according to the variations generated from its BKV [20].

Q-gram based indexing takes each blocking (or sorting) key value and converts it into a list of q-grams. A q-gram (also known as n-gram [172]) is a substring of length *q* characters. Common choices for *q* are *q* = 2 (called *bigrams* or *digrams* [162]) or *q* = 3 (called *trigrams* [258]). A string *s* that is *c* = |*s*| characters long contains *k* = *c* - *q* + 1 q-grams. The list of q-grams of a string *s* is generated using a sliding window approach that extracts *q* characters from *s* at any position from 1 to *k* of the string. For example, the bigram list that is generated from the string 'christen' is ['ch', 'hr', 'ri', 'is', 'st', 'te', 'en'].

To create variations of a BKV, sub-lists of the q-gram list are generated in a recursive approach, as illustrated in Fig. 4.7. If the original q-gram list contains *k* q-grams, then in the first step *k* sub-lists of length *k* - 1 q-grams are generated. In each of these sub-lists, one q-gram is removed. The process is then applied to each of these sub-lists in a recursive manner. A minimum threshold *t* (*t* < 1) is set by a user to decide the minimum relative length, *l*, of the shortest q-gram sub-lists that are to be generated. For a BKV that contains *k* q-grams and with a threshold set to *t*, all sub-lists down to a length

(4.4) $l = max(1, \lfloor k \cdot t \rfloor)$

RecID	BKVs (Surname)	Bigram sub-lists	Index key values
r1	miller	[mi,il,ll,le,er], [il,ll,le,er], [mi,ll,le,er], [mi,il,le,er], [mi,il,ll,er], [mi,il,ll,le], [ll,le,er], [il,le,er], [il,ll,er], [il,ll,le], [mi,le,er], [mi,ll,er], [mi,ll,le], [mi,il,er], [mi,il,le], [mi,il,ll]	'miillllleer', 'illlleer', 'millleer', 'miilleer', 'miilller', 'miillle', '**llleer**', 'illeer', 'illler', 'miller', 'illlle', 'mileer', 'millle', 'miiler', 'miille', 'miilll'
r2	muller	[mu,ul,ll,le,er], [ul,ll,le,er], [mu,ll,le,er], [mu,ul,le,er], [mu,ul,ll,er], [mu,ul,ll,le], [ll,le,er], [ul,le,er], [ul,ll,er], [ul,ll,le], [mu,le,er], [mu,ll,er], [mu,ll,le], [mu,ul,er], [mu,ul,le], [mu,ul,ll]	'muullllleer', 'ullllleer', 'mullleer', 'muulleer', 'muulller', 'muulllle', '**llleer**', 'ulleer', 'ulller', 'muller', 'ullle', 'muleer', 'mullle', 'muuler', 'muulle', 'muulll'

Figure 4.7: Q-gram based indexing with two surname values used as BKVs. Q-grams of length *q* = 2 (bigrams) are used, and the minimum threshold is set to *t* = 0.75. Duplicate q-gram sub-lists are removed. The index key value that is generated for both records is highlighted in boldface

are generated, with … denoting the rounding to the next lower integer value. All sub-lists are then converted back into strings and used as the actual index keys in an inverted index data structure, as described for standard blocking and illustrated in Fig. 4.5. Each record is inserted into several inverted index lists, according to how many index keys have been generated from its BKV [64].

As shown in Fig. 4.7, a major drawback of this indexing approach is that (even with short BKVs) a large number of sub-lists (and thus index keys) are generated, and each record is likely inserted into many blocks. The recursive generation of sub-lists is a computationally expensive procedure, especially for long BKVs and low threshold values.

The advantage of q-gram based indexing is that it can overcome errors and variations in the BKVs, and therefore records that refer to true matches are more likely inserted into the same index list, even if their BKVs are different from each other. This leads to more true matching records being compared and thus an improved matching quality.

A recent theoretical and empirical evaluation of q-gram based indexing has illustrated that the drawback of having a high computation complexity outweighs the advantage of being able to match data that are dirty [64]. Q-gram based indexing is not suitable for the matching or deduplication of large databases because generating the many q-gram sub-lists takes a prohibitive amount of time.

An approximate string join technique within a database framework that is related to q-gram based indexing was proposed by Gravano et al. [127]. It uses q-grams to reduce the computation complexity of the naive pair-wise approach of comparing all possible record pairs when joining two database tables. This technique augments a database with a table that for each record contains tuples that are made of the identifier of the record, the q-grams extracted from the record's attribute value and the positions of these q-grams within the attribute value (positional q-grams will be discussed further in Sect. 5.4). Using this q-gram table, an SQL query is used to only compare candidate record pairs that fulfil three filtering criteria. These criteria limit the number of pairs that have to be compared using an expensive user-defined function (UDF) such as edit distance (which will be discussed in Sect. 5.3). The first criteria, count filtering, selects pairs that have a certain minimum number of q-grams in common. The second criteria, position filtering, removes pairs where the common q-grams are at positions too far apart. Finally, the third criteria, length filtering, removes pairs that have a length difference of their corresponding strings above a certain threshold. An experimental evaluation by the authors using a commercial database showed that this proposed approach results in a very effective approximate string join implemented completely within a relational database [127].

4.7 Suffix-Array Based Indexing

This indexing technique is related to q-gram based indexing. It also aims to overcome errors and variations in the BKVs by generating suffix substrings (called suffixes) of the BKVs. The suffixes of a string are all its substrings with one or more characters at the beginning removed. For example, the suffixes of the string 'peter' are 'eter', 'ter', 'er' and 'r'. Each unique suffix string becomes the key of an index block, and all records that contain this suffix string are inserted into this block [7]. Similar to q-gram based indexing, the identifier of a record will likely be inserted into several inverted index lists. Figure 4.8 illustrates this approach on four example BKVs.

RecID	BKVs (GivenName)	Suffixes
r1	katherina	katherina, atherina, therina, herina, erina, rina
r2	catrina	catrina, atrina, trina, rina
r3	catherine	catherine, atherine, therine, herine, erine, rine
r4	catherina	catherina, atherina, therina, herina, erina, rina
r5	katrina	katrina, atrina, trina, rina

Suffix	RecID
atherina	r1, r4
atherine	r3
atrina	r2, r5
catherina	r4
catherine	r3
catrina	r2
erina	r1, r4
erine	r3
herina	r1, r4
herine	r3
katherina	r1

Suffix	RecID
katrina	r5
~~rina~~	~~r1, r2, r4, r5~~
rine	r3
therina	r1, r4
therine	r3
trina	r2, r5

Figure 4.8: Suffix-array based indexing example of five given name values, adapted from [64]. The minimum suffix length is set to l_{min} = 4 and the maximum block size to b_{max} = 3. As a result, the block for suffix 'rina' is too large (because four records contain this suffix value) and it is deleted before the candidate record pairs are generated

When applied for indexing, the shorter a suffix string is the more BKVs (and thus more records) will contain this suffix. This will result in very large blocks. For example, the identifiers of all records that contain a BKV that ends with the suffix 'r' will be inserted into the block with index key 'r'. To prevent such large blocks to occur, suffix-array based indexing has two parameters that influence the size of the index blocks that are generated.

- The first parameter is the minimum suffix length, l_{min}. This parameter sets the minimum length of suffix strings that are generated. With l_{min} = 4, for example, for the string 'christen' the following suffixes will be generated: 'hristen', 'risten', 'isten' and 'sten'. The identifier of each record that has the BKV 'christen' will therefore be inserted into 5 blocks (the value 'christen' will also be used as a key of the inverted index). For BKVs that are shorter than l_{min} characters (such as 'tan'), only their actual value will be used as index key.

  A BKV that is c characters long will result in $k = (c - l_{min} + 1)$ suffix strings, and therefore a record that has a BKV of length c will be inserted into k inverted index lists and thus blocks. The longer a suffix string is, the less likely it will occur frequently in all the BKVs in a database. For example, there will be more records in a database that contain the suffix value 'tina' in their BKVs compared to 'ttina' (from given name 'bettina'), because 'christina', 'kristina', 'martina' and 'santina' also contain the suffix 'tina'.

- To limit the size of the generated blocks, the second parameter used in suffix-array based indexing is the maximum block size that is allowed, b_{max}. Once the BKVs and their suffix strings have been generated for all records in a database and the record identifiers have been inserted into the suffix array index, then all index blocks that contain more than b_{max} record identifiers will be deleted. These large blocks were likely generated by short suffix strings that appear in many BKVs. By removing these large blocks, the number of candidate record pairs that are generated is limited. The idea of this pruning step is that each record identifier is likely being inserted into several index blocks. Even after the large blocks have been removed, a record identifier is still kept in one or more of the smaller blocks. In case of where a large block is deleted, there is, however, a chance that such a block contained record identifiers where their BKV was only inserted into this large block (because the BKV for these records was of length l_{min}). As a result, these records would not be part of any candidate record pair, and thus would not be compared with other records. Therefore, in such a case a large block should not be deleted completely, but its size (number of record identifiers in it) be reduced by only removing record identifiers that have the longest original BKV (i.e. that have also been inserted in several other blocks).

Suffix-array based indexing has successfully been applied to the deduplication of both English and Japanese bibliographic databases, where suffix arrays were created based on English names and Japanese characters, respectively [7].

The number of candidate record pairs, c, that will be generated with this indexing technique can be estimated assuming that all blocks that are generated contain the maximum allowed number of record identifiers, b_{max}. If b such blocks are generated, then in total

(4.5) $$c = b \cdot b_{max}^2$$

candidate record pairs are generated for the matching of two databases, and

(4.6) $$c = b \cdot (b_{max}(b_{max} - 1))/2$$

for the deduplication of one database [64]. In practice this will likely be an upper bound, because not all blocks will reach the maximum allowed size b_{max}. Compared to standard blocking, the number of blocks b will also be much larger with suffix-array based indexing, because each BKV used in standard blocking will likely generate several different suffix values.

As Fig. 4.8 illustrates, the suffix-array based indexing technique has the drawback that variations or errors at the end of BKVs

will lead to record identifiers being inserted into different blocks, potentially resulting in missed true matches. This is especially of concern as empirical studies have shown that more data entry errors appear at the end of strings compared to their middle or beginning [214]. A modification to the suffix generation process can help overcome this drawback. Rather than only generating the suffix strings of a BKV, all substrings down to the minimum length l_{min} can be generated in a sliding window fashion (similar to the generation of q-grams). For example, for the string 'christen' and l_{min} = 5, this approach would generate the substrings: 'christen' (length 8); 'christe' and 'hristen' (length 7); 'christ', 'hriste', and 'risten' (length 6); and 'chris', 'hrist', 'riste', and 'isten' (length 5) [64]. This approach can help to overcome errors both at the beginning and end of BKVs, but the computational cost (similar to q-gram based indexing) increases significantly because a larger number of substrings are generated, and a record identifier is inserted into a larger number of blocks.

In an approach that is similar to the adaptive sorted neighbourhood technique [292] discussed in Sect. 4.5, a recently developed improvement to suffix-array based indexing is to merge blocks if their suffix values are similar to each other [265, 266]. This merging is based on an approximate string similarity function that calculates the similarity between consecutive strings in the sorted array of suffix values. If the similarity of a string pair is above a certain threshold, then their corresponding lists of record identifiers are merged to form a new combined larger block. As a result, suffix-array based indexing becomes more robust [265, 266].

As an example, assume the suffix array from the right-hand side of Fig. 4.8 has been generated. If the edit distance string comparison function (which returns a normalised similarity value between 0.0 and 1.0, as will be discussed in detail in Sect. 5.3) with a minimum similarity threshold set to t = 0.85 is used, three pairs of neighbouring suffix strings have a similarity above this threshold. The string pair 'atherina' and 'atherine' has a similarity s = 0.875, and thus their record identifier lists are merged into the list [r1,r3,r4]. The pair 'catherina' and 'catherine' has a similarity s = 0.889, and their lists are merged into [r3,r4]. Finally, the string pair 'therina' and 'therine' has a similarity s = 0.857 and their merged list is [r1,r3,r4].

A detailed theoretical analysis of this robust suffix-array based indexing technique has been presented by the developers [265]. A recent extension of this technique is to employ Bloom filters to improve the computational performance and reduce the amount of main memory that is required [266]. An experimental evaluation showed that with Bloom filters the number of database accesses needed during the generation of the index data structure can be reduced by up to 70 %.

4.8 Canopy Clustering

The indexing step can be seen as a clustering of the records in the databases to be matched or deduplicated in such a way that records that are similar to each other are inserted into the same cluster. Many clustering algorithms have high computation complexity [135]. Indexing, however, should be computationally cheap, and it must be feasible to generate the candidate record pairs in a fast and scalable manner. The canopy clustering approach achieves this goal by efficiently calculating distances between the BKVs [85, 185], and inserting records into one or more overlapping clusters. Each cluster then becomes a block from which candidate record pairs are generated.

The similarities between BKVs are calculated using either the Jaccard or the TF-IDF/Cosine (Term-Frequency / Inverse Document Frequency [288]) similarity measures, using tokens generated from the BKVs. Both these measures are also used in approximate string comparison functions, as will be described in detail in Chap. 5. Jaccard similarity, $sim_{Jaccard}$, is a measure based on the number of tokens two BKVs, b_1 and b_2, have in common, normalised by the union of the tokens contained in the two BKVs. If the function $token(b)$ returns the set of tokens in the BKV b, then the Jaccard similarity is calculated as

$$sim_{Jaccard} = \frac{|token(b_1) \cap token(b_2)|}{|token(b_1) \cup token(b_2)|}, \tag{4.7}$$

with |…| denoting the number of elements in a set [64]. When the TF-IDF/Cosine similarity, sim_{TFIDF}, is used instead of the Jaccard similarity, then TF-IDF weighting (described below) is taken into account when the similarity between two BKVs is calculated.

Depending upon the content of the BKVs, the tokens used in canopy clustering can either be words or q-grams (substrings of length q, as was discussed in Sect. 4.6). The canopy clustering indexing technique is based on an inverted index data structure where the index keys are the tokens extracted from the BKVs, and the index lists contain the identifiers of the records that contain this token in their BKVs. Figure 4.9 shows a small example of such a data structure. The Document Frequency (DF) is the count of how many times a token appears in a BKV in a certain record. For example, the token 'an' appears twice in the BKV of records 'r1' and 'r2' in the example. The DF can be calculated as the BKV for each record is being processed. The Term Frequency (TF), is the count of how many records in a database contain a certain token (the token 'an' for example appears in all three records in Fig. 4.9). Once a database has been loaded and the inverted index data structure for canopy clustering has been generated, the TF values can be converted into Inverse Document Frequencies (IDF) [288]. While there are several variations of how IDF can be calculated [135, 288], the basic approach is using the following equation:

(4.8) $$idf(t) = \frac{n}{tf(t)},$$

RecID	BKVs (Surname)	Sorted bigram lists
r1	hanlan	[(an,2), (ha,1), (la,1), (nl,1)]
r2	gansan	[(an,2), (ga,1), (ns,1), (sa,1)]
r3	gargan	[(an,1), (ar,1), (ga,2), (rg,1)]

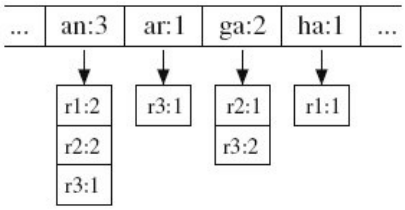


Figure 4.9: Canopy clustering example with BKVs based on surnames. The tokens used are the bigrams extracted from these surname values. The sorted bigram lists include Document Frequency (DF) counts. The Term Frequency (TF) and DF counts in the inverted index data structure in the right-hand side are used to calculate weights in the TF-IDF/Cosine similarity measure, while the Jaccard similarity measure will be calculated based only on the bigrams in the inverted index lists. This figure is adapted from [64]

where n is the number of records in the database and $tf(t)$ is the number of records in the database that contains the term t [288].

After the inverted index data structure is built, the canopy clustering indexing algorithm can be started [64]. The algorithm iteratively generates overlapping clusters by repeating the following steps [85].

1. The identifiers of all records in the databases to be matched are inserted into a set, P.
2. One record identifier, r_c, is randomly selected from P. This record identifier will become the centroid of a new canopy cluster, $C_i = \{r_c\}$.
3. Using the tokens in the BKV of record r_c, either the Jaccard, $sim_{Jaccard}$, or TF-IDF/Cosine similarity, sim_{TFIDF}, is calculated with all records $r_x \in P$ that have at least one token in common with r_c. Using the inverted index data structure, this can be accomplished very efficiently.
4. All records r_x that have a Jaccard orTF-IDF/Cosine similarity value above a loose threshold, t_l, are inserted in the canopy cluster C_i (i.e. either $sim_{Jaccard}(r_c, r_x) \geq t_l$ or $sim_{TFIDF}(r_c, r_x) \geq t_l$).
5. All records $r_x \in C_i$ that have a similarity with r_c above a tight threshold, t_t, are removed from the set P of records (i.e. either $sim_{Jaccard}(r_c, r_x) \geq t_t$ or $sim_{TFIDF}(r_c, r_x) \geq t_t$). The cluster centroid, r_c, is also removed from P.
6. As long as the set of records is not empty, $P \neq \emptyset$, go back to step 2.

The loose threshold, t_l, needs to be smaller or equal to the tight threshold, t_t, i.e. $t_l \leq t_t$. If both thresholds are set to the same value ($t_l = t_t$), then the generated canopy clusters will not be overlapping, and each record identifier will only be inserted into one cluster. If the thresholds are set to $t_l = t_t = 1.0$, then canopy clustering will generate clusters that are the same as the blocks generated by standard blocking.

Each cluster generated by canopy clustering will become one block, and candidate record pairs will be generated from all pairs of record identifiers within each cluster. Similar to the sorted neighbourhood approach, and q-gram and suffix-array based indexing, a specific candidate record pair will likely be generated from several clusters (blocks). However, each unique candidate record pair will only be compared once in the comparison step.

One drawback with the threshold-based canopy clustering approach is that the size of the generated clusters depends upon the distribution of BKVs, the similarity function used and the setting of the two similarity thresholds. Besides setting the two threshold values, there is no explicit way to limit the size of the generated clusters, and therefore it is not possible to limit the number of candidate record pairs generated.

An alternative way of using the two similarity thresholds is to generate clusters using a nearest-neighbour based approach [64]. Similar to the two thresholds t_l and t_t, two nearest-neighbour parameters are required. The first, n_l, is the number of record

identifiers whose records r_x have the highest similarity values with the record selected as cluster centroid, r_c. These n_l records are inserted into a canopy cluster C_i in step 4 of the algorithm. Of these n_l records, the n_t (with $n_t \leq n_l$) will then be removed from the set P of all record identifiers in step 5 of the algorithm. This approach requires that the n_l records in the set P that are most similar to the centroid record r_c need to be identified in each iteration of the algorithm.

The advantage of this nearest-neighbour based approach is that each canopy cluster generated will contain n_l record identifiers. The two parameters n_l and n_t allow explicit control of the number of candidate record pairs that will be generated. A drawback of this approach is however that the fixed size of the generated clusters, n_l, might mean that records that are similar to each other are not inserted into the same cluster. This drawback is similar to the drawback of the sorted neighbourhood approach where a fixed window size can lead to true matches not being compared with each other because they are not inserted into the same window. A recent experimental study has shown that threshold-based canopy clustering can achieve better results than the nearest-neighbour based approach, especially as the databases to be matched or deduplicated get larger [64].

4.9 Mapping Based Indexing

The idea behind mapping based indexing techniques is to convert the BKVs into objects that are mapped into a multi-dimensional space, such that the original similarities between BKVs are preserved [151]. Blocks are then generated similar to canopy clustering by inserting similar objects into the same clusters [151].

The distances between strings are calculated using an approximate string comparison function which needs to be a metric distance function (discussed in Sect. 5.1), such as one of the edit distance based functions (see Sect. 5.3 for details). The multidimensional space generated by the mapping process is created one dimension after another using a modification of the *FastMap* [105] algorithm called *StringMap* [151]. StringMap has a linear complexity in the number of strings that are mapped into the multi-dimensional space. The set of strings this algorithm works on is the set of all BKVs. Mapping based indexing consists of two phases.

- In the first phase, the algorithm iterates over the number of dimensions d (which needs to be chosen by the user). For each dimension, StringMap selects two *pivot strings* which ideally are far apart from each other measured using the selected string similarity function. These two strings are used to form the orthogonal directions of the multi-dimensional Euclidean space that is generated.

  An iterative farthest-first algorithm can be used to find the two pivot strings [151]. This algorithm starts by randomly selecting a string, and then finding the string that is farthest away from the first string. In the second iteration, the string in the set of all strings that is farthest away from the second string is found. This process is repeated several times. The last two strings found will be selected as the two pivot strings.

  Once two pivot strings have been selected for a dimension, the coordinates of all other strings for this dimension are calculated based on the two pivot strings. The process is repeated until all dimensions d have been created and a d-dimensional object has been generated for each string.

  A crucial parameter required in this algorithm is the dimensionality d of the space that is generated. Experiments by the developers of StringMap have shown that good results can be achieved with $15 \leq d \leq 25$ [151]. A second crucial issue for mapping based indexing is the type of data structure that is used to store the multi-dimensional objects.

  The developers of StringMap have used an R-Tree [151], which is a popular data structure for efficient storage and retrieval of multi-dimensional objects. However, like most tree-based multi-dimensional index data structures, the higher the dimensionality of the objects to be stored is, the less efficient an R-Tree data structure becomes. The reason for this degradation in efficiency is that with increased dimensionality, more subtrees in an R-tree need to be searched to find objects that are similar to a query object. For dimensions larger than 15 to 20, nearly all objects in a tree-based multi-dimensional index need to be accessed when similarity searches are conducted [3]. This problem is commonly known as the *curse of dimensionality* [135].

- In the second phase of mapping based indexing, similar to canopy clustering based indexing, groups or clusters of objects (that refer to BKVs) are extracted from the multi-dimensional index, and all records that contain any of these BKVs are inserted into the same block. As with canopy clustering, clusters (and thus blocks) can be generated using a similarity threshold or a nearest-neighbour based approach [64]. It is difficult to predict the size of the generated blocks when using a threshold based approach, because block sizes depend upon the distribution of the objects in the multi-dimensional space. Their distribution depends upon the BKVs, the string distance function employed in the mapping and the dimensionality d of the space generated.

A double-embedding based approach to indexing has recently been proposed [1]. The idea of this approach is to first map the BKVs into a space of k dimensions using the StringMap algorithm [151]. This is followed by a mapping of the objects in the k dimensional space into a lower dimensional space (with $k' < k$ dimensions) using the FastMap algorithm [105]. A binary KD-tree data structure is combined with a nearest neighbour based similarity approach to extract similar objects that are then used to generate blocks [1]. An experimental evaluation by the developer of this technique using names and addresses from a publicly available Canadian voter's database has shown improvements in the run-time of between 30 and 60 % reduction compared to the original StringMap based mapping approach, while at the same time achieving the same matching accuracies [1].

4.10 A Comparison of Indexing Techniques

To illustrate the differences in performance of the indexing techniques described in this chapter, this section presents some experimental results of a recent comparative evaluation [64]. This study was based on three small data sets that have previously been used by the data matching community for comparative evaluations. These and other data sets used in data matching research will be discussed in more detail in Sect. 7.5. Table 4.3 provides basic details of these three data sets. They are all available in the *SecondString* toolkit.[1] The true match status of all record pairs is known in all three data sets.

Table 4.3: Details of the data sets used for the comparative evaluation presented in Fig. 4.10

Data set name	Description	Task	Number of records	Total number of true matches
Census	Synthetic data generated by the US Census Bureau	Linkage	449/392	327
Cora	Bibliographic records of machine learning papers	Deduplication	1,295	17,184
Restaurant	Records from Fodor and Zagat restaurant guides	Deduplication	864	112

All indexing techniques described in Sects. 4.4–4.9 (and variations of them) have been implemented in the programming language Python as part of the FEBRL[2] open source record linkage system [61]. FEBRL will be described in detail in Sect. 10.2.4. The results presented here are based on experiments conducted on an otherwise idle compute server with two 2.33 GHz quad-core CPUs and 16 GB of main memory, running Linux 2.6.32 (Ubuntu 10.04) and using Python 2.6.5. Further results, such as the quality of the candidate record pairs generated, are available in a detailed experimental evaluation [64].

The results presented in Fig. 4.10 show the three basic measures time used, number of candidate record pairs generated, and main memory required. Other measures that are commonly used to evaluate indexing techniques will be discussed in Sect. 7.3. As these results illustrate, there are large variations between the different indexing techniques for all three measures. Not just are there differences between indexing techniques, but also for the same technique when applied to different data sets.

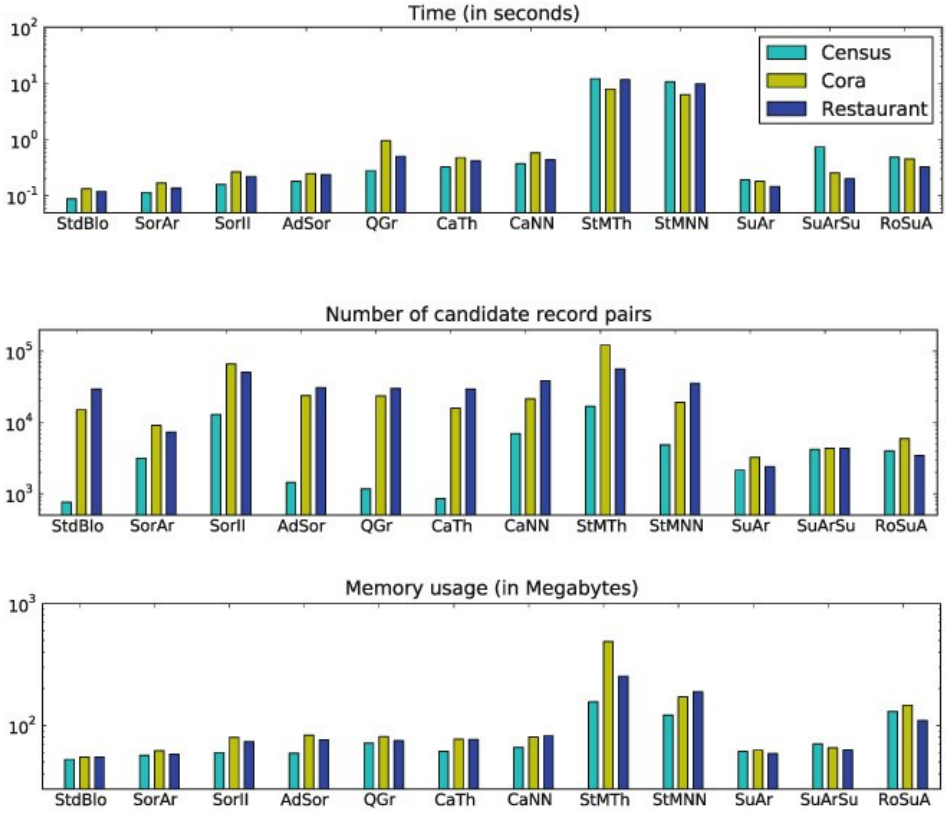


Figure 4.10: Experimental results for the three small data sets presented in Table 4.3. These results are based on an earlier

experimental evaluation by the author [64]. The abbreviations used for the different indexing techniques are—'StdBlo' refers to the standard blocking technique; 'SorAr' to the array based sorted neighbourhood approach, 'SorII' to the inverted index based sorted neighbourhood approach and 'AdSor' to the adaptive sorted neighbourhood technique; 'QGr' refers to q-gram based indexing; 'CaTh' and 'CaNN' to the threshold and nearest-neighbour based canopy clustering techniques; while 'StMTh' and 'StMNN' refer to the threshold and nearest-neighbour based StringMap indexing techniques; finally, the three suffix-array based approaches are labelled as 'SuAr' (the original suffix array technique), 'SuArSu' (the variation where all substrings are generated), and 'RoSuA' refers to the robust suffix array approach

Mapping based indexing is the overall slowest technique, followed by q-gram based indexing, canopy clustering and two variations of suffix-array based indexing. Mapping based indexing also produces a large number of candidate record pairs. The more simpler techniques, such as standard blocking, the sorted neighbourhood based approaches and suffix-array based indexing, are faster and require less memory. A major result of this and other comparative studies [20, 64] is that an important factor for successful indexing is the appropriated definition of blocking or sorting keys. How to automatically generate optimal blocking keys based on supervised learning approaches will be discussed in Sect. 4.12 below.

[1]http://secondstring.sourceforge.net

[2]Available from: https://sourceforge.net/projects/febrl/

4.11 Other Indexing Techniques

Besides the indexing techniques (and their variations) covered in Sects. 4.4–4.9, different approaches aimed at reducing the time needed to match two databases or deduplicate a single database have been explored.

An iterative blocking technique has recently been proposed that uses the information gained when records within a block are compared to improve the records contained in other blocks [277]. For example, when two records in a block are compared, the resulting merged record can contain information that leads to the merged record being inserted into another block, because the merged record might have a different BKV than the two original records. The issues involved in merging records that have been classified as matches will be covered in Sect. 6.12. The process of indexing a database into blocks (as discussed in Sect. 4.4) with this technique is therefore not static and done only once, but rather blocks are iteratively refined as record pairs are being compared and merged. The experimental evaluation presented by the authors showed that such iterative blocking can be both more accurate and faster at the same time [277].

While all indexing techniques discussed so far only use the attribute values in an individual record to decide into which block to insert the record, a recently developed technique uses the relationship of a record with other records to improve the quality of the indexing process [200]. The idea is to build a relationship graph where records are nodes and relationships between records are vertices. A relationship can for example be between individuals who live at the same address or who share the same telephone number, or between co-authors who have contributed to the same article. Blocks are then generated by randomly selecting a record (similar to canopy clustering and mapping based indexing) and including all records into the block that are connected to this record in the relationship graph. This so-called *semantic blocking* approach achieved much improved matching quality on several data sets while having similar computational requirements as standard blocking or the sorted neighbourhood approach [200].

Techniques that are orthogonal to indexing (i.e. that can be employed complementary to indexing techniques) include running data matching or deduplication on a parallel computer or in a distributed computing environment, where each processing unit will conduct the matching of a subset of the original databases. This topic will be further covered in Sect. 9.5. Related to parallel data matching is the BigMatch technology developed by the US Census Bureau [295]. The challenge faced by this organisation is that very large databases (some containing billions of records) need to be matched with 'smaller' databases (still containing millions of records) on a regular basis. The approach implemented in BigMatch is to load and process the 'smaller' of the two databases into an inverted-index based data structure [288], assuming this data structure fits into the main memory of a large parallel compute server. The indexing step can then be carried out by a single scan through the larger of the two databases while still using several blocking criteria. For each of these blocking criteria, a file will be written that contains plausible matches. Each of these files will be smaller than the original large database. More detailed matching is then carried out separately between the smaller database and each of the generated blocking files [295].

Another approach is to reduce the time required in the expensive detailed comparison of the candidate record pairs that are generated in the indexing step. Comparison functions will be covered in detail in the following chapter. Many of these functions are computationally expensive, so limiting the number of comparisons that need to be conducted can provide substantial performance improvements. Two recently developed techniques [91, 193] explore how an early decision can be made if a candidate record pair is classified as a non-match, and therefore does not need to be compared in detail across many of its attributes. The first proposed approach assumes a distributed environment (such as a crime investigation by a police officer

who needs to gather information from several distributed law enforcement databases). A matching tree is developed (similar to a decision tree) that allows an early decision if a candidate record pair will be a match or not. This technique can lead to a significant reduction in communication overhead [91]. A second recent approach assesses the attribute comparisons for a candidate record pair such that a match or non-match decision can be made as early as possible [193]. This is achieved by an optimal ordering of the attributes that are used in the comparison step. Such an optimisation is especially important if the records in the databases to be matched contain many attributes that need to be compared.

Special indexing techniques have recently been proposed for real-time data matching [69, 70], where the aim is to match a stream of query records that contain entity information with a large database that contains records of known entities, such as known criminals or people that have a bad credit history. If the matching of a query record is required in (near) real-time, such as for a police investigation for example, then the amount of permitted matching time for each query record is limited. The matching task then becomes similar to the task of Web search which is successfully carried out by various commercial search engines on very large data collections. The topic of real-time matching will be further discussed in Sect. 9.3.

A large body of work related to indexing has also been conducted by the database community. The two areas relevant to indexing are similarity joins and uncertain or probabilistic databases. The objective of a similarity join is to enable an efficient and scalable approximate join function that calculates the similarities between attribute values in two database tables using a string similarity function such as edit distance or Jaccard distance (to be discussed in the following chapter). Various techniques have been developed that facilitate similarity joins within database environments [22, 127, 170, 232, 272, 289]. The area of uncertain or probabilistic databases is concerned with information that is probabilistic in nature and thus has some uncertainty attached to each value [2]. Indexing techniques for uncertain data have been developed based on standard inverted index and tree-based approaches [216, 298]. Finding similarities between objects in probabilistic databases has recently also received some attention [28]. None of these techniques has however been directly applied in the domain of data matching.

The information retrieval community, besides developing techniques to improve inverted index based indexing for Web search [21], has also investigated techniques to allow efficient detection of duplicate documents returned by Web search engines [131]. Because Web documents are commonly much larger than the records used in data matching and contain more detailed information, a main challenge is to extract relevant parts of documents to allow efficient indexing for scalable duplicate detection [139].

4.12 Learning Optimal Blocking Keys

The blocking or sorting keys that are required for all indexing techniques presented in this chapter are traditionally being defined manually by data matching and domain experts. As experimental evaluations have shown [20, 64], finding optimal blocking key definitions that lead to high quality matching results is challenging. The aim of a good blocking key definition is to get a set of candidate record pairs that includes as many true matches as possible while at the same time keeping the total set of candidate record pairs as small as possible. An alternative way to manually define blocking keys is to learn them automatically from the data that are to be matched.

Two machine learning based approaches to define optimal blocking keys have recently been proposed [34, 188]. Both work in a supervised learning fashion and require training data in the form of record pairs that correspond to true matches and true non-matches. The learning process generates candidates of blocking keys, and using those training examples the candidates that achieve the highest coverage and highest accuracy are selected. The two measures coverage and accuracy are commonly used to evaluate rule-based classification approaches [135]. Coverage measures the number of true matching candidate record pairs in the training set that are covered by a blocking key definition, while accuracy measures how many of the candidate record pairs in the training set that are covered by a blocking key definition correspond to true matches.

The approach developed by Michelson et al. [188] learns so-called *blocking schemes* (blocking criteria) made of {*method*, *attribute*} tuples, where the *method* is a function which compares the values of the given *attribute* from two records. Example methods are *first-1-match*, which returns true if the first characters in the attribute values of two records are the same and false otherwise; *first-3-match*, which returns true if the first three characters are the same, or *token-match*, which returns true if two attribute values have at least one token (word or q-gram) in common. The actual learning algorithm is based on a variation of the sequential covering algorithm [135]. This algorithm learns one rule (one blocking scheme) at a time. It starts with a set of candidate blocking schemes that includes all possible *methods* applied on all available *attributes*. The coverage and accuracy of each possible {*method*, *attribute*} tuple is then calculated using the record pairs in the training data. Individual tuples are combined into conjunctions to improve the accuracy of the learned blocking schemes. The conjunction of {*method*, *attribute*} tuples with the highest coverage and accuracy is selected, and the record pairs that are covered by that conjunction of tuple are removed from the training data set [188]. The process is continued until all training record pairs are covered by a tuple.

An approach similar to learning optimal blocking keys was presented by Bilenko et al. [34]. Instead of learning one rule at a

time using the sequential set covering algorithm, blocking schemes are learned by solving an optimisation problem that was shown to be equivalent to the red-blue set cover problem [34]. Both this and the technique proposed by Michelson et al. [188] can in principle be employed with any of the indexing techniques presented in this chapter.

4.13 Practical Considerations and Research Issues

The most important aspects to consider in the indexing step for a practical data matching or deduplication exercise are how to define the blocking key or keys, and what indexing technique to employ. The selection of which attributes to use in a blocking key definition depends upon the number of unique values of an attribute, their frequency distribution and also how many records have an empty value in an attribute. These basic statistics can be gathered through data profiling or data exploration tools, as was previously discussed in Chap. 3. Ideally, attributes that have no missing values and that have a nearly uniform frequency distribution of their values are preferred. The reason for this is that using such attributes will result in blocks or clusters that are of similar sizes, compared to when the values of an attribute follow for example a Zipf-like frequency distribution [64].

The number of blocks and the distribution of their sizes can be further influenced by the use of a (phonetic) encoding function during the generation of BKVs. The choice of a phonetic encoding function depends upon the language of the values that will be used to generate the BKVs. Most phonetic encoding functions, including those presented in Sect. 4.3, have been developed for English names only. For any database that contains either name values from a language other than English, or that contains multilingual name values, the use of phonetic encoding functions should be carefully evaluated, and variations of such functions (that have been appropriately adapted to a certain language) should be considered.

Which indexing technique to use for a certain data matching or deduplication project is a second important practical aspect that needs to be carefully considered. With different indexing techniques, there is usually a trade-off between how many candidate record pairs are generated and the resulting quality of the achieved matching. The more record pairs are removed by an indexing technique from the set of all possible pairs, the more likely some true matching pairs will be removed as well. For databases that contain data of low quality, employing an indexing technique that inserts records into several blocks or clusters will be of advantage compared to employing a technique that inserts each record into one block only. On the other hand, if the data to be matched or deduplicated are of good quality, then using the traditional blocking technique (that inserts each record into one block only) might be appropriate.

One avenue of research in the area of indexing could tackle the challenge of developing multilingual phonetic encoding functions that can be applied on databases that contain names from different languages and cultures. Another area of research is the development of indexing techniques that are scalable, while at the same time also highly efficient. This means that ideally the number of candidate record pairs that are generated only increases linearly with the size of the databases that are matched, while a high matching quality is still achieved by keeping all (or a very high portion of) true matching record pairs in the generated candidate record pairs.

All the indexing techniques that have been presented in this chapter are heuristic approaches. Their aim is to split the records in a database (or databases) into blocks or clusters (that potentially overlap) in such a way that all records that match with each other are inserted into the same block, and records that are not matching are inserted into different blocks. An ultimate goal of research on indexing for data matching is the development of techniques that generate blocks such that it can be proven that (1) all comparisons between records within a block will have a certain minimum similarity with each other (according to some similarity metric), and (2) the similarity between records in different blocks is below this minimum similarity. Specifically, if r_i and r_j are two records, and $sim(r_i, r_j)$ is a similarity measure (such as one of the techniques described in the following chapter) applied to a pair of records (with $sim(r_i, r_j) = 1$ if $r_i = r_j$ and $sim(r_i, r_j) = 0$ if r_i and r_j are totally different from each other), then an optimal indexing technique would generate blocks B such that $sim(r_i, r_j) \geq t$, $\forall\ r_i, r_j \in B_k$, and $sim(r_i, r_j) < t$, $\forall\ r_i \in B_k$, $r_j \in B_l$, $B_k \neq B_l$, for some threshold $0 \leq t \leq 1$.

4.14 Further Reading

A recent survey by the author arguably provides the most comprehensive comparison of indexing techniques for data matching presented so far [64]. The survey analyses the computation complexity of different indexing techniques, measured as the number of candidate record pairs that are expected to be generated if certain data distributions are assumed. A detailed experimental evaluation on both synthetic and real-world data sets is also provided in this survey. These experiments are highlighting the significant differences in both performance and accuracy that are achieved by various indexing techniques.

Two publications discuss implementation details of how indexing techniques can be employed in industry to match large databases. A TF-IDF based approach, that is similar to canopy clustering described in Sect. 4.8, was used by Koudas et al. in an SQL database environment to match customer information [170]. Various enhancements were presented that can lead to significant performance improvements when large databases are matched. More recently, Weis and Naumann discuss

extensions to the sorted neighbourhood approach to allow the scalable deduplication of a customer relationship database containing records in XML format for more than 60 million individuals [272]. The recent introductory book to duplicate detection by the same two authors also contains a discussion of several indexing techniques [195].