



University of Hamburg
Faculty of Mathematics,
Informatics and Natural Sciences

Bachelor Thesis

Search-space reduction techniques for duplicate detection in probabilistic data

Steffen Friedrich (mat. no. 5944434)

Wolfram Wingerath (mat. no. 5945090)

BSc Informatics, 6th semester
7friedri@informatik.uni-hamburg.de

BSc Informatics, 6th semester
7wingera@informatik.uni-hamburg.de

1st assessor: Prof. Dr. Norbert Ritter

2nd assessor: Prof. Dr.-Ing. Wolfgang Menzel

Solum certum nihil esse certi

– *Pliny the Elder*

Foreword

Since we began our studies of computer science three years ago, we have been introduced to many topics and discovered our great interest for the theoretical aspects of computer science. Our favourite field of research, however, was databases.

We got an introduction to relational databases in our third semester by Prof. Dr. Norbert Ritter that was widely extended in the subsequent lecture by him and Prof. Dr.-Ing. Wolfgang Menzel one semester later. In our fifth semester, we participated in the Hanseatic Mainframe Summit 2009 where we learned of IBM's mainframe technology, the huge scale of databases in modern finance business and their complexity by IBM employees and IBM customers. Our interest for databases had grown with every course we had taken on the subject and hence we chose to write our bachelor thesis about a related topic.

We would like to thank Prof. Dr. Norbert Ritter and Prof. Dr.-Ing. Wolfgang Menzel for having agreed to supervise and assess this thesis.

Our special thanks go to Fabian Panse for his helpful criticism and support, especially during the last weeks.

Steffen Friedrich

Wolfram Wingerath

Hamburg, November 25th, 2010

Abstract

English

Search space reduction (SSR) techniques are used today to minimise the computational costs of duplicate detection in certain data, but there are no SSR techniques available for probabilistic data yet. The purpose of this thesis is to investigate how the SSR techniques Blocking and the Sorted Neighborhood Method (SNM) used for certain data can be adapted to *probabilistic* data. Another aim is to compare the adapted SSR techniques regarding their effectiveness and the quality of their results.

Among others, uncertainty-lineage databases are presented where every tuple consists of one or more mutually exclusive representations of one and the same real-world entity, so-called tuple alternatives. Furthermore, three strategies using certain key values, namely *multi-pass over possible worlds*, *key per tuple* and *key per alternative*, and one SNM strategy using probabilistic key values for search space reduction are discussed. The implementations of the adapted SSR strategies were tested on generated (synthetic) probabilistic data and the achieved results are compared by the metrics pairs completeness, reduction ratio and precision as well as by their respective execution times.

The comparison results indicate that the key per alternative strategy outperforms the other strategies, because it resulted in the best pairs completeness with an acceptable precision and was comparatively fast.

German

Techniken der Suchraumreduzierung (SSR, von engl. *search space reduction*) werden heutzutage benutzt, um den Rechenaufwand der Duplikatenerkennung in sicheren Daten (engl. *certain data*) zu minimieren. Es gibt jedoch noch keine SSR-Techniken für die Duplikatenerkennung in probabilistischen Daten. Zielsetzung dieser Arbeit ist es, zu untersuchen, wie die zur Duplikatenerkennung in sicheren Daten eingesetzten SSR-Techniken *Blocking* und *Sorted Neighborhood Method* (SNM) für den Einsatz in *probabilistischen Daten* angepasst werden können. Weiterhin sollen die angepassten SSR-Techniken bezüglich ihrer Effektivität und der Qualität ihrer Resultate verglichen werden.

Unter anderem werden *uncertainty-lineage databases* präsentiert, bei denen jedes Tupel aus mehreren Repräsentationen ein und derselben Realweltentität, sog. *Tupelalternativen*, besteht, welche sich gegenseitig ausschließen. Weiterhin werden die drei Strategien *multi-pass over possible worlds*, *key per tuple* und *key per alternative* diskutiert, welche sichere Schlüsselwerte (engl. *certain key values*) verwenden, sowie auch eine SNM-Strategie, die mit probabilistischen Schlüsselwerten arbeitet. Die Implementationen wurden auf generierten (synthetischen) probabilistischen Daten getestet und die Resultate werden anhand der Metriken *pairs completeness*, *reduction ratio* und *precision* sowie anhand der jeweiligen Laufzeiten verglichen.

Die Vergleichsresultate deuten darauf hin, dass die Strategie *key per alternative* die anderen Strategien übertrifft, weil mit ihr die beste *pairs completeness* erzielt werden konnte, sie eine akzeptable *precision* lieferte und vergleichsweise schnell war.

Contents

Abstract	3
1. Introduction	7
1.1. Motivation	7
1.2. State of research	8
1.3. Goals	9
1.4. Chapter outline	9
2. Basics	11
2.1. Modeling uncertainty	11
2.1.1. Probabilistic databases	11
2.1.2. Lineage databases	13
2.1.3. Uncertainty-lineage databases	14
2.2. Data integration	16
2.2.1. Normalisation of data models	17
2.2.2. Schema management	17
2.2.3. Duplicate detection	18
2.2.4. Data fusion	20
2.3. Search space reduction in certain data	20
2.3.1. Evaluation metrics	22
2.3.2. Blocking	25
2.3.3. Sorted Neighborhood Method	26
2.3.4. Other techniques	29
3. Adapting certain data SSR techniques to probabilistic data	31
3.1. Generating certain key values for search space reduction	31
3.1.1. Multi-pass over possible worlds	32
3.1.2. Key per tuple	34
3.1.3. Key per alternative	36
3.1.4. Modification: combining key values	37
3.2. Processing probabilistic keys	39
4. Experiments	41
4.1. Testing framework	41
4.2. Test data	43

4.3. Experiment configuration	45
4.3.1. The database size	45
4.3.2. The database configuration	46
4.3.3. The SSR technique configuration	47
4.4. SSR experiments	49
4.4.1. Possible worlds experiments	49
4.4.2. Key per tuple experiments	50
4.4.3. Key per alternative experiments	52
4.4.4. Probabilistic key experiments	53
4.4.5. Experimental results	54
5. Summary and future prospects	57
Appendix	59
A. Duration experiment series	59
B. Database size experiment series	60
C. Database configuration experiment series	61
D. SSR technique configuration experiment series	63
E. SNM: possible worlds experiment series	64
F. Blocking: possible worlds experiment series	65
G. SNM: key per tuple experiment series	66
H. Blocking: key per tuple experiment series	67
I. SNM: key per alternative – combined keys experiment series	68
J. Blocking: key per alternative – combined keys experiment series	69
K. SNM: key per alternative experiment series	70
L. Blocking: key per alternative experiment series	71
M. SNM: probabilistic keys experiment series	72
N. Evaluation experiment series	73
O. Bigram Indexing: all alternatives experiment series	74
P. Evaluation experiment series 50k	76
Bibliography	77
List of Figures	81
List of Tables	83
Statutory Declaration	85

1. Introduction

We begin the first chapter by motivating the use of duplicate detection in probabilistic data. For that purpose, we give a short overview over the state of research in the field of probabilistic databases and their integration. Furthermore, we point out the role of duplicate detection in the data integration process and state why search space reduction techniques are required. Finally, we define our goals and provide a chapter outline.

1.1. Motivation

Many applications use multiple data sources, so that it is often necessary to merge information from several databases into one single database. This process is referred to as *data integration*.

One of the major problems with data integration, however, is that data of different sources may overlap, i.e. two or more sources may have contradictory information on the same real-world entity¹. So, even if the sources only contain certain data, the result of the data integration process can be uncertain, because it is unknown which of the conflicting source data sets represents reality best.

Data in the different data sources contradicting one another can make data integration extremely hard. If the integrated database is a certain database, a choice has to be made which database to trust more, as only one possible world can be represented. Making such a choice during the integration process – if possible at all – can be very time-consuming and thus also expensive, as it cannot be fully automated.

In [vKdK09], van Keulen et al. mention a thumb rule stating that about 90 percent of the development effort of the data integration process is devoted to the conflict resolution of those few conflict cases.

Several approaches to overcome this problem have been proposed, e.g. van Keulen et al. suggest the integration of certain data into a probabilistic database, so that conflicts do not have to be resolved during the integration process, but instead the *uncertainty can be modeled directly into the database* ([vKdK09]). The conflict-free data could be used right away and the resolution of uncertainty could be done later.

Traditionally, a database is considered to represent a part of reality. We call such a representation a *certain world*. But in many applications, data are known to be uncertain

¹By real-world entity, we do not only mean objects and persons, but also more abstract things that can be modeled into a database, such as buying transactions.

or imprecise. In such cases it does not suffice to represent reality by just one certain world, as there actually are many *possible worlds*.

Though there already have been many applications that would profit from uncertain databases, there are still no commercial products available on the market. But probabilistic databases have been researched in recent years and there are some prototypes for the relational data model such as MayBMS, MYSTIQ and Trio as well as for the XML data model such as IMPrECISE or PXML ([vKdK09], [dK08]). We only cover Trio in this work.

A tuple in the probabilistic databases considered in our work consists of one or more mutually exclusive representations of one and the same real-world entity, so-called *tuple alternatives*, that are stored with their respective probabilities.

1.2. State of research

In contrast to the integration of certain data, the integration of probabilistic data has not been researched much as the development of probabilistic databases is still in an early stage.

An essential step of data integration is the identification of *duplicates*, i.e. different tuples corresponding to the same real-world entity. This process is called *duplicate detection* and is accomplished by tuple comparisons. Generally, comparing all pairs of tuples from different source databases is not feasible due to their sheer number. Hence, the process of increasing the efficiency by reducing the number of comparisons, referred to as *search space reduction*, is a crucial part of duplicate detection.

In [PvKdKR09], duplicate detection in the process of integrating probabilistic data is discussed and some ideas to adapt existing search space reduction techniques to probabilistic data are outlined. In this work, we first investigate methods of search space reduction used in the duplicate detection process of certain data integration and then focus on how those methods can be adapted to the integration of *probabilistic* data. The methods we concentrate on are Blocking and the Sorted Neighborhood Method (SNM).

Blocking was proposed as a means of search space reduction already over 40 years ago ([New67]). According to [Jar89], the idea of Blocking is to compute a key value for each tuple, to divide the tuples into mutually exclusive blocks by their key values and finally to compare only tuples in the same block. But since corresponding tuples can differ (e.g. due to typos) and thus also their key values, tuples with similar (and not necessarily equal) key values are assigned to the same block. Of course, the key values have to be computed in such a way that corresponding tuples have similar key values.

The Sorted Neighborhood Method as described in [HS98] consists of three steps: The first step is to compute a key value for each tuple from relevant attributes. In the next step, the tuples are sorted by their keys and finally, in step three, every tuple is compared only to a fixed number of its neighbours in the sorted list of tuples. Again, corresponding tuples

should have similar key values to ensure they are neighbouring tuples in the sorted tuple list and hence are compared.

1.3. Goals

The goal of our work is to discuss three variants of adapting the two methods mentioned above as introduced in [PvKdKR09], implement those variants in Java and finally evaluate their effectiveness and the quality of the results, i.e. the number of rejected matches. We cover the following variants using *certain* key values in this work:

multi-pass over possible worlds: Since probabilistic databases are sets of *certain* databases, search space reduction can be performed on each without any adaptations. As the number of possible worlds in a probabilistic database can be very large, running passes over *all* of them seems to be infeasible and using only a selection of possible worlds seems more appropriate. But as running multiple passes on the most probable worlds is likely to yield very similar results, we take more than one selection approach into account.

key per tuple: Another variant is to resolve the uncertainty on tuple-level by computing one certain key value for each tuple from its alternatives. By now, there are conflict resolution techniques that seem suitable for this purpose, for example those used for fusion of certain data.

key per alternative: It is also possible to create one key value for each tuple alternative in the database. Thus a tuple can be assigned to several blocks (Blocking) or can appear several times in the sorted list (SNM).

In addition to the above certain key techniques, we examine the possibility of using *probabilistic key values* for search space reduction which is also proposed in [PvKdKR09].

1.4. Chapter outline

After the introduction of probabilistic and lineage databases, we describe the ULDB data model and Trio database management system in particular in Chapter 2. We also give an overview over the information integration process. In more detail, we survey search space reduction strategies as a part of the duplicate detection in data integration of *certain data* with focus on Blocking and the Sorted Neighborhood Method (SNM).

In Chapter 3, we investigate how those methods used for certain data integration can be adapted to the integration of *probabilistic* data.

We describe our testing framework and our test data in Chapter 4. We also compare the adapted techniques of search space reduction by experimental results w.r.t. effectiveness and quality.

Finally, a summary of our work and future prospects are presented in Chapter 5.

2. Basics

In this chapter, we introduce the concepts our work is based on.

We begin with a view on some data models extending the relational data model primarily by concepts for modeling uncertainty. Then we give an overview over data integration in order to help locating search space reduction in this process. Finally, we survey search space reduction in *certain data* with focus on the techniques Blocking and the Sorted Neighborhood Method.

2.1. Modeling uncertainty

In this section, we begin with the introduction of probabilistic and lineage databases, then describe uncertainty-lineage databases and present Trio database management system as the only implementation according to [Wid08] and [BSH⁺08].

2.1.1. Probabilistic databases

According to [DS04], a probabilistic relational database (PDB) is a probability distribution over many possible worlds, each of which is one possible state of the database. A probabilistic database is a special form of an uncertain database (UDB), which is the general term for a database that represents a set of possible worlds. In the typical notion of probabilistic relational databases, a probability value in the range of [0,1] is assigned to every tuple t_i representing the probability of its existence. This probability value is mostly called *confidence* $c(t_i)$.

An essential shortcoming of such probabilistic databases, however, is that they support only uncertainty about a tuple's existence, and not about its attribute values.

In order to model not only uncertainty on both tuple and attribute level, but also dependencies between attribute values, the *x-tuple* concept was introduced. An x-tuple consists of one or more mutually exclusive representations of a real-world entity, so-called *tuple alternatives*. So, tuple alternatives within an x-tuple represent the same real-world entity in different possible worlds and differ in attributes that were declared uncertain at schema creation time. We write $t_{i,j}$ when we refer to the j^{th} alternative of tuple t_i .

X-relations are sets of x-tuples and correspond to relations known from common RDBMSs. For convenience, we call x-tuples, x-relations and tuple alternatives also plainly tuples, relations and alternatives respectively.

The confidence value for an alternative is optional and represents its probability to be the

correct representation of its respective real-world entity. The confidence values of all alternatives of one tuple sum up to the probability of this tuple's existence; if this sum is smaller than 1, it is implicitly uncertain whether the tuple exists at all. Tuples whose existence is uncertain are called *maybe x-tuples*. In order to model tuple uncertainty explicitly and independently from confidence values, x-tuples can be marked with a *maybe annotation* (?) indicating possible non-existence. When we refer to the alternative of a maybe tuple t_i that represents its non-existence, we write $t_{i,?}$. Since the confidence of a tuple t_i is $c(t_i)$, the non-existence alternative of t_i has a confidence of $c(t_{i,?}) = 1 - c(t_i)$.

studios (s)			
	name	c	
t_{s_1}	Republic Pictures	1	
t_{s_2}	Twentieth Century Fox	1	
t_{s_3}	Zulu Pictures	-	?

movies (m)					
	title	year	studio	c	
t_{m_1}	Batman	1966	Twentieth Century Fox	0.8	
		1967	Republic Pictures	0.2	
t_{m_2}	Catwoman	1969	Twentieth Century Fox	0.6	?

view (v)				
	studio	movies	c	
t_{v_1}	Republic Pictures	0	0.8	
		1	0.2	
t_{v_2}	Twentieth Century Fox	0	0.08	
		1	0.32	
		1	0.12	
		2	0.38	
t_{v_3}	Zulu Pictures	0	-	?

Figure 2.1.: A traditional probabilistic database (PDB) with two x-relations movies and studios holding certain x-tuples (t_{s_1} and t_{s_2}), maybe x-tuples (t_{m_2} and t_{s_3}) and an x-tuple with several alternatives (t_{m_1}). The view shows all studios with their respective numbers of produced movies. Note that tuples $t_{v_{2,2}}$ and $t_{v_{2,3}}$ are identical, but have been derived from different alternatives.

Figure 2.1 illustrates a probabilistic database with two x-relations and a view. The studios relation holds studio names and the movies relation holds information about title, production year and studio of movies. The view is composed of studio names from the studios relation with their respective numbers of produced movies, which are computed from the movies relation.

Tuples t_{s_1} and t_{s_2} are certain tuples as their confidences are both 1. Since there is no confidence value assigned to t_{s_3} , its existence is not certain and it is marked with a maybe annotation.

The movies relation illustrates that the x-tuple concept does not only offer modeling un-

certain attribute values for a tuple, but also dependencies between them. For example, the year and studio attribute values of tuple t_{m_1} occur only in two combinations: If the movie is from 1966, it was produced by Twentieth Century Fox, whereas it is a Republic Pictures movie, if the movie is from 1967. The combinations of 1966 and Republic Pictures or 1967 and Twentieth Century Fox as production year and studio respectively are not possible. As a consequence, finding out the actual value for one of the attributes would resolve uncertainty for both attribute values.

Besides the movie Batman, there is another movie entitled Catman, which exists with a probability of only 60 percent.

In the view, all studios are listed with the number of movies they might have produced and their respective confidences. Tuples $t_{v_{2,2}}$ and $t_{v_{2,3}}$ are identical, but have been derived from different alternatives. It is important to note that $t_{v_{2,2}}$ only exists, given that Twentieth Century Fox's only movie was Batman. In more formal words, $t_{v_{2,2}}$ has been derived from $t_{s_{2,1}}$, $t_{m_{1,1}}$ and $t_{m_{2,?}}$. Hence, its confidence value is $c(t_{v_{2,2}}) = c(t_{s_{2,1}}) \cdot c(t_{m_{1,1}}) \cdot c(t_{m_{2,?}}) = 1 \cdot 0.8 \cdot (1 - 0.6) = 0.32$. Accordingly, $t_{v_{2,3}}$ is only correct, if Twentieth Century Fox only produced the movie Catwoman and *not* Batman.

The above makes one conceptual flaw of purely probabilistic databases obvious: The query result tuple alternatives are in no way connected to the alternatives they were derived from, and hence such a database cannot answer the question, *where* the data have been derived from. As a consequence, confidence values have to be computed *along with* the actual data of the query result. This can lead to a tremendous computation overhead for complex queries, if only few confidences are needed. Furthermore, it cannot be determined which alternatives are mutually exclusive, once the result has been computed. For instance, it cannot be derived from the view that the alternatives $t_{v_{1,2}}$ and $t_{v_{2,4}}$ cannot be present in the same possible world, although it is quite obvious: If Batman is a Republic Pictures movie, $t_{v_{1,2}}$ is true, but in this case Twentieth Century Fox can at most have produced one movie, namely the movie Catman, and so $t_{v_{2,4}}$ cannot not be true.

A concept to keep track of the derivation of data is called *lineage* and is introduced in the next subsection.

2.1.2. Lineage databases

In a lineage database (LDB) according to [BSH⁺08], tuples are supplemented with information about from which tuples in the database (*internal* lineage) or from which outside sources like programs or devices (*external* lineage) they were derived. Lineage can be defined as a function λ over all tuples t_i in the database where $\lambda(t_i)$ denotes the set of tuples from which t_i was derived. Tuples with empty lineage are called *base tuples*¹ and relations of such tuples are called *base relations*.

In Figure 2.2, the only base relation holds information about movies corresponding to Figure 2.1. The view is composed of studio names from the movies relation, i.e. names

¹Base tuples may have external lineage, but we only consider internal lineage in our work.

movies (m)		
t_{m_1}	Batman	1966
t_{m_2}	Catman	1969
		Twentieth Century Fox
		Twentieth Century Fox

view (v)		
t_{v_1}	Twentieth Century Fox	$\lambda(t_{v_1}) = \{t_{m_1}\}$
t_{v_2}	Twentieth Century Fox	$\lambda(t_{v_2}) = \{t_{m_2}\}$

Figure 2.2.: A lineage database (LDB). Tuples t_{v_1} and t_{v_2} have lineage information attached to them stating that they were derived from tuples t_{m_1} and t_{m_2} respectively.

of studios that have already produced a movie. Empty lineage is omitted and hence only lineage for the view is illustrated.

The lineage feature can be very useful, when it becomes important *why* certain data are in the database. Queries in relational databases, for example, often contain double entries, just like the illustrated view. But in contrast to common relational databases, in lineage databases it is possible to find out why there are double entries based on their lineage.

Though not illustrated by this simple example, the lineage of a tuple does not necessarily consist only of base tuples. But this is not a problem, because a lineage set can always be traced to a set of base tuples by recursively replacing derived tuples in the lineage set by their own lineage, until only base tuples remain.

2.1.3. Uncertainty-lineage databases

An uncertainty-lineage database (ULDB) can be understood as a probabilistic database where each possible world is a lineage database. Hence, lineage in a ULDB is not defined over all tuples, but over all tuple *alternatives*. A tuple alternative can only be present in a possible world, if the alternatives referred to by its lineage are present as well.

In our work, we only consider internal lineage and furthermore presume that it is *well-behaved*. Well-behaved lineage is always acyclic, deterministic and uniform and thus its semantics are quite intuitive. For example, there can be no alternatives that are derived from each other and alternatives of the same tuple cannot have equal lineage.

A more formal view on well-behaved ULDBs is provided in [BSH⁺08].

Figure 2.3 illustrates a view that is again composed of studio names from the already known studios relation with their respective numbers of produced movies derived from the movies relation (see Figure 2.1). However, in contrast to the corresponding probabilistic view without lineage, the confidence values have not been computed so far. By using lineage in probabilistic databases, the computation of data and their confidences become independent, so that in ULDBs *confidence values can be computed on-demand* based on the respective alternatives' lineage.

studios (s)		
	name	c
t_{s_1}	Republic Pictures	1
t_{s_2}	Twentieth Century Fox	1
t_{s_3}	Zulu Pictures	-

movies (m)				
	title	year	studio	c
t_{m_1}	Batman	1966	Twentieth Century Fox	0.8
		1967	Republic Pictures	0.2
t_{m_2}	Catwoman	1969	Twentieth Century Fox	0.6

view (v)			
	studio	movies	c
t_{v_1}	Republic Pictures	0	-
		1	-
t_{v_2}	Twentieth Century Fox	0	-
		1	-
		1	-
		2	-
t_{v_3}	Zulu Pictures	0	-

$\lambda(t_{v_{1,1}}) = \{t_{s_{1,1}}, t_{m_{1,1}}\}$
$\lambda(t_{v_{1,2}}) = \{t_{s_{1,1}}, t_{m_{1,2}}\}$
$\lambda(t_{v_{2,1}}) = \{t_{s_{2,1}}, t_{m_{1,2}}, t_{m_{2,?}}\}$
$\lambda(t_{v_{2,2}}) = \{t_{s_{2,1}}, t_{m_{1,1}}, t_{m_{2,?}}\}$
$\lambda(t_{v_{2,3}}) = \{t_{s_{2,1}}, t_{m_{1,2}}, t_{m_{2,1}}\}$
$\lambda(t_{v_{2,4}}) = \{t_{s_{2,1}}, t_{m_{1,1}}, t_{m_{2,1}}\}$
$\lambda(t_{v_{3,1}}) = \{t_{s_{3,1}}\}$

Figure 2.3.: An uncertainty-lineage database (ULDB) combines the x-tuple concept with lineage on alternative level.

Furthermore, lineage enables the ULDB to perform *coexistence checks* of alternatives, i.e. to determine whether several given alternatives can exist in the same possible world or whether they cannot. This is possible by simply checking the different lineages for contradictions. For example, checking the lineage sets of the alternatives $t_{v_{1,2}}$ and $t_{v_{2,4}}$ for contradictions reveals that those alternatives are mutually exclusive, as they are derived from the mutually exclusive alternatives $t_{m_{1,1}}$ and $t_{m_{1,2}}$.

Data lineage can also help defining meaningful confidence values for tuple alternatives that originated from different data sources. If some of the sources are deemed more reliable than others, alternatives from those sources can be assigned a greater confidence value.

Constructing possible worlds in a ULDB

In a possible world (also called possible-instance), every tuple is represented by exactly one alternative. Note that tuple t_i is represented by alternative $t_{i,?}$, when it is missing, i.e. not present in a possible world. Base alternatives cannot be derived and hence the correct base alternatives have to be specified in each possible world, while lineage makes implicitly clear which of the derived alternatives are the correct ones – depending on the base tuples. So, a possible world in a ULDB can be constructed by choosing exactly one alternative for every base tuple.

Since a possible world is determined entirely by the base tuples, the confidence of a possible world can be computed from the base tuples alone: It is the product of the confidence

values of all alternatives of which the possible world is composed. Note that this includes non-existence alternatives for tuples that are not present.

Trio – an uncertainty-lineage database

Trio DBMS is the only implementation of the ULDB data model that we are aware of. It has its own query language called TriQL² which is an enhancement of SQL that supports probabilistic and lineage features by some new semantics and constructs.

Programs can work on a Trio database via an extended Python DB 2.0 API and users can perform administrative tasks via a command-line or a graphical user interface.

Trio works on a common relational database and translates most TriQL queries or data modification commands to SQL queries or commands which are then executed on the underlying database. There are also some functionalities implemented by stored procedures for efficiency or other reasons.

To be precise, Trio's datamodel is not the standard ULDB data model, but the ULDB^v data model, which is discussed in [STW08]. The ULDB^v extends the ULDB data model by a *versioning system*: When performing data modification commands (insert, update, delete) in Trio, old data are never deleted, but instead remain in the database as an older version, while new data are stored as new versions.

2.2. Data integration

Integration of IT systems can be divided into two classes: application integration and information integration. While application integration is a matter of combining IT processes, information integration describes the process of merging several information sources into a single one. The more specific term *data integration* is often used for information integration applied to structured data ([MM08]). We only consider integration of structured data in our work.

Integrating databases means to facilitate uniform access to homogeneously structured databases by either a virtual or a materialised data integration system. *Virtual* data integration means that queries are transformed and then forwarded to the source databases, whereas *materialised* data integration implies that the integrated data actually reside in the integrated database and that queries are executed without accessing the source databases ([CGL01]). Querying the integrated information system is likely to yield a better result than querying only one of the source databases and, of course, it is less complicated for the user than querying all source databases. So, one of the great advantages of using an integrated information system is that all existing data can be accessed through *only one interface*. Data warehouses which have been used successfully in enterprises for many years are a classic example of (materialized) integrated information systems.

²pronunciation: 'tri:kl

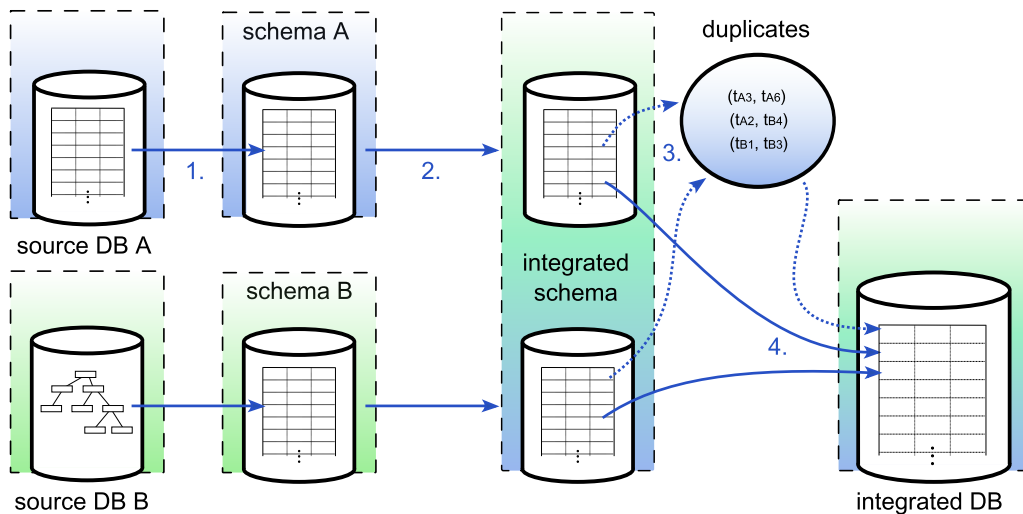


Figure 2.4.: The data integration process can be divided into four steps: the normalisation of data models (1.), schema management (2.), duplicate detection (3.) and data fusion (4.).

In this section, we describe the different steps of the data integration process according to [LN06], where Ulf Leser and Felix Naumann discuss many problems which can occur during the data integration process and present adequate approaches to solve them.

An overview of the steps in the data integration process is given in Figure 2.4: First, the different data sources are transformed into compatible data models if necessary. Then heterogeneity arising from differences between the source databases' schemata and the integrated database's schema are resolved, so that all data are available in a common data model and schema. The next step, the *duplicate detection*, is to identify tuples referring to the same real-world entity in order to merge them in the final data fusion step.

2.2.1. Normalisation of data models

When integrating databases, the first problem that has to be sorted out is the heterogeneity of the sources on data model level. If the source databases differ in their data models, e.g. some are relational and some are XML databases, a conversion of data from one data model into another becomes necessary.

2.2.2. Schema management

Schema management subsumes all methods to handle structural (and to some extent even semantic) heterogeneity on schema level and is based on meta data, i.e. the schemata of the data sources and the integrated system. Resolving schema heterogeneity tends to be a very complicated task best performed by domain experts, because this task requires the ability to understand and interpret the data.

Schema management is often performed in two steps: mapping creation and schema mapping.

Mappings are attribute correspondencies between the schemata of the data sources and the destination schema and thus reflect the semantic relationship between source and integrated schemata. They can be created manually, but for large databases it is often more convenient to have them created automatically by *schema matching* techniques ([RB01]). In order to guarantee semantic correctness, all results of schema matching have to be verified by a domain expert.

In the second step, which is called *schema mapping*, data transformation rules from the source database schemata to the integrated database schema are derived from the mappings. Some schema mapping formalisms and a good overview of related work are given in [FHH⁺09].

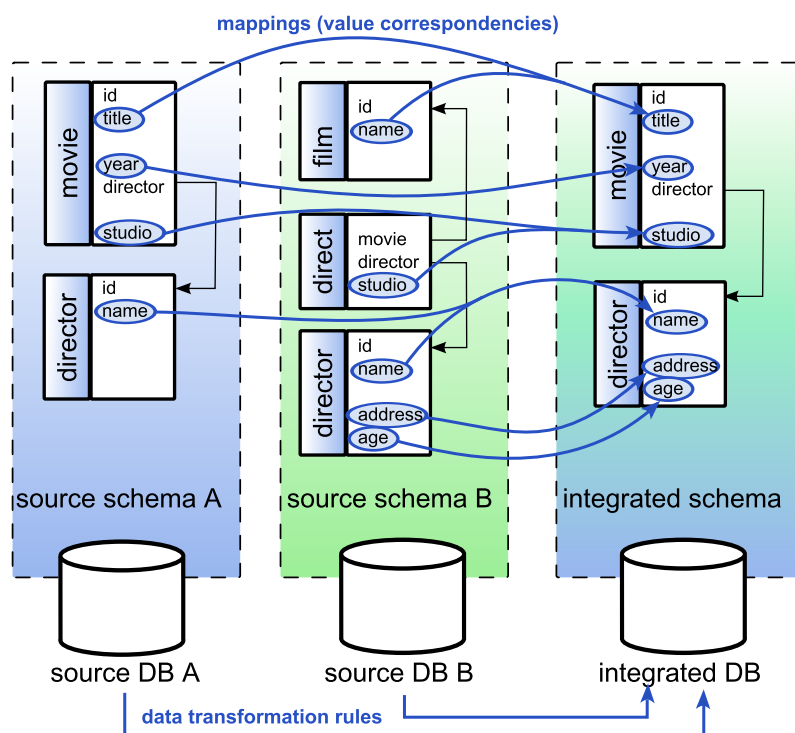


Figure 2.5.: Schema management. First, mappings are created and, secondly, transformation rules are derived from the mappings.

Schema management is illustrated in Figure 2.5 with the example of two movie databases A and B that are to be integrated into one single database.

2.2.3. Duplicate detection

Once the sorts of heterogeneity described above have been resolved, the source data are available in a global context, i.e. they are consistent w.r.t. data model and schema. But still, data of the different sources may overlap and so some real-world entities can have multiple – possibly contradictory – representations in the integrated data, which are called

duplicates. As implied by the name, *duplicate detection*³ refers to the detection of such duplicates for their later fusion and is an essential step in the data integration process to keep the data free from contradictions.

The research of this topic goes back to 1969, when Fellegi and Sunter first published a theoretical framework for duplicate detection in [FS69] that was based on the work [NKAJ59] from 1959 by Newcombe. The basic idea of duplicate detection is to compare all tuples, mostly in pairs, and to decide whether they are treated as matches (duplicates) or not. In order to increase effectiveness and efficiency, duplicate detection comprises four phases according to [HS98]: data preparation, search space reduction, in-depth comparison and building the transitive closure of the declared matches.

During the *data preparation*, rather obvious errors are detected and – if possible – removed in order to increase the effectiveness of the duplicate detection. For example, the data can be represented in different formats (“The Dark Knight” vs. “Dark Knight, The”) or units (€ vs. £) or they can be plainly incorrect (e.g. because of typos) or inconsistent. An example of inconsistencies is a director who is younger than a movie he or she has directed. Some typical data preparation techniques are *normalisation* to convert values in a standardised format or unit, *finding missing values or outliers* and *reference tables* which help to find and resolve inconsistencies. A good overview of these techniques is given in [MF03], but since data preparation is also particularly interesting for data mining, really detailed descriptions can also be found in data mining books like [DJ03].

The *search space reduction* is supposed to reduce the number of thorough and hence expensive tuple comparisons by filtering out highly dissimilar tuple pairs as unmatches with cheap comparison techniques. While pairwise comparison of *all tuples* seems merely inefficient for small databases, it is downright *infeasible* for large databases. As the number of duplicates in the data is usually very small in comparison to the overall number of tuples, search space reduction can strongly reduce the number of comparisons and thus increase the efficiency of the duplicate detection considerably.

In Section 2.3, we go into detail about search space reduction.

In the following passage, we give a short overview of the in-depth comparison according to [EIV07] and [PvKdKR09]. Each of the remaining tuple pairs is assigned to either the set of matching tuples M or the set of unmatching tuples U depending on the result of the *in-depth comparison* of the involved tuples⁴.

The similarity of two tuples is determined based on the similarity of their attribute values. This is commonly referred to as *field matching* or *attribute value matching*. There is a great variety of distance measures that are quite commonly used here, such as character-based or phonetic similarity metrics, all of which are more or less appropriate depending on the data.

³Other commonly used names for duplicate detection are *record linkage*, *merge-purge problem* and *entity resolution*.

⁴In semi-automatic approaches, tuple pairs can also be assigned to the set of possibly matching tuples P for later examination by domain experts.

Whether a tuple pair is assigned to M or U depends on the applied *decision model*, which might for example be probabilistic or rule-based. The probabilistic matching model was first mentioned by Newcombe et al. in 1959 ([NKAJ59]) and formalised by Fellegi and Sunter in 1969 ([FS69]). The first proposal of a rule-based approach to duplicate detection that we are aware of was made by Wang and Madnick in 1989 ([WM89]).

According to [HS98], the set of declared matches (declared duplicates) is extended to its transitive closure in the final phase of the duplicate detection. Imagine three tuples t_a , t_b and t_c where the set of declared matches (after the in-depth comparison) contains (t_a, t_b) and (t_b, t_c) , but not the tuple pair (t_a, t_c) . Either (t_a, t_c) should be in the set of declared matches or (t_a, t_b) or (t_b, t_c) should not. So extending the set of declared matches to its transitive closure simply means resolving conflicts by inferring the transitivity of equality.

Although the topic of duplicate detection has been researched for many years, apparently there are still no standardised large-scale benchmarking data and thus its hardly possible to draw a meaningful comparison between different duplicate detection techniques.

2.2.4. Data fusion

Data fusion is the last step of data integration as we describe it and refers to the process of merging all duplicates into single tuples and thus resolving the remaining conflicts. According to [BN08], automatic data fusion is still in its infancy and hence commercial data integration systems often fuse data by mediating strategies, i.e. computing average values, or by deciding strategies, i.e. simply choosing one of the existing tuples, and filling missing values with data from other tuples corresponding to the same real-world entity. Due to the lack of adequate automatic solutions, human interaction can be required to achieve a satisfying result and this makes data fusion a potentially expensive and time-consuming task.

In recent years, ideas about how to save time and reduce costs for data fusion have been published. For example in [dKvK07], van Keulen et al. suggest to finish data integration without fusing the duplicates, but instead storing the integrated (partially conflicting) data into a probabilistic database and thus delaying the conflict resolution, e.g. in order to leave it to application users.

2.3. Search space reduction in certain data

This section is devoted to search space reduction for duplicate detection in the process of certain data integration.

We begin with a motivation of search space reduction for duplicate detection and examine metrics that can be used for evaluation of search space reduction techniques. After that, we survey the two techniques Blocking and the Sorted Neighborhood Method in detail and, finally, give an overview of other existing approaches.

As described in Subsection 2.2.3, a key issue of data integration is the duplicate detection, i.e. identifying all tuples corresponding to the same real-world entities, in most cases by pairwise comparisons of all tuples with each other. The point of search space reduction in particular is to improve the efficiency of duplicate detection by detecting obvious unmatches cheaply and thus reducing the number of expensive in-depth comparisons⁵. Finding adequate and inexpensive comparison methods, however, requires knowledge about the data to be compared and hence search space reduction is usually done semi-automatically. Nonetheless, fully automated search space reduction has already been object of research ([Bil06]).

The first search space reduction techniques were invented over 40 years ago and divided the set of all tuples into so-called partitions or blocks and all subsequent in-depth tuple comparisons were only performed between tuples of the same partition or block respectively. For this reason, search space reduction is also often referred to as *partitioning* or *blocking*. We do not use the term blocking in this context, though, because we find the ambiguity of the word *blocking* very confusing: Blocking is not only used as a common term for the search space reduction technique described in subsection 2.3.2, but also often used for search space reduction in general. We only use the term Blocking for the search space reduction technique.

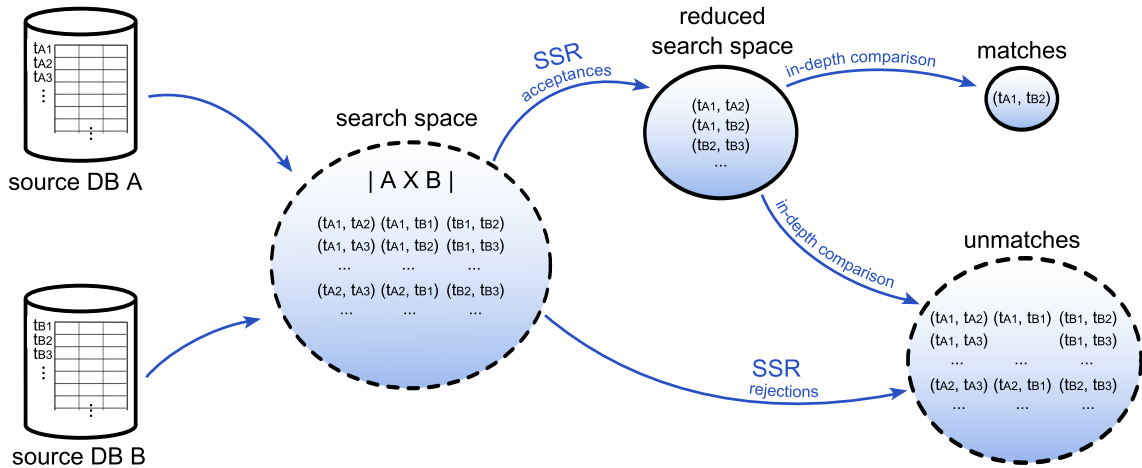


Figure 2.6.: Search space reduction during duplicate detection. The dashed boundaries of the initial search space and of the unmatches indicate that the sets are never materialised, since they are of no use for the duplicate detection process.

While duplicate detection basically means to assign each tuple pair either to the set of matching tuples M or unmatching tuples U , in terms of search space reduction, tuple pairs are either assigned to U or not assigned at all. So all tuple pairs remaining in the reduced search space are no obvious unmatches and hence might (or might not) be matches.

⁵Another approach to improving the efficiency of duplicate detection we do not cover in our work is to reduce the costs of the tuple comparisons and not only to reduce their number.

The (final) assignment of tuple pairs to *either M or U* is carried out during the in-depth comparison. This is illustrated in Figure 2.6.

Search space reduction is based on the observation that in many cases highly similar tuples make up only a very small portion of a data set ([Bil06]) and hence most of the in-depth comparisons are unnecessary, because obvious dissimilarities can also be exposed by cheaper means. Since the number of tuple pairs grows quadratically with the size of the data set, search space reduction becomes a matter not only of performance, but of feasibility for integration of larger sets of data.

Consider two data sources A and B with 100,000 tuples each. Basically, there are $(100,000 + 100,000)^2 = 4 \cdot 10^{10}$ tuple pairs, but most of them are still unnecessary: Taking into account that we do not have to compare reflexive tuple pairs, i.e. tuple pairs in the form of (a,a) , we can reduce this number by 200,000. And still, the number of remaining tuple pairs is twice as large as the number of necessary tuple comparisons, because every comparison of two tuples a and b has two symmetric pair representations in the search space, namely (a,b) and (b,a) . So there actually are $\frac{(100,000+100,000)^2 - 2 \cdot 100,000}{2} \approx 2 \cdot 10^{10}$ tuple pairs to consider for duplicate detection. If A and B are known to be duplicate-free already, this number is reduced by approximately one half to $100,000 \cdot 100,000 = 10^{10}$, but still astronomously huge.

In [HS95], Hernández and Stolfo address the problem of disk-resident databases often being much larger than the main-memory, which causes the common search space reduction methods to be very slow due to the I/O bottleneck, and present a possibility to perform search space reduction completely in the main-memory, independently of the size of the data set: Their approach is called the *Clustering Method* and mainly consists of, first, clustering the data, i.e. mapping them into an n-dimensional cluster space by an n-attribute key, and secondly, perform search space reduction on each cluster and combine the resulting sets of tuple pairs⁶.

But of course, this approach can only be beneficial, if the resulting clusters can be loaded completely into the main-memory and, furthermore, if tuples representing the same real-world entity are assigned to the same cluster.

2.3.1. Evaluation metrics

In general, search space reduction is based on cheap comparisons and hence is known to cause two kinds of errors: *false acceptance*, i.e. leaving an actual unmatched in the search space, and – even worse – *false rejection*, i.e. removing an actual match from the search space by assigning it to U. False rejection is worse than false acceptance, because an actual match that is rejected is not considered again (unless recovered by the transitive closure) and therefore may change the result for the worse, whereas a false acceptance is eventually

⁶As a matter of fact, the authors propose using the Clustering Method not only for search space reduction in particular, but for duplicate detection in general.

corrected during the in-depth comparison and hence has no impact on the quality of the duplicate detection result at all.

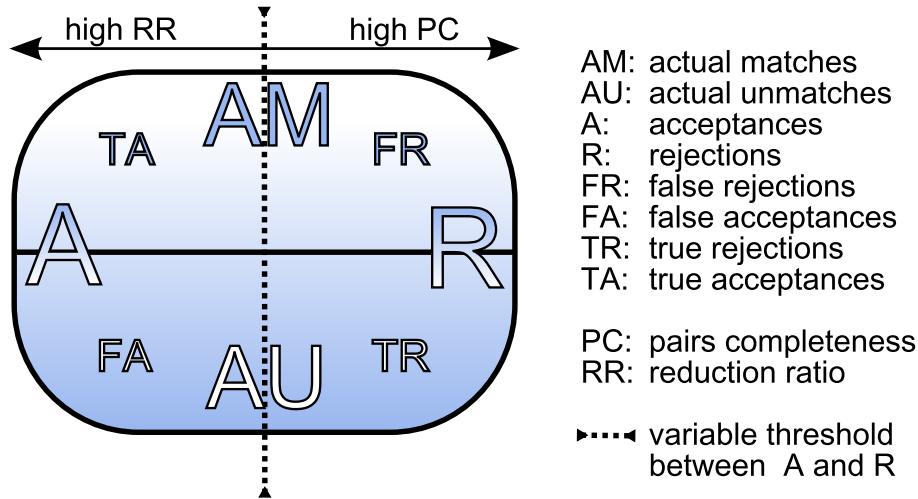


Figure 2.7.: The trade-off between PC and PR. Shifting the similarity threshold to the left increases the precision, but decreases the pairs completeness. Shifting it to the right has the opposite effect.

Two metrics used to measure quality and effectiveness are *pairs completeness* (PC), also referred to as *recall*, and *reduction ratio* (RR) ([LF05]). The pairs completeness represents the share of correctly accepted tuple pairs in the actual matches and the reduction ratio indicates by which factor the search space was reduced. In simple words: High pairs completeness indicates high quality of the result and high reduction ratio indicates high effectiveness of the search space reduction.

Of course, it is desirable to maximise both PC and RR , but usually these are conflicting goals, as it is illustrated in Figure 2.7: The entire search space consists of actual matches (AM) and actual unmatches (AU). Search space reduction means to assign each tuple pair to either the set of acceptances (A) or to the set of rejections (R) on the basis of some similarity threshold. Figuratively spoken, it means to draw a line (in this case a dashed one) between the tuple pairs that are accepted as possible duplicates and the tuple pairs that are rejected. Since errors occur during this process, the set of acceptances (A) consists of two subsets, namely the set of true acceptances (TA), i.e. correctly identified matches, and the set of false acceptances (FA), i.e. tuple pairs that are in fact unmatches. Similarly, the set of rejections (R) consists of true rejections (TR) and false rejections (FR).

Back to the trade-off between pairs completeness and reduction ratio: Loosening the rejection criteria (moving the dashed line to the left) and thus rejecting more tuples tends to *increase reduction ratio*, since less tuple pairs are accepted as possible matches. But it also tends to *decrease pairs completeness*, because actual matches are rejected more easily. Making the rejection criteria more strict has the opposite effect.

We want to introduce two more measures, the first of which is the *precision* (PR) ([BS06]). The precision is the share of true acceptances (TA) in the accepted tuples pairs (A) and it

is far more helpful than the reduction ratio, when it comes to comparing different SSR techniques by their results: Consider two duplicate-free data sources with 100,000 tuples each. As mentioned before, the entire search space comprises 10^{10} tuple pairs. Assume we wanted to compare a perfect SSR technique that yielded $PC = 1$ and $PR = 1$ to another SSR technique that achieved a result with $PC = 1$ and $PR = 0.001$. If there were 10,000 duplicate tuples, the better of the two techniques would reduce the search space to 10,000 tuple pairs, while the worse technique would only reduce it to $\frac{10,000}{0.001} = 10,000,000$. As a consequence, using the better technique would result in a reduction ratio of $RR \approx 1 - \frac{10,000}{10^{10}} = 0,999999$ and applying the worse technique in a reduction ratio of $RR \approx 1 - \frac{10,000,000}{10^{10}} = 0,999$. The problem simply is that $RR \approx 1$ in almost every case anyway.

The other measure we want to introduce is the f_β -score (F_β) according to [vR79], which combines two metrics in one single value. The basic f_β -score is computed from pairs completeness (recall) and precision, but it can also be computed from pairs completeness and reduction ratio as we define it below. Since two measures are combined in it, a trade-off between those two measures has to be modeled into the f_β -score, i.e. a decision as to how much more important one measure is than the other. For example, a high pairs completeness is – usually – much more important for a search space reduction technique than a high reduction ratio. So increasing the pairs completeness while decreasing the reduction ratio is often acceptable and thus should improve the f_β -score in many cases. However, to which extent it is acceptable to reduce the search space at the expense of the pairs completeness depends on many factors, for example the data you are working with or the available computing power. This trade-off has to be made in every scenario. The pairs completeness in the f_β -score is weighted by factor β : For $\beta^2 = 1$ PC and RR are equally weighted, while PC has a higher weight for $\beta^2 > 1$ and a smaller weight for $\beta^2 < 1$. Combining two metrics in one value makes comparing different SSR techniques much simpler, because only one value has to be considered. The f_β -score could for example be helpful when it comes to optimising effectiveness and quality of an SSR technique automatically.

We define the above metrics as real numbers in the range of $[0,1]$, where 0 is the worst and 1 is the best value, as follows:

$$\begin{aligned} \text{pairs completeness (PC)} &= \frac{|\text{true acceptances}|}{|\text{actual matches}|} \\ \text{reduction ratio (RR)} &= 1 - \frac{|\text{acceptances}|}{|\text{all tuple pairs}|} \\ \text{precision (PR)} &= \frac{|\text{true acceptances}|}{|\text{acceptances}|} \\ \text{f}_\beta\text{-score (F}_\beta\text{)} &= \frac{(\beta^2 + 1) \cdot \text{PC} \cdot \text{RR}}{\text{PC} + \beta^2 \cdot \text{RR}}. \end{aligned}$$

2.3.2. Blocking

The idea of blocking can be traced back to [NK62] from 1962, where Newcombe proposed reducing the search space by only considering tuple⁷ pairs that agree on a characteristic such as surname or date of birth. Blocking has been referred to as *the* standard or traditional search space reduction technique ever since and has been used in many practical applications ([Jar89], [BCC03], [EIV07], [Chr08b], [dVKCC09]).

Blocking means to separate a single large set of tuples into smaller mutually exclusive sets called blocks and to restrict further comparisons to tuples within the same block ([New67]). As described in [EIV07], first, a *block key value* is computed for each tuple from a single attribute value or a combination of several attribute values. Often the block key value is a simple concatenation of attribute values or parts of attribute values, but in many cases it has proven beneficial to process the attribute values before building the key value to make it more robust against small data errors, for example by applying phonetic encodings to the key attribute values.

Tuples that agree in the block key are finally assigned to the same block and considered candidate duplicates. Blocking roughly works like hashing, where the key attribute values correspond to the hash input and the block key value corresponds to the hash code.

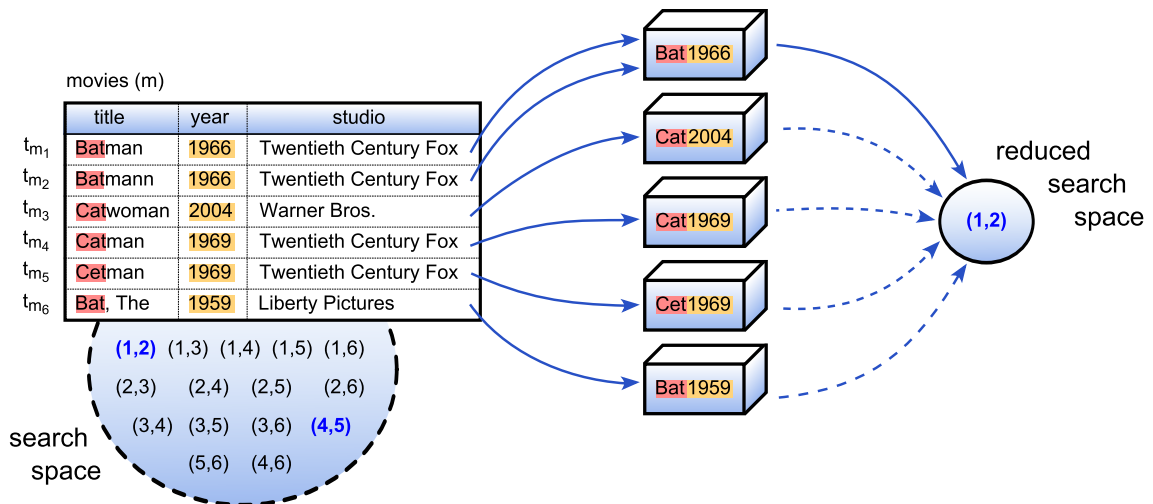


Figure 2.8.: In Blocking, a key value is computed for each tuple and all tuples with identical key values are paired. In this example, not all actual matches (bold tuple pairs) are accepted, because some key attribute values are erroneous.

An illustration of the Blocking technique is shown in Figure 2.8 where Blocking is used for duplicate detection in a movie relation. In this example, a block key value is the concatenate of the first three characters of a movie's title and its production year. The initial search space consists of ten tuple pairs, but most of them are quite obviously no duplicates and hence unnecessary to compare in detail. The only two actual matches in this example are (1,2) and (4,5).

⁷Since our work is based on relational databases, we use the term tuple known from relational databases instead of the more general term record as the author does.

As the blocks only contain very few tuples each, only few tuple pairs for later comparison remain in the search space. But unfortunately, not only highly dissimilar tuple pairs are left out. Typos or other differences in block key attribute values which may result in false rejections are a major problem with Blocking: Although tuple 2 is erroneous in the title attribute value, tuples 1 and 2 agree in the block key and the tuple pair (1,2) remains in the search space, because the part of the title that is relevant for the block key value is not affected. Tuples 4 and 5, however, are not assigned to the same block, although they are almost identical, because the block key value of tuple 5 is affected by a typo in the title. In this example, we yield a pairs completeness of $PC = \frac{1}{2} = 0.5$, a reduction ratio of $RR = 1 - \frac{1}{15} = 0.9\bar{3}$ and a precision of $PR = \frac{1}{1} = 1$.

Choosing the block key always means a trade-off between pairs completeness and reduction ratio (or precision). A key by which the tuples are sorted into large blocks makes Blocking very ineffective since the search space is hardly reduced, but on the other hand, a key that causes the tuples to be assigned to too small blocks is likely to cause more false rejections. However, key attributes should not only be chosen by their discriminating power, but also by their error rate; using key attributes which are likely to be erroneous is also likely to result in tuples being sorted into wrong blocks. Common ways to cope with false rejections due to erroneous attribute values are to prefer credible key attributes or to run multiple Blocking passes with different keys and combine the resulting tuple pairs.

2.3.3. Sorted Neighborhood Method

The first detailed description of the Sorted Neighborhood Method we are aware of was published by Hernández and Stolfo in [HS95] and [HS98]. This subsection is based on those sources.

Consider a data set in which duplicate tuples are to be detected. In order to reduce the set of candidate duplicate pairs, the Sorted Neighborhood Method is performed in three phases: First, for each tuple a key value is computed (similarly to the block key value in Blocking). Secondly, the tuples in the data set are sorted – usually lexicographically – by their respective keys into a list and, finally, all tuples in the list within a certain distance are paired with each other and the resulting pairs remain in the reduced search space.

The underlying assumption is that duplicate tuples have similar key values and hence are neighbouring tuples in the sorted list resulting from step two. As a consequence, keys should be designed in such a way that duplicate tuples do have similar key values. When creating keys by concatenating (portions of) tuple attributes, it is not only critical to consider their discriminating power and reliability, but also that the attributes determining the first parts of the key have the highest impact on where the tuples are inserted into the sorted list. Hence, error-prone key attributes can still be facilitated for the key’s tail and do not have to be neglected for key design.

Figure 2.9 illustrates the three phases of the Sorted Neighborhood Method on the data known from the illustration of Blocking in Figure 2.8. The key values built in phase one

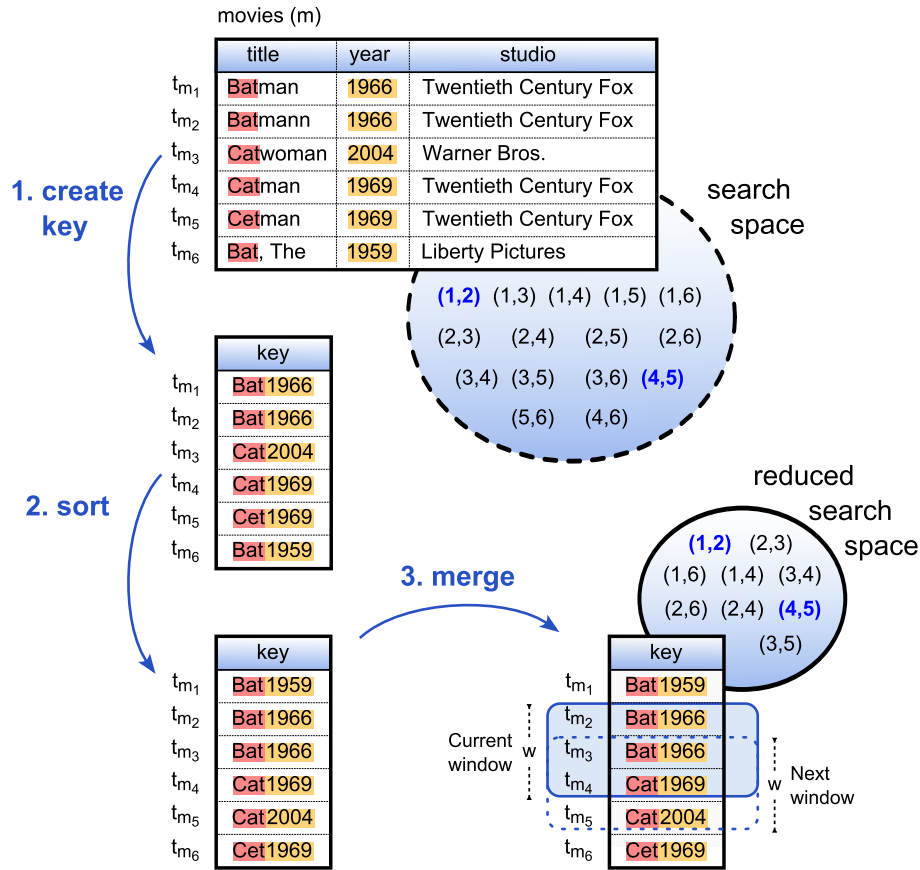


Figure 2.9.: The Sorted Neighborhood Method is more error tolerant than Blocking in most cases, but also results in a smaller reduction ratio.

are again designed as a concatenation of the first three characters of the movie title and the movie's production year. After a key value has been computed for each tuple, a sorted list of all tuples ordered by their keys is created⁸. Phase three can be visualised as a window of fixed size $w = 3$ that is moved sequentially over the sorted list, where all tuples appearing within the window at the same time are paired with each other. At the beginning, the first w tuples are paired and then the window is moved one entry down, so that the first tuple slides out of the window and the next tuple slides into the window. Note that in each step, the window corresponds to a block of tuples in Blocking, only that the blocks here overlap. Obviously, the pairs completeness $PC = \frac{2}{2} = 1$ is perfect, but reduction ratio $RR = 1 - \frac{9}{15} = 0.4$ and precision $PR = \frac{2}{9} = 0.2$ are not as good as in our Blocking example.

The trade-off between pairs completeness and reduction ratio lies not only in the key design, but in the window size w . The window size is $w = 2$ at least, meaning only direct neighbours in the sorted list are paired, and can naturally not be larger than the number of all tuples, where the search space is not reduced and all tuples are paired with each other. Increasing w tends to increase the pairs completeness, but also results in a smaller reduction ratio, since more tuples are paired. Decreasing w improves the reduction ratio,

⁸In order to guarantee a deterministic tuple order, tuples with equal key values in the sorted list are ordered by their internal ID.

but can lead to more false rejections; if there are more than w tuples with the same key value, not even all tuples with the same key value will get paired, although they are rather likely duplicate candidates. In our example, in spite of the key values of tuples 1 and 2 being the same, it depends on the order of those two tuples in the sorted list⁹ whether tuple 1 or tuple 2 is paired with tuple 3.

According to [HS98], simply increasing the window size is not a very effective solution to this problem, because the number of false acceptances grows very fast with the window size. Hernández and Stolfo recommend to apply the transitive closure to the result of the in-depth comparison. This, however, might result in a low reduction ratio when using an already large window.

To compensate for the shortcomings of the basic Sorted Neighborhood Method, many modifications and extensions have been developed, some of which we describe in the following.

Multi-pass SNM

As already mentioned, tuples that correspond to the same real-world entity can differ in some attribute values and thus might differ in their key values as well, depending on how the key is built, i.e. which part of the key value is computed from which attribute values. To make up for major differences, deciding for one key design and just increasing the window size is not an adequate option. An easier and more effective approach is to run several independent passes with different keys and relatively small windows and to combine all resulting tuple pairs of the different passes. When all passes have a high precision (and thus a relatively high reduction ratio), the combined result can have a high pairs completeness, although the pairs completeness was small in every single pass. According to Hernández and Stolfo, running only few passes with small window sizes and computing the transitive closure of the result of the in-depth comparison yields much better pairs completeness and reduction ratio than a single pass with a large window.

Increasing the window size hardly affects the pairs completeness, but decreases the reduction ratio and hence is no promising option to improve the result. Given that small windows are used, however, the pairs completeness approaches 1 rather quickly with every additional pass, while the reduction ratio decreases only slowly. Still, since every additional pass requires sorting the entire set of tuples, the number of passes should be kept minimal as well to save computation time.

Adaptive SNM

A problem of the basic Sorted Neighborhood Method is that tuples with highly similar or even identical key values may not be paired with each other, if their number is large in comparison to the static window size. But while in the basic SNM a predefined number of

⁹Note that the tuple order depends on the sorting algorithm used in the second phase.

fixed-size overlapping blocks are created from the sorted list of tuples, the sorted tuple list in the *Adaptive SNM* approach presented in [YLyKG07] is divided into mutually exclusive blocks of variable size where, optimally, all corresponding duplicates are assigned to the same block.

Size and range of the blocks are determined dynamically through measuring the metrical distance¹⁰ between the key values of the first and the last tuple in the window and comparing it to a distance threshold ϕ . As long as the key distance does not exceed ϕ , the window is enlarged. But if it exceeds the threshold, the window is shrunk, until the threshold value is met again. Now a block is defined over the range of the window and the procedure is repeated, beginning at the first tuple in the list succeeding the block. In this approach, the window is not used to stencil blocks out of the list, but rather to shape them according to the data.

Let t_i denote the tuple on index i of the sorted list of n tuples and $dist(t_a, t_b)$ denote the key distance of tuples t_a and t_b . Roughly, this approach is not only based upon the assumption that similar tuples are near neighbours in the sorted list, but also that $dist(t_i, t_{i+1}) \leq dist(t_i, t_{i+2}) \leq \dots \leq dist(t_i, t_n)$ for $i < n$ and that $dist(t_i, t_j) \leq \phi < dist(t_i, t_k)$ holds when tuples t_i and t_j are in the same block and t_k is in a different one.

2.3.4. Other techniques

Even though Blocking and the Sorted Neighborhood Method seem to be the most prominent approaches in the related work, there are some other approaches to search space reduction that can even outperform them when configured properly ([BCC03]). We want to introduce the approaches that seem most promising to us in this subsection.

Q-gram Indexing

The Q-gram Indexing technique as described in [Chr08a] on page 35 is similar to Blocking, but much more flexible. The concept is to create more than one key for every tuple and to assign every tuple to *several* blocks. We illustrate this procedure by an example in Figure 2.10.

The first step is to create key values for the tuples, just like it is in normal Blocking. Afterwards, all q -grams of a key, i.e. all substrings of the length q , are saved to a list. A very common form of Q-gram Indexing used for search space reduction is Bigram Indexing ($q = 2$). In step three, the final key values are computed from the q -gram list and a threshold t in the range of 0 and 1, which is 0.8 in this example. The number of q -grams in a valid key is the threshold t multiplied by the length l of the q -gram list and hence $t \cdot l = 0.8 \cdot 6 = 4.8 \approx 5$ in our example. So the final key values in our example only contain 5 out of the initial 6 q -grams. Note that the order of the remaining q -grams in the final

¹⁰Note that not the distance of the tuples in the sorted list is meant, but a metrical distance between keys, for example the edit distance.

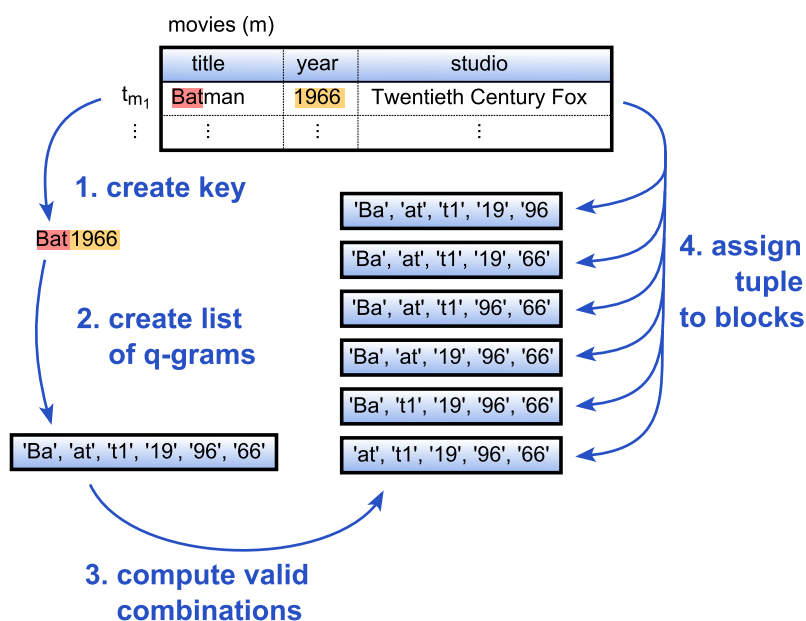


Figure 2.10.: *Q*-gram Indexing works similarly to Blocking, but can assign every tuple to *more than one* block.

key value is not changed. Finally, the *q*-gram combinations are concatenated to build the new key values by which the corresponding tuple is assigned to several blocks.

Clustering methods

In [ME97] Monge and Elkan describe the problem of search space reduction in terms of making out clusters, i.e. the connected components, of an undirected graph where every tuple is represented by a vertex and where vertices representing actually matching tuples are connected with an edge. The idea is that the duplicate relationship of tuples is transitive, i.e. that tuples *a* and *c* are inferred to be duplicates, if *a* and *b* as well as *b* and *c* are duplicates.

For the sake of efficiency, a union-find data structure is used to identify the connected components. The tuples are only compared to a representative of each cluster in order to minimise the number of tuple comparisons. This is assumed to be a major improvement to the efficiency with little negative effect on the result.

In [MNU00] McCallum et al. propose a clustering variant for search space reduction involving overlapping clusters, so-called *canopies*. All tuples are grouped in canopies with approximately similar tuples. The tuples in each canopy are then paired and the resulting sets of tuple pairs are unified to become the reduced search space.

3. Adapting certain data SSR techniques to probabilistic data

In the previous chapter we introduced data models extending the common relational data model by the x-tuple concept for modeling data uncertainty. We also described the process of duplicate detection in certain data and went into detail with the search space reduction techniques Blocking and the Sorted Neighborhood Method.

This chapter is devoted to the adaption of the above conventional techniques of search space reduction to the ULDB model.

The one big issue here is that uncertain attribute values may result in uncertain key values. So in order to make conventional search space reduction techniques applicable to probabilistic data, the uncertainty of the key values has to be resolved.

There are basically two approaches to those adaptations mentioned in [PvKdKR09] that we discuss in the following: generating only certain key values and thus *resolving uncertainty during the key generation*, or generating uncertain key values and so *considering the uncertainty during the remaining steps* of the individual search space reduction techniques.

3.1. Generating certain key values for search space reduction

A certain data tuple corresponds to an x-tuple with only one alternative with a confidence of 1. And since – in terms of x-tuples – every certain data tuple has only one alternative, there is only one valid key value per tuple.

Probabilistic tuples, however, can have several alternatives which can differ in the attribute values from which the key values are computed. As a consequence, generating certain key values from probabilistic tuples is much less straight-forward: For example, what key values should be computed for a movie tuple when it consists of several alternatives differing in the spelling of the title and the production year? But the advantage of using certain key values for search space reduction in probabilistic data is that only the key generation has to be modified and the rest of the search space reduction technique can be applied unchanged. In this section, we discuss three strategies of certain key generation and illustrate them for the Sorted Neighborhood Method.

3.1.1. Multi-pass over possible worlds

As described earlier, ULDBs can be seen as probability distributions over certain databases, so-called possible-instances or possible worlds. One conceptually simple way to perform search space reduction on a probabilistic database is to construct possible worlds from a ULDB (see 2.1.3), to apply perfectly conventional search space reduction techniques to each world individually and to combine the results.

Figure 3.1 illustrates the sorted key lists according to SNM in different possible worlds constructed from a very small ULDB with only three tuples. In spite of the small number of tuples and alternatives in this example, there are already quite many possible worlds. As the number of possible worlds in a ULDB in a realistic scenario with thousands or millions of tuples is usually tremendous, in practise, running passes on all possible worlds is infeasible and using *only a selection* of possible worlds seems the only option. But as illustrated, the most probable worlds are normally almost equal, since they differ in only few tuple alternatives. Furthermore, some tuples are not present in some possible worlds and thus cannot be paired with other tuples for later in-depth comparison. For example, tuple t_{m_2} is missing in possible world W_2 .

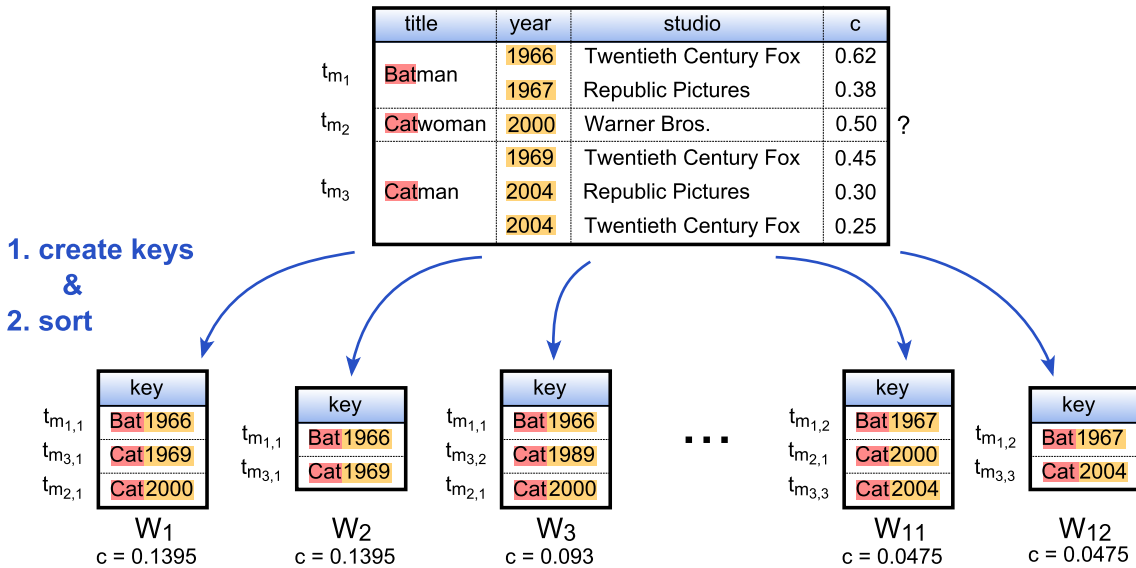


Figure 3.1.: The above x-relation represents twelve possible worlds. Performing SNM on different possible worlds yields different sorted key lists.

The decision which possible worlds should be used for subsequent passes is not easily made; once the first run has been performed on the most probable world, an additional second pass over the second most probable world will not improve the result much, because the most probable possible worlds are usually very similar. For a better result, worlds should be considered that have not only a rather high probability, but are also as dissimilar from one another as possible.

We propose performing at least the first run on the most probable world with *all tuples present*, in order to prevent missing tuples, even when the subsequent passes are executed

on possible worlds with missing tuples. In our implementations of this strategy only worlds without missing tuples are considered at all. Furthermore, we compute and save the key values for all alternatives once at the beginning of the procedure, so that they do not have to be computed again for subsequent passes¹.

We implemented two variants of this strategy. One is to construct the *most probable worlds* (W_1, W_2, \dots) and the other is to construct *highly dissimilar possible worlds*.

Building the most probable worlds

First, W_1 is built from the most probable tuple alternative of every tuple, while sorting the remaining tuple alternatives into a list by their confidence values as illustrated in Figure 3.2. The basic idea is to build the next most probable world W_n by choosing an alternative $t_{i,j}$ from the list of remaining tuple alternatives to replace the corresponding alternative $t_{i,k}$ from W_1 in such a way that the confidence $c(W_n)$ of W_n is maximal².

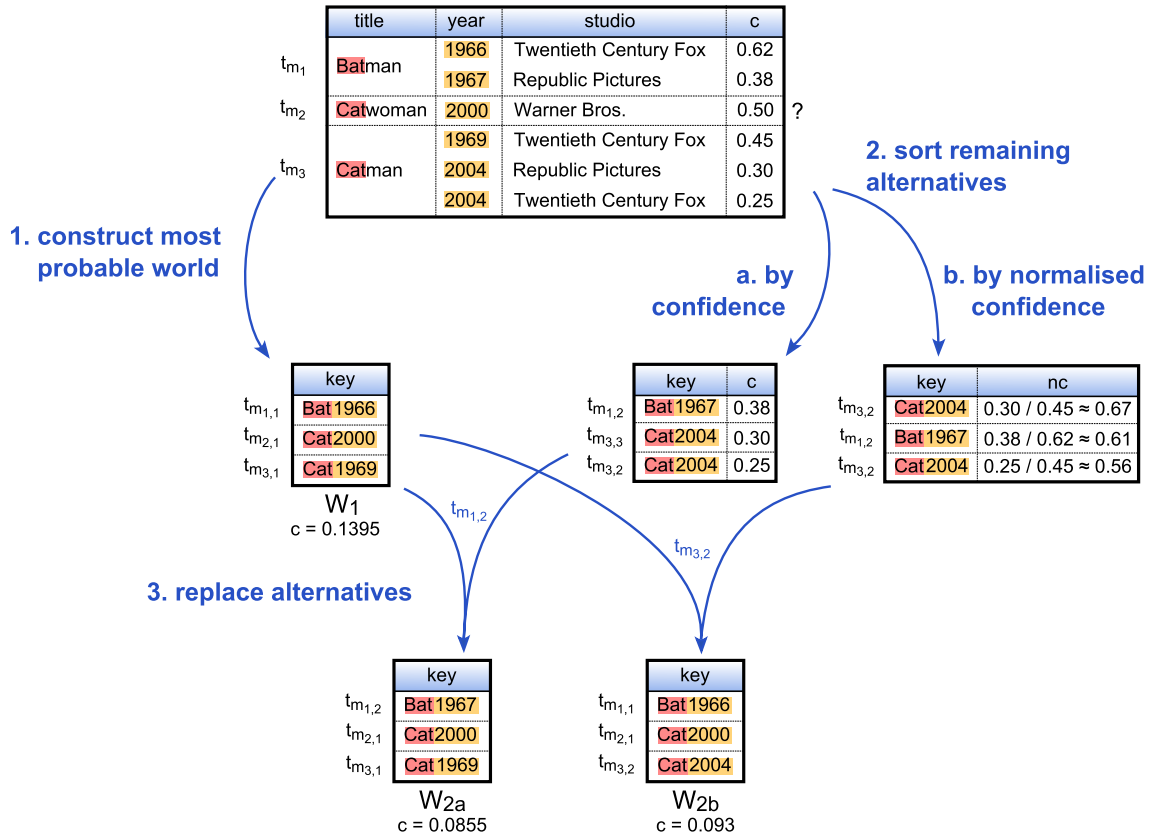


Figure 3.2.: In order to guarantee that the *most* probable worlds are built, the confidence values have to be normalised before sorting.

Choosing the most probable alternative from the list is quite simple, but the resulting probable world is not necessarily the next probable one. To guarantee that the next most

¹Since our other strategies are single-pass strategies, computing and saving key values before does not bring any advantages there.

²Note that $t_{i,j}$ and $t_{i,k}$ both represent the same tuple t_i .

probable world is generated, the confidence values of the alternatives in the list have to be normalised before sorting. Since the confidence of a possible world is the product of all contained tuple alternatives' confidences, $c(W_n) = c(W_1) \cdot \frac{c(t_{i,j})}{c(t_{i,k})}$ holds, where $\frac{c(t_{i,j})}{c(t_{i,k})}$ is the *normalised confidence* of $t_{i,j}$. In other words: $c(W_1)$ and $c(W_n)$ are the same product, differing only in one single factor which is $c(t_{i,k})$ in $c(W_1)$ and $c(t_{i,j})$ in $c(W_n)$. Hence, the confidence of W_n is maximal, where $\frac{c(t_{i,j})}{c(t_{i,k})}$ is maximal.

In our illustrating example in Figure 3.2, the choice of $t_{m_{1,2}}$ as the most probable of the remaining alternatives results in replacing alternative $t_{m_{1,1}}$. Logically, the resulting probable world is $\frac{c(t_{m_{1,2}})}{c(t_{m_{1,1}})} = \frac{0.38}{0.62} \approx 0.61$ times as probable as W_1 . But the normalised confidence is even greater for $t_{m_{3,2}}$ and the world that results from replacing $t_{m_{3,1}}$ by this alternative is $\frac{c(t_{m_{3,2}})}{c(t_{m_{3,1}})} = \frac{0.3}{0.45} \approx 0.67$ times as probable as W_1 .

If there is more than one alternative with a normalised confidence of 1, several equally probable and yet dissimilar possible worlds can be built by replacing tuple alternatives from the most probable world by the alternatives with a normalised confidence of 1 in any combination. So there are scenarios in which the most probable worlds actually are not very similar.

Building highly dissimilar possible worlds

Our second implementation of the possible worlds approach as illustrated in Figure 3.3 is to perform search space reduction on several possible worlds that are very dissimilar from each other. Here as well, the most probable world is constructed for the first pass. Afterwards, a possible world is built by using only the second most probable alternative of each tuple. If a tuple has only one single alternative (and hence no second most probable alternative), the most probable one is used. Accordingly, a possible world is then built from the third most probable tuple alternatives, and so forth. This procedure is repeated, until all alternatives have been used.

The number of worlds constructed by this procedure is rather small, as it cannot be greater than the maximum number of alternatives per tuple. Furthermore, each additional pass is likely to add many new tuple pairs and thus to improve the result much. So, this variant of the possible world strategy seems by far more promising than constructing the most probable worlds.

3.1.2. Key per tuple

Another strategy is to resolve the uncertainty on tuple-level by computing one certain key value for each tuple from its alternatives. Our idea is to first compute a single alternative for every tuple as a representative and to create a key value from this representative. To do so, meta data such as confidence values may be used as well as the actual attribute values. Of course, when computing a representative for key creation, only key attributes have to be considered. There are basically two strategies of computing a representative, which can

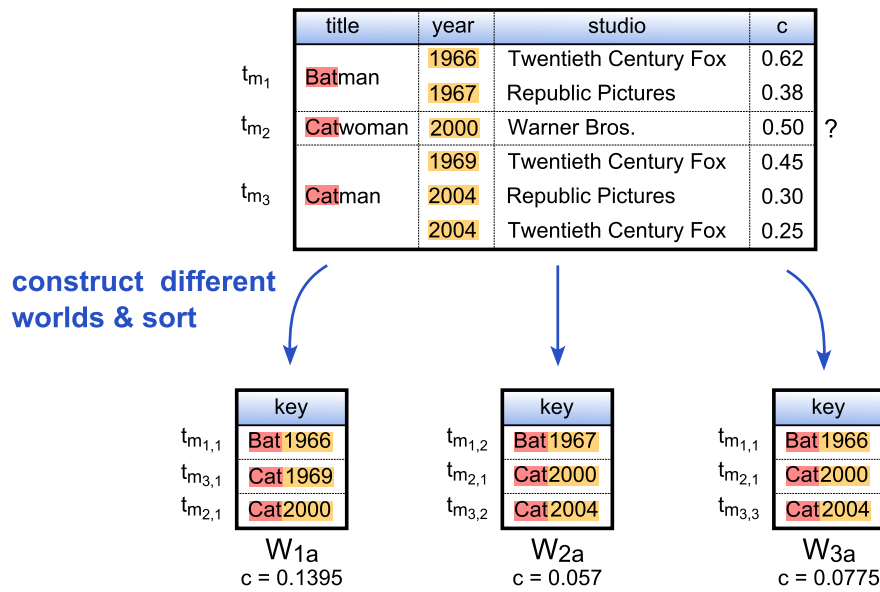


Figure 3.3.: When building highly dissimilar worlds, the confidence of the constructed worlds is of secondary importance.

be derived from the strategies discussed in [BN08]: *deciding* strategies, i.e. choosing one of the already existing alternatives, and *mediating* strategies, i.e. computing one that does not necessarily exist.

A very simple deciding strategy is to pick the most probable alternatives³ as illustrated in Figure 3.4. This is equivalent to search space reduction on just the most probable world with no missing tuples, because the resulting SNM list here is also sorted by the key values of the most probable tuple alternatives.

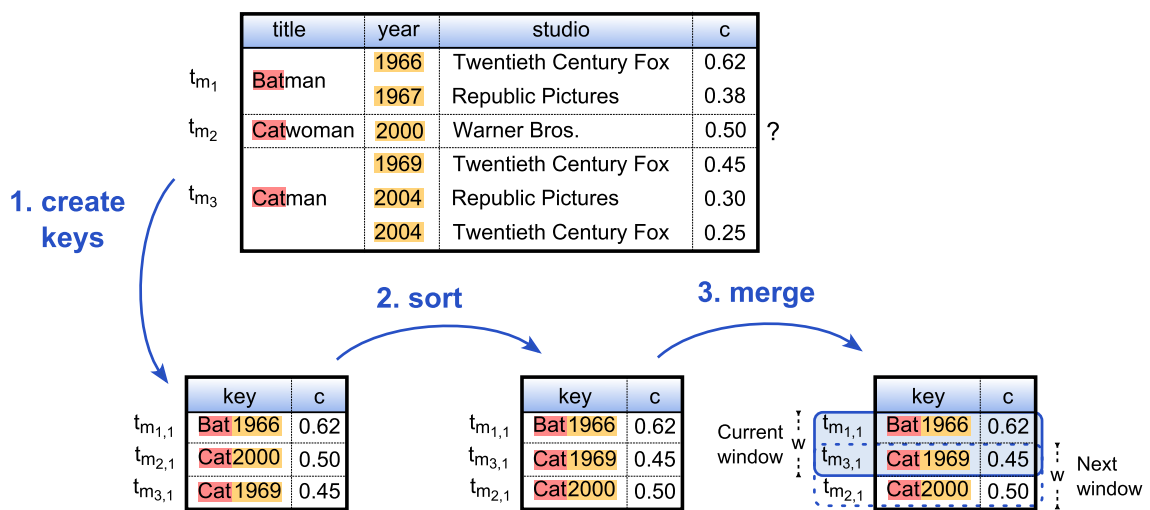


Figure 3.4.: The key per tuple strategy applied to the Sorted Neighborhood Method. In this example, a tuple’s key value is computed from its most probable alternative.

³Note that alternatives representing tuples’ absence are *not* chosen.

But it seems possible that an alternative computed with a mediating strategy represents a tuple better than one of the already existing ones. As pointed out in [PvKdKR09], conflict resolution techniques used for fusion of certain data might be suitable for mediating strategies. In [BN08], some conflict resolution techniques are mentioned that seem usable for computing attribute values for a tuple representative from the tuple's alternatives. We describe the concepts of some techniques with the help of the tuple depicted in Figure 3.5. The *cry with the wolves (CWTW)* technique describes choosing the most often occurring value. In our example, the production year chosen by CWTW for the tuple representative would be 2004. Obviously, this production year has an overall confidence of only $0.15 + 0.15 + 0.05 = 0.35$, whereas 1969 is the correct production year with a confidence of 0.4. For our purpose, it seems logical to adapt CWTW in such a way that not the most often occurring, but the *most probable value* is chosen for the representative.

By another technique, referred to as *meet in the middle (MITM)*, an average value is chosen. For techniques like Blocking, where only tuples with equal keys are paired it makes sense to take one of the existing values, e.g. the median: 1996 for the production year in our example. But if a technique like the Sorted Neighborhood Method is used where tuples are paired due to key value *similarity*, choosing the expectancy value can be even more appropriate. The expectancy value of the production year of the illustrated tuple is $1969 \cdot 0.4 + 1996 \cdot 0.25 + 2004 \cdot (0.15 + 0.15 + 0.05) = 1988$.

By *roll the dice (RTD)*, one of the given values is picked randomly. We use this simple technique as a fallback strategy, when the primary technique brings an ambiguous result. The two most probable studio names in our example, namely *Twentieth Century Fox* and *Republic Pictures*, both have the same confidence of 0.4, so that the adapted CWTW approach would not bring one, but two results. In this case, it seems appropriate to just pick one value at random.

Take the information (TTI) specifies that any value is preferred to a missing value and makes sense in combination with any of the other techniques.

Naturally, different techniques may be used for different attributes, e.g. MITM can be used for numbers, while string values can be processed with CWTW first and post-processed with RTD in case of an ambiguous result.

3.1.3. Key per alternative

It is also possible to create key values not for tuples, but for tuple alternatives, so that tuples may have more than one key value computed for them. As a consequence, a tuple can be assigned to several blocks (Blocking) or can appear several times in the sorted list (SNM) as shown in Figure 3.6. Since tuples may appear several times in the sorted list and hence also more than once in a window (SNM), the number of resulting tuple pairs per window can vary. In order to prevent this effect, our implementations of this approach work with a modified window: Like in our example, the window size is increased by one for every added tuple that already was in the window, so that each window holds the same

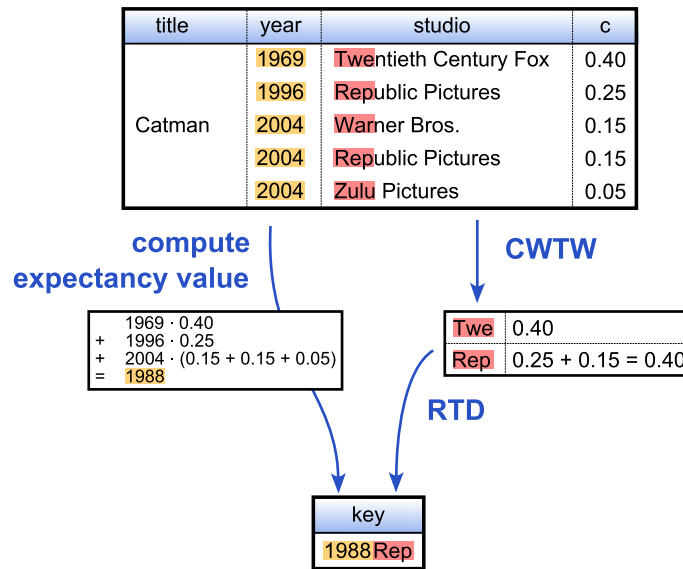


Figure 3.5.: Different variants of the key per tuple strategy can be combined. The key value of the illustrated x-tuple is composed of the expectancy value of the year and a prefix of the most probable studio name.

number of *different* tuples⁴.

There are many approaches to decide which alternatives should also be used for key value creation. One of them is to simply use *all alternatives*. Another idea is to use only a predefined number of alternatives per tuple, e.g. the two most probable alternatives, or to use the most probable alternative of every tuple and, in addition, a share of the remaining alternatives, for example the 10,000 most probable remaining alternatives in the database. Just as well, a confidence threshold could be defined and only the remaining alternatives with a confidence exceeding the threshold could be chosen.

3.1.4. Modification: combining key values

One aspect of certain key generation that has not been considered so far is that different alternatives of the same tuple can have equal key values. Problems that can arise here can be prevented by computing a key value for every alternative first and then combining alternatives with equal key values for every tuple.

It makes sense to combine the key values when building the most probable possible worlds to make sure the constructed SNM lists really are different ones and the most likely key value set is used for search space reduction. For example, if an alternative is replaced by another alternative of the same tuple with the same key value, the resulting SNM list is identical to the predecessor list. As to building highly dissimilar worlds, though, we assume that better results can often be achieved without combining key values, because

⁴Actually, the last window may hold less tuples.

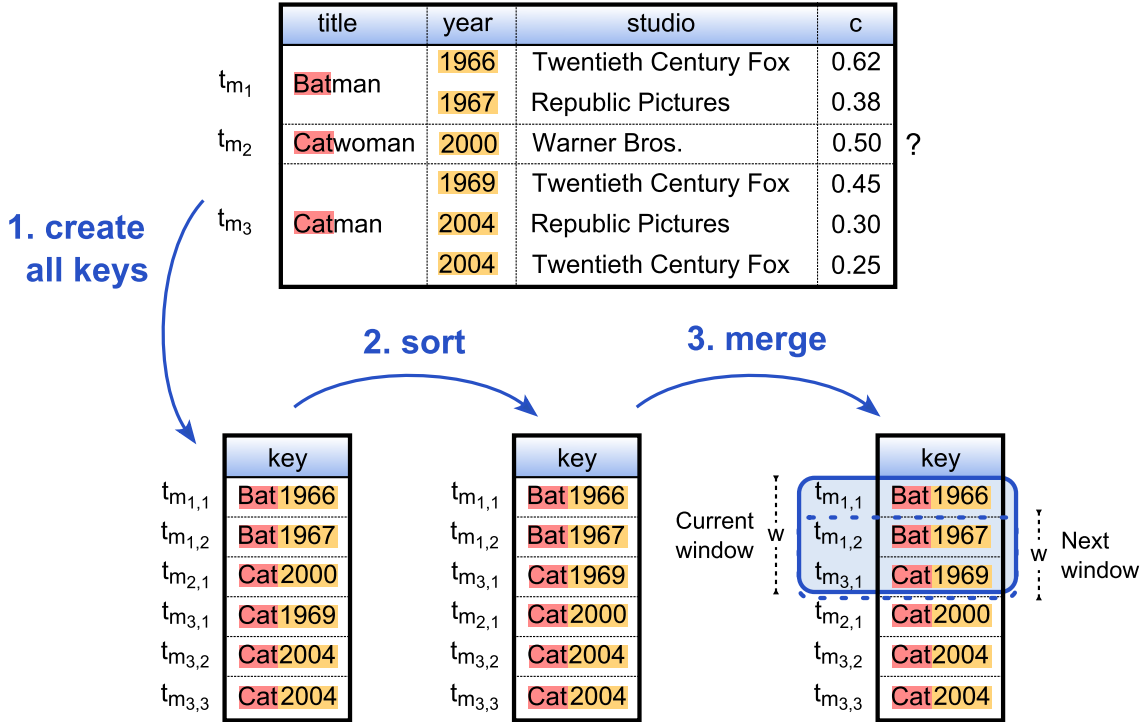


Figure 3.6.: The key per alternative strategy applied to the Sorted Neighborhood Method where a key value is generated for every alternative.

the number of constructed worlds⁵ in this approach is very small anyway and might even be reduced when equal key values are combined for every tuple.

Combining key values can also make sense for *deciding* key per tuple strategies: In Figure 3.7, tuple alternative $t_{m_{3,1}}$ is the most probable alternative, but with a confidence of 0.45 its key value is not the most probable, since alternatives $t_{m_{3,2}}$ and $t_{m_{3,3}}$ agree in their key value and have a combined confidence of $0.3 + 0.25 = 0.55$. This example demonstrates that the combined key value of some alternatives may represent a tuple better (or at least may be more probable) than the key value of the most probable alternative.

Although computing and combining key values before choosing a representative does not make much sense for *mediating* strategies, since the representative here is computed from attribute values (and not from key values), it can actually be beneficial to compute and combine *subkeys* first: For example, if the key of a tuple consists of the first three characters of the movie title concatenated with the production year, all possible title subkeys can be computed and combined, so that the most probable subkey value is chosen – and not the subkey value of the most probable title.

When using *all* alternatives, combining key values does not change the SSR result, because tuples are only considered once per block and only once per window with our improved windowing technique. But it can be beneficial for SNM to combine key values nonetheless,

⁵Even though we use the term 'possible worlds' here again for the sake of simplicity, we want to make clear that we actually do not build possible worlds when combining key values first, but *sets of possible worlds* containing different alternatives of the same tuple with equal key values.

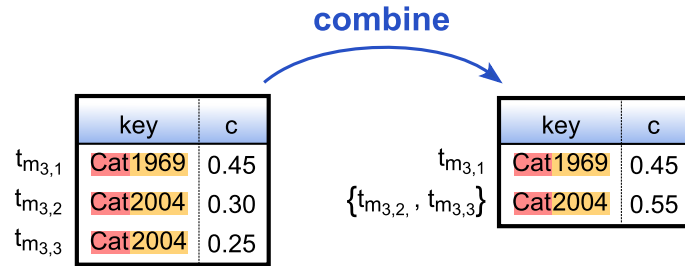


Figure 3.7.: When combining key values, the confidences of equal key values are summed up.

because the sorted list of tuples is reduced and, as a consequence, the sorting can be executed faster.

When *not all* alternatives are used, combining the key values does not only speed up search space reduction, but also has an impact on its result. If for example the two most probable alternatives of every tuple are used, there might be tuples with two alternatives in the list having the same key value. This cannot happen, if key values are combined first.

We always combine key values in our implementations when computing key values per alternative, because it can make the SNM list shorter and thus can accelerate the procedure. Furthermore, it maximises the diversity of the key values of the selected alternatives.

3.2. Processing probabilistic keys

In [PvKdKR09], it is proposed to deal with data uncertainty by generating a *probabilistic key value* for every x -tuple first and to resolve the uncertainty when the tuples are sorted into a list according to their probabilistic key values. The probabilistic key value of an x -tuple is computed in such a way that every x -tuple alternative has a corresponding key value alternative. The variant of the probabilistic key strategy discussed here is applicable to the Sorted Neighborhood Method, but not to Blocking. We do not discuss a probabilistic key approach for Blocking, because it is out of the scope of our work.

In order to sort the tuples into a list by their probabilistic key values, we perform pairwise comparisons of the tuples with each other as illustrated in Figure 3.8: There are two tuples t_{m_1} and t_{m_2} with two alternatives each. The tuple alternatives can only occur in four combinations, where each of these combinations occurs with a certain probability. In order to determine in which order the tuples should be inserted into the SNM list, their *most likely order* is computed: Alternatives $t_{m_{1,1}}$ and $t_{m_{2,1}}$ are part of the same world with a probability of $c(t_{m_{1,1}}) \cdot c(t_{m_{2,1}}) = 0.8 \cdot 0.6 = 0.48$. In a possible world, in which those alternatives are both present, t_{m_1} comes after t_{m_2} in the SNM list, because the key value of $t_{m_{1,1}}$ is greater than the key value of $t_{m_{2,1}}$ (BAT1969 > BAT1959). Tuple t_{m_1} comes *before* t_{m_2} in two combinations with a probability of $0.32 + 0.08 = 0.4$ and the tuples have both equal key values with a probability of only 0.12. Accordingly, the two tuples are sorted into the list in their most likely order: t_{m_1} after t_{m_2} .

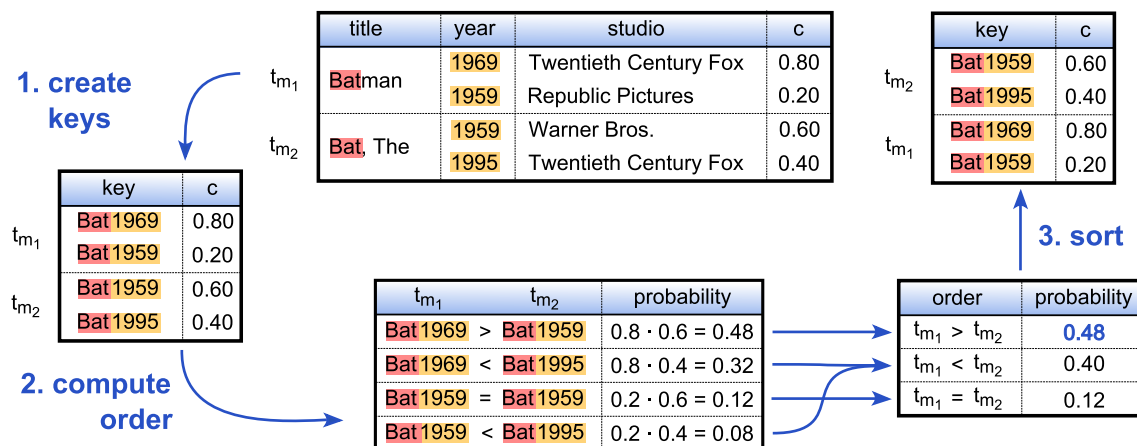


Figure 3.8.: The probabilistic key approach applied to the Sorted Neighborhood Method. The tuples are sorted considering all tuple alternatives' key values.

If two tuples are most likely equal, their list order is determined by their second most likely order and by their database IDs to guarantee a deterministic list order.

Another idea is to not just compare key alternatives by a comparison function, but by a distance measure, so that it would also be taken into consideration *to what extent* one key value is greater or smaller than another – and not only whether at all.

Besides processing probabilistic key values as described above, the sorting function can be chosen in such a way that this strategy corresponds to the certain key per tuple or key per alternative strategy. For example, if the sorting function sorts tuples by their most probable key alternative, the procedure is practically the same as one of the certain key per tuple variants with combined keys. Similarly, sorting tuples several times into the SNM list by all key alternatives corresponds to one of the certain key per alternative strategies with combined keys.

4. Experiments

Our implementations of the search space reduction techniques are actually not standalone pieces of work, but part of the project *QloUD*¹ concerning integration of probabilistic data sources. We implemented the search space reduction techniques in Java. The input for an SSR technique is always a set of x-relations, from which the *reduced search space* (a set of tuple pairs) is computed as output.

However, with the goal of our work not only being the implementation of the discussed SSR techniques, but also to try them out and to evaluate them regarding their effectiveness and the quality of their results, we were facing two challenges: First, a great amount of data like the number of actual matches, actual non-matches and so forth had to be collected during the experiments and many metrics had to be computed in order to make a meaningful comparison and evaluation of the different SSR techniques. Doing this by hand is very time-consuming and error-prone – let alone formatting the results. The second, probably more severe challenge was the absolute lack of appropriate testing data. We are simply not aware of any probabilistic benchmarking data at all on which to perform our experiments. In this chapter, we begin with an overview over the testing framework for our implementations of the adapted SSR techniques and a description of our test data in the first two sections. We then explain how we configured our experiments in the third section and, in the fourth section, compare the different SSR techniques on the basis of experimental results.

4.1. Testing framework

The documents in the appendix were generated with our testing framework: Data like the number of correctly identified duplicate tuple pairs were recorded and then used to rate the techniques according to the metrics discussed in Subsection 2.3.1. Furthermore, we did not only automatise data aggregation during our SSR experiments, but also the generation of diagrams to illustrate the results.

Besides the above mentioned metrics we also consider the execution times of the different SSR techniques. Our experiment hardware was a notebook with a Core 2 Duo P8700@2.8GHz and 2 · 2048MB DDR2 PC 6400 memory. Since we executed all experiments in Java under Windows, the measured execution times can be expected to be rather inaccurate: The duration of an experiment can, for example, be influenced by background

¹Quality of Uncertain Data (<http://vsis-www.informatik.uni-hamburg.de/projects/QloUD/index.php>)

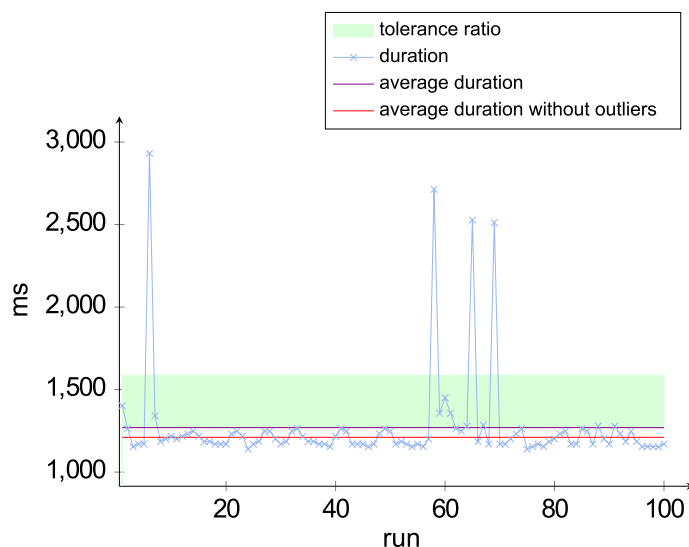


Figure 4.1.: Computing the average duration of an experiment. First, the average duration of all runs is built (purple line). To reduce the impact of outliers, only values that do not exceed 125% of this value are then used to determine the average duration of an experiment without outliers (red line).

tasks running on the operating system or by Java’s garbage collection.

To keep the impact of side effects on the runtime minimal, we stopped all running processes not necessary for the experiments and disabled swap files to prevent overshoots as far as possible. Furthermore, we executed all our experiments several times in a row and computed an average value to get a better result. But since there were still obvious overshoots, by which we mean values that exceed 125% of the average value, we took the average value *excluding* the outliers as final result.

Figure 4.1 illustrates the execution times of an experiment over 100 runs and the average runtime with and without outliers. We are of the opinion that, if two SSR techniques differ greatly in their measured execution times, it can be assumed that the one that was faster according to the measured data actually is faster.

Average duration	
first experiment	451.35 ms
second experiment	449.92 ms
third experiment	452.57 ms
fourth experiment	452.40 ms

Table 4.1.: The average duration of an experiment can be reproduced in independent experiments.

To demonstrate that the average duration is a reproducible value, we repeated one Blocking experiment series (key per tuple, most probable alternative) four times. The single average

duration values are presented in Table 4.1. Ranging from about 449ms to about 453ms, the maximum deviation is smaller than 1%. Detailed diagrams can be found in Appendix A.

4.2. Test data

Since there are only few *probabilistic* database systems at all, it is hardly surprising that we did not find probabilistic data sets to test our probabilistic search space reduction techniques.

Our solution to this problem was to generate probabilistic data ourselves. In order to make the data as realistic as possible, we decided to use real-world data from an existing certain database. So we extracted title, production year, studio and director of about 1,500,000 movies from the online movie database *IMDb*² with the Java application *JMDB*³ and stored the data to an *HSQLDB*⁴.

But as many movies are released several times in different versions, there were still quite many duplicates in the data: Several movie tuples had the same title, only that some had additional information put in brackets at the end of the title string. To keep the database free from duplicates, we removed all tuples with additional information in brackets in their title strings. Then we identified tuples with identical titles and removed most of them in such a way that every title became unique. Our certain real-world movie data set had been reduced strongly to a size of about 300,000 tuples, but was also duplicate-free.

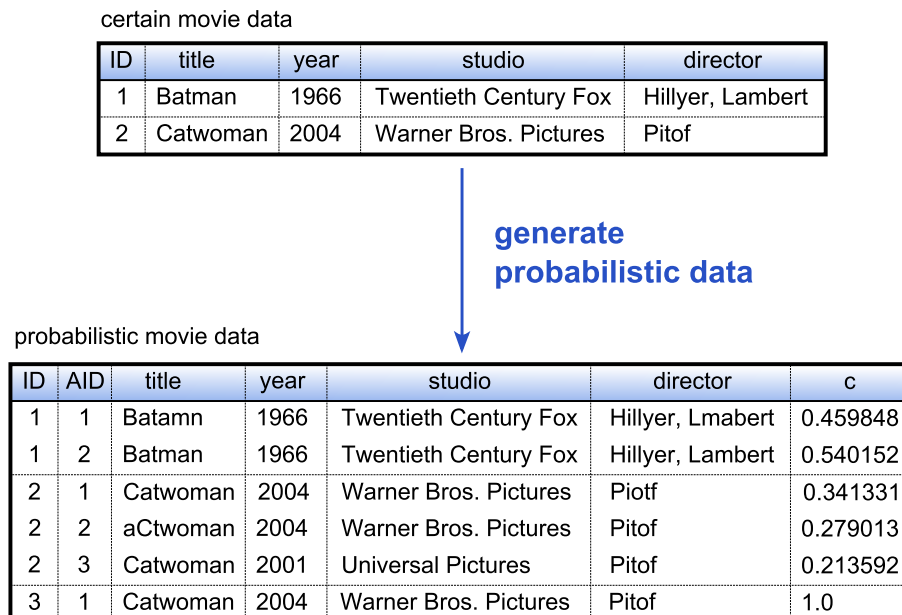


Figure 4.2.: Probabilistic data can be generated from certain data with ProbDataGen.

²The Internet Movie Database (<http://www.imdb.com>)

³Java Movie Database (<http://www.jmdb.de>)

⁴HyperSQL DataBase (<http://hsqldb.org>)

We programmed a Java application named *ProbDataGen* to generate probabilistic data from the duplicate-free certain data. With *ProbDataGen* it is possible to choose among several HSQL databases holding certain movie data to generate a probabilistic movie database with duplicates, where the user can make several adjustments, e.g. the number of duplicates or the maximal number of alternatives per tuple. The generated probabilistic data are then stored in such a way that they can be loaded into our x-tuple data model and can be used for SSR experiments: In addition to the actual movie attribute values, every HSQLDB movie tuple has an x-tuple ID, an alternative ID and a confidence value. So every *HSQLDB tuple* represents an *x-tuple alternative*.

Figure 4.2 illustrates probabilistic data that were generated from certain data with *ProbDataGen*. The process of generating a probabilistic database is performed in the following steps:

1. *Create a new database of a given size:* First, a new database is created and filled with the desired number of randomly chosen (certain) movie tuples.
2. *Generate x-tuple IDs:* Every x-tuple is assigned a unique x-tuple ID.
3. *Generate duplicates:* Now, some movie tuples are duplicated, so that they appear twice in the movie relation. The duplicates are then assigned x-tuple IDs. Since it is necessary for the computation of the metrics discussed in Subsection 2.3.1 to know the duplicate pairs, i.e. the actual matches, they are also stored to the database.
4. *Generate confidence values:* In this step, a confidence value is assigned to every tuple in the database. The confidence values are generated randomly to some degree, but they are also influenced by several parameters. For example, the user can define the probability of generating a confidence value smaller than 1, i.e. the probability to turn a tuple into a maybe tuple.
5. *Adding alternatives:* At this point, some alternatives are added to the tuples, so that there actually are x-tuples with more than one alternative. Adding a few alternatives to an x-tuple means to duplicate the (only) x-tuple alternative several times and distribute the tuple's confidence among all alternatives afterwards. Since all alternatives of that tuple are identical except for the confidence value, their alternative IDs are modified in such a way that the alternatives are enumerated from 1 to the number of the x-tuple's alternatives in order to guarantee that the combination of x-tuple ID and alternative ID is unique for every x-tuple alternative. The user can define many parameters for this step as well, e.g. the minimal alternative confidence or how many alternatives at least and how many at most are generated for a tuple.
6. *Add errors:* Finally, some errors are added to the x-tuples. Whether errors are added to a certain attribute value and of what kind or how serious they are, is decided randomly according to several user-defined parameters, but always following two rules: The first rule is that the chance of generating an error is greater for alternatives with small confidence values. The second rule is that the alternatives of a tuple have to differ somehow, when the errors have been added, since x-tuple

alternatives are mutually exclusive, i.e. they must not be identical.

Errors that can occur in the production year are simply wrong numbers. Strings, i.e. the title, director and studio, are affected by typo-like errors, for example missing or transposed neighbouring characters and wrong spelling such as a 'novie' instead of 'movie'. The director and studio attribute values are even exchanged with other values to simulate not only erroneous, but downright wrong data.

4.3. Experiment configuration

Doing experimental research on the differences of search space reduction techniques is a wide field. There are countless experiment configurations that might be worth trying for one reason or another. In order to keep the number of experiments small, though, we chose the experiments with our primary goal in mind: to draw a meaningful comparison between the different SSR variants presented in Chapter 3.

In the first subsection, we describe our experiments to determine a reasonable database size for our main experiments. The second subsection is about the properties of the database used for the experiments. In the third subsection, we discuss several key designs (and window sizes) for our SSR techniques.

4.3.1. The database size

In spite of hardware limitations, it was our intention to execute the SSR experiments on a preferably large database, i.e. a database with many alternatives. On the other hand, we had to make sure the search space reduction techniques could still be executed at all, since we only used *3GB* memory for search space reduction. When nearly as much memory is in use as is available, Java's garbage collection is done with a slower more thorough algorithm than usual, the MarkSweep garbage collector. Using this garbage collector can make the duration values of the SSR experiments less comparable, because the garbage collection overhead grows faster. We chose the number of alternatives in the database in such a way that this particular – more expensive – garbage collector is not used at all.

	DB _{100k}	DB _{200k}	DB _{300k}	DB _{400k}
overall alternatives	100,000	200,000	300,000	400,000
GC time, PS Scavenge (ms)	5,661	22,753	25,521	54,208
GC time, PS MarkSweep (ms)	0	0	8,120	22,037

Table 4.2.: We chose to execute the experiments on DB_{200k}.

We tried search space reduction with several SNM and Blocking variants on databases with 100,000 to 400,000 tuple alternatives and recorded the maximal garbage collection time for

each database. Table 4.2 clearly shows that the MarkSweep garbage collector is not used for SSR at all on a database with 100,000 or 200,000 alternatives, so we decided to use a database with 200,000 tuple alternatives. The search space reduction variants we used are listed in Appendix B.

4.3.2. The database configuration

Having determined a suitable number of alternatives, it was still unclear what difference the number of duplicates in the database or the number of alternatives *per tuple* made for the different SSR techniques. Table 4.3 shows the properties of the different databases we used. In Appendix C, there are more information on the corresponding experiments. All databases had 200,000 tuples, but differed in the number of duplicates or the number of alternatives per tuple.

We took a database with 3,000 duplicates and maximally 3 alternatives per tuple as reference database and created two more databases with 30,000 and 50,000 duplicates and two other databases with a maximum of 6 and 12 alternatives per tuple. Then we performed some search space reduction techniques on all databases and compared the pairs completeness, the reduction ratio and the precision. The SSR techniques were configured rather arbitrarily: We just built the key values by concatenating the first three characters of the title string and the production year and used a window size of 3 for the SNM variants.

	DB _{3kDP}	DB _{30kDP}	DB _{50kDP}	DB _{max6Alt}	DB _{max12Alt}
overall tuples	113,000	130,000	160,000	73,000	43,000
overall alternatives	200,000	200,000	200,000	200,000	200,000
duplicate pairs	3,000	30,000	50,000	3,000	3,000
min alternatives	1	1	1	1	1
max alternatives	3	3	3	6	12

Table 4.3.: Database configuration experiment series – databases.

Although the actual results differed from database to database, the proportions of the results achieved by the different SSR techniques were very similar each time. There is one exception, though: The pairs completeness of search space reduction based on the *highly dissimilar worlds* variant of the possible worlds strategy got much better with more alternatives per tuple. And this is not very surprising, because the number of possible worlds constructed here depends on the number of alternatives per tuple. Ranking all other SSR techniques by their pairs completeness yields the same order, independently of the database. For reduction ratio and precision, the order even remains the same without any exceptions.

In other words, our experiments suggest that an SSR technique is – in relation to the other techniques – *roughly* as good on a database with few duplicates or alternatives per tuple as on a database with many. We chose to execute all main SSR experiments on

database $DB_{max6Alt}$ with 3,000 duplicates and at most 6 alternatives per tuple. The detailed database properties are listed in Table 4.4.

$DB_{max6Alt}$	
overall tuples	113,000
overall alternatives	200,000
duplicate pairs	3,000
min. alternatives	1
max. alternatives	6
smallest tuple confidence	0.080025
biggest tuple confidence	1.0
smallest alternative confidence	0.08
biggest alternative confidence	1.0

Table 4.4.: Some properties of the database we executed subsequent experiments on.

4.3.3. The SSR technique configuration

Finally, we wanted to find good configurations for Blocking and the Sorted Neighborhood Method in order to make the comparison of the different SSR techniques more significant: After all, comparing a well-configured approach with an ill-configured one could distort the results. In Appendix D, there are two tables holding information on the results of performing Blocking and the Sorted Neighborhood Method, e.g. the pairs completeness, reduction ratio and precision that were achieved using a certain key function. All alternatives (key per alternative) with combined key values were used for SNM. For Blocking, we did not combine key values, because it slows down Blocking (see Subsection 4.4.3).

key function	title:prefix(3), year:attribute	title:prefix(3), year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute	title:attribute
window size	2	12	12	2	2	12
pairs completeness	0.387666666667	0.759333333333	0.924666666667	0.889333333333	0.889	0.924666666667
reduction ratio	0.999976391051	0.999762498329	0.999767230456	0.999977709099	0.999977705026	0.999767230926
precision	0.007715783188	0.001502325373	0.001866625754	0.018747145417	0.018736695681	0.001866629522
average duration (ms)	1,668.66	34,084.96	37,557.56	1,773.7	1,579.7	34,884.73

Table 4.5.: Sorted Neighborhood Method: configuration experiment series. The experiments were executed using all alternatives with combined key values.

Table 4.5 shows the results for the Sorted Neighborhood Method. We started with the concatenation of the first three characters of the title and the production year as key with the minimal window size 2, which resulted in a rather low pairs completeness of just above 0.38 and a precision of less than 0.01. Increasing the window size to 12 improved the pairs completeness significantly to over 0.75, decreased the precision even further and increased

the average experiment duration by factor 20. Using the entire title instead of a three-character prefix led to another improvement of the pairs completeness to over 0.92, slightly increased reduction ratio, precision and duration. Reducing the window size to 2 at that point, also reduced the pairs completeness a little to about 0.88 and the duration strongly by more than factor 20, but at the same time increased the precision by the ten-fold. Using the raw title only yielded very similar, but slightly worse results than using title and year – with a window size of 2 as well as 12.

key function	title:prefix(3), year:attribute	title:prefix(8), year:attribute	title:attribute, year:attribute	title:attribute	title:prefix(6)	title:prefix(8)
pairs completeness	0.539	0.465	0.384333333333	0.754333333333	0.889333333333	0.859
reduction ratio	0.999775037245	0.99999888949	0.99999819405	0.99999641002	0.999943811078	0.999981497056
precision	0.001125839157	0.216111541441	1	0.987347294939	0.007437244763	0.021814764964
average duration (ms)	3,303.53	910.03	1,004.13	711.36	1,180.86	857

Table 4.6.: Blocking: configuration experiment series. The experiments were executed using all alternatives without combining key values.

To find a good key for Blocking, we made similar experiments. The results are presented in Table 4.6. First, we tested the key composed of the three-character prefix of the title plus the production year. Since the precision was very low, we tried a larger title prefix of 8 characters and the entire title as well. The pairs completeness decreased, while reduction ratio and precision grew, when the title prefix did. And this makes sense, because tuples are less likely assigned to the same block when they must agree on the first 8 (or all) instead of 3 characters of their title. Although with the key composed of the full title and the production year pairs completeness was only about 0.38, it is noteworthy that actually a *perfect precision*, i.e. a precision of 1, could be achieved.

Using the raw title attribute alone as key improved the pairs completeness to over 0.75 and reduced reduction ratio and precision a little. Now, the pairs completeness could easily be improved at the expense of reduction ratio and precision by only using a prefix of the title: the shorter the prefix, the better the pairs completeness.

With a title prefix of the length 6, the pairs completeness was over 0.88, which is comparable to the result achieved by SNM with our chosen key and window size, but also with a worse precision of less than 0.01.

We chose to use a key composed of the raw title and the production year for the Sorted Neighborhood Method, because it led to a good pairs completeness and the best precision in this experiment series. Regarding Blocking, we decided for the title prefix of the length 8 as key, because it led to a similar precision as the SNM variant just mentioned, and also to a good pairs completeness of 0.859.

4.4. SSR experiments

This section is devoted to the comparison of our implementations of the adapted SSR variants presented in Chapter 3. We compare the different implementations in separate subsections and explain which implementation we chose for Blocking and which for SNM for the overall comparison in Subsection 4.4.5. To have well comparable results, we chose implementations that seemed best for search space reduction in one single pass⁵.

4.4.1. Possible worlds experiments

There are only two variants of the multi-pass over possible worlds strategy, namely *building the most probable worlds* and *building highly dissimilar worlds*. Our first experiments concerned the Sorted Neighborhood Method and the results are visualised in Figure 4.3. For more details see Appendix E. We tested building the 6 most probable worlds and achieved a pairs completeness of 0.815 and a precision of roughly 0.02. Doubling the amount of constructed worlds almost doubled the average duration, but had virtually *no other effect* on the result. This corresponded to our expectations: Building twice as many worlds is roughly twice as expensive, but since the constructed worlds only differ in very few tuple alternatives, the result remains pretty much the same.

Next, we tried the other variant, building highly dissimilar worlds, with only 4 worlds and observed a pairs completeness of about 0.88 and a precision of almost 0.015. Increasing the number of different worlds increased the pairs completeness and decreased the precision slightly where the effect on the result got smaller for very additional world. The average experiment duration increased steadily. Building 6 dissimilar worlds, i.e. the maximum number of different worlds, was better with combined key values and yielded a pairs completeness of 0.889 and a precision of little more than 0.014.

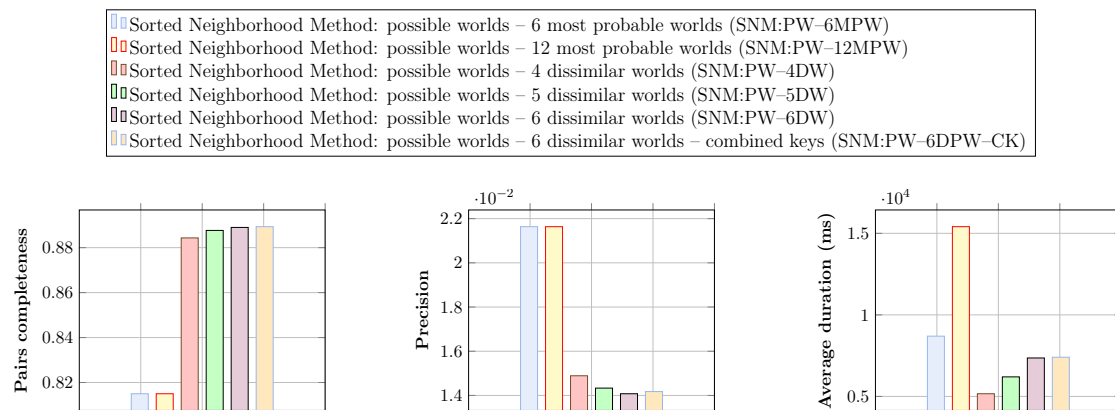


Figure 4.3.: Sorted Neighborhood Method: possible worlds experiment series.

⁵By single-pass SSR strategies we also mean the multi-pass over possible worlds strategy, although it is, strictly speaking, no single-pass strategy.

Similar results were achieved by the Blocking experiments, which are illustrated in Figure 4.4 and presented in more detail in Appendix F. Blocking on the 6 most probable worlds was – again – slower than on 4 highly dissimilar worlds and resulted in a worse pairs completeness. Furthermore, it was even worse w.r.t. the precision, if only slightly. Each time we increased the number of dissimilar worlds, we observed an improved pairs completeness *and* a (slightly) improved precision. Naturally, the experiments with more dissimilar worlds also took more time.

Finally, we tried Blocking on 6 highly dissimilar worlds with an abbreviated title prefix of length 7 to bring the precision down a little to the level of the corresponding SNM variant. The result was a pairs completeness of 0.867, i.e. about 2 percents under the corresponding SNM pairs completeness, at a precision of roughly 0.013 (0.014 with SNM).

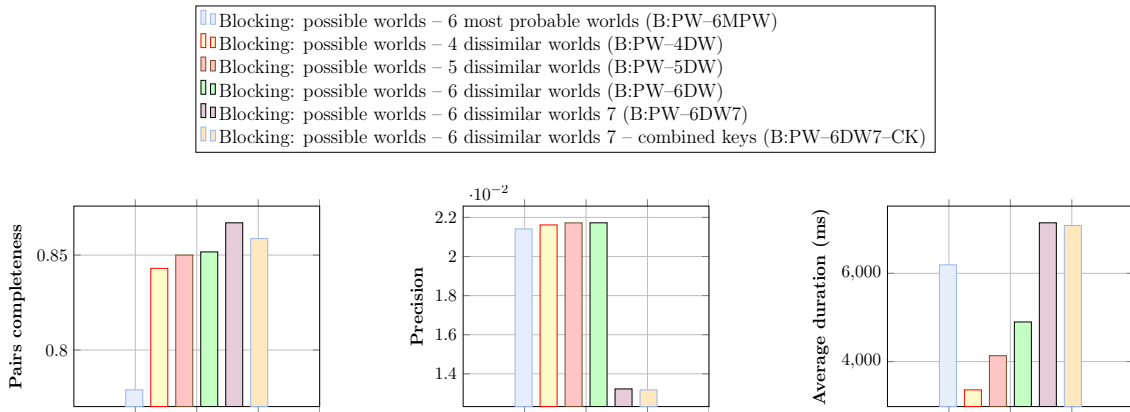


Figure 4.4.: Blocking: possible worlds experiment series.

In summary, building dissimilar worlds was notably faster than building some of the most probable worlds, while it also resulted in a comparable precision and a pairs completeness that was more than 6 percent better for SNM and even over 8 percent better for Blocking.

4.4.2. Key per tuple experiments

The detailed results of our key per tuple experiments are shown in Appendix G (SNM) and Appendix H (Blocking). The results of the experiment series concerning the Sorted Neighborhood Method are shown in Figure 4.5. We started with two deciding key per tuple variants. Our initial experiment was using the *most probable alternative* (MPA) variant, i.e. computing one key value from each tuple’s most probable alternative, and resulted in a pairs completeness of almost 0.82 and a precision of about 0.02. Afterwards, we tried the *cry with the wolves* (CWTW) variant, i.e. to combine the key values and use the most probable one. The result was almost the same. We also tried the title attribute alone as key for both variants instead of title plus production year, but this did not change the result significantly either.

The *mediating* variant we chose for our experiment used a key composed of the most probable title and the expectancy value of the production year. And again, the results remained practically unchanged, but when we finally used an eight-character prefix instead of the entire title, pairs completeness and precision decreased.

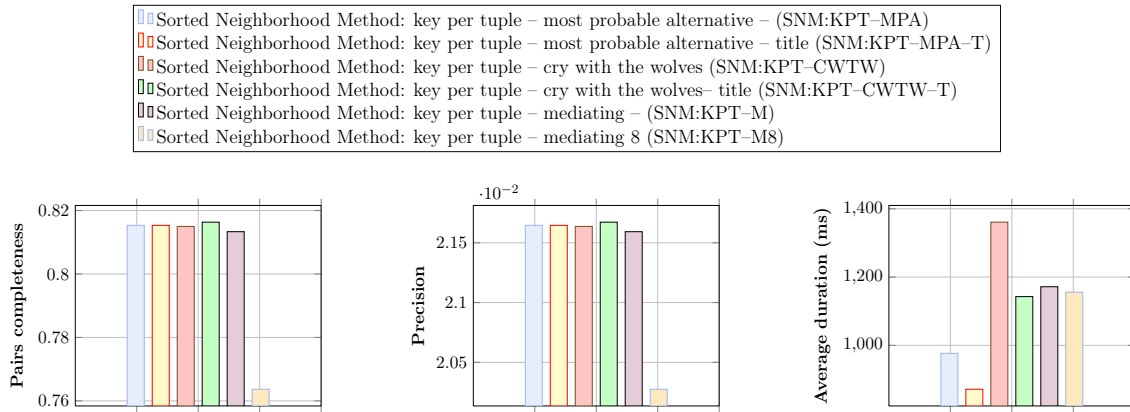


Figure 4.5.: Sorted Neighborhood Method: key per tuple experiment series.

As Figure 4.6 illustrates, the Blocking experiments returned similar results: Both the MPA and the CWTW variants led to a pairs completeness between 0.77 and 0.78 and a precision of about 0.02. For Blocking, we used a different mediating strategy where the key was a concatenation of the title and the median of the production year instead of the expectancy value. A pairs completeness of only 0.321 was the result, but the precision was once again *perfect*. We tried to improve the pairs completeness by trimming the title subkey to 8 and later to 3 characters. However, the pairs completeness did not even reach 0.50, before the precision dropped to less than 0.002.

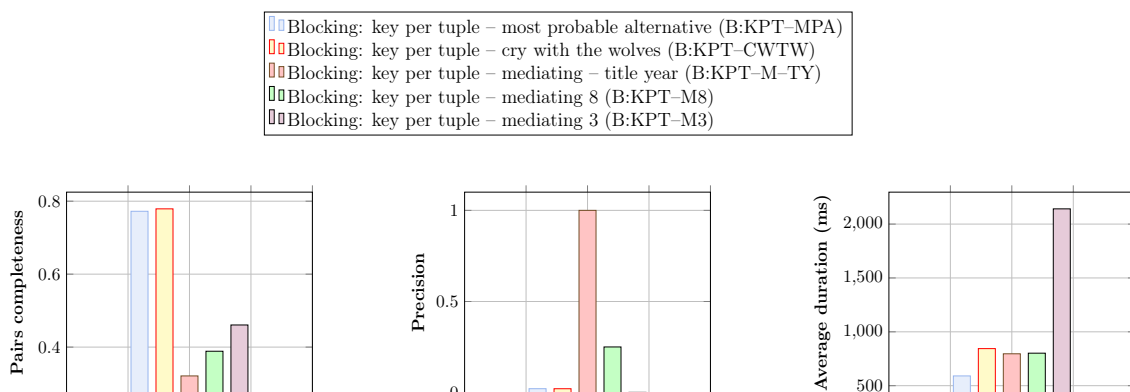


Figure 4.6.: Blocking: key per tuple experiment series.

It is hard to draw a meaningful comparison between the key per tuple variants discussed above, since most of them achieved very similar results. The cheapest of the variants – and arguably the most intuitive – is simply using the most probable alternatives to compute

key values, and it worked nearly as well as any of the other variants, only a bit faster. However, using CWTW tended to improve the pairs completeness slightly, and hence we chose the CWTW variants for SNM and Blocking for the overall comparison.

4.4.3. Key per alternative experiments

Before starting the experiments to compare the different variants of the *key per alternative* strategy we conducted experiments concerning the question what effect combining the key values has on the SSR result. We compared different variants with and without combining key values for both the Sorted Neighborhood Method (Appendix I) and Blocking (Appendix J).

First, we tried the 2 most probable alternatives with and without combined key values for SNM, but could not make out any noteworthy differences in the results apart from a small acceleration. We guessed that combining key values might have had so little effect, since the key was composed of the entire title and the production year and hence there simply were barely any equal key values to combine. So we trimmed the key to a three-character title prefix concatenated with the production year and repeated the experiments: Pairs completeness and precision decreased considerably, but the differences between the variant using combined keys and the other variant were again negligible. Unsurprisingly, pairs completeness, reduction ratio and precision were exactly the same when using all alternatives.

Blocking delivered different results, though: Combining key values led to better pairs completeness *and* precision with little negative effect on the average duration. Using all alternatives with combined key values only *slowed the procedure down*.

In conclusion, combining key values led to slightly better pairs completeness – and in case of Blocking better precision – and was faster for SNM while slower for Blocking. For this reason, we chose to combine key values for all key per alternative variants apart from Blocking with all alternatives.

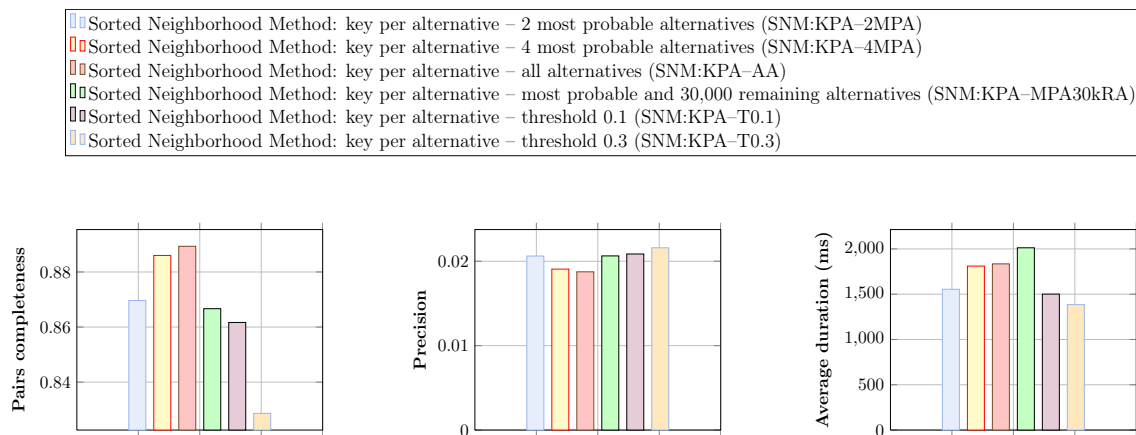


Figure 4.7.: Sorted Neighborhood Method: key per alternative experiment series.

In Appendix M, detailed information about our SNM key per alternative experiments can be found. Figure 4.7 also illustrates the results. All our key per alternative variants use the most probable key value of each tuple, but there are different approaches to select additional key values. We started the experiment series with variants using the 2 and 4 most probable as well as all alternatives per tuple. The pairs completeness ranged from roughly 0.87 with 2 alternatives and a precision of over 0.021 to almost 0.89 with all alternatives and a precision of nearly 0.019.

After that, we tried using the 30,000 most probable tuple alternatives in the database remaining after the most probable alternatives had already been chosen. This turned out to be even slower than using all alternatives, probably because sorting the list of remaining alternatives by the confidence is quite expensive. Another way to choose remaining alternatives not on tuple, but on database level is to select only alternatives with a confidence exceeding a certain threshold value. A threshold of 0.1 yielded a result comparable to using the 30,000 most probable remaining tuple alternatives, but it was significantly faster – even faster than using only the 2 most probable alternatives per tuple. Increasing the threshold reduced pairs completeness and average duration, but led to a higher precision.

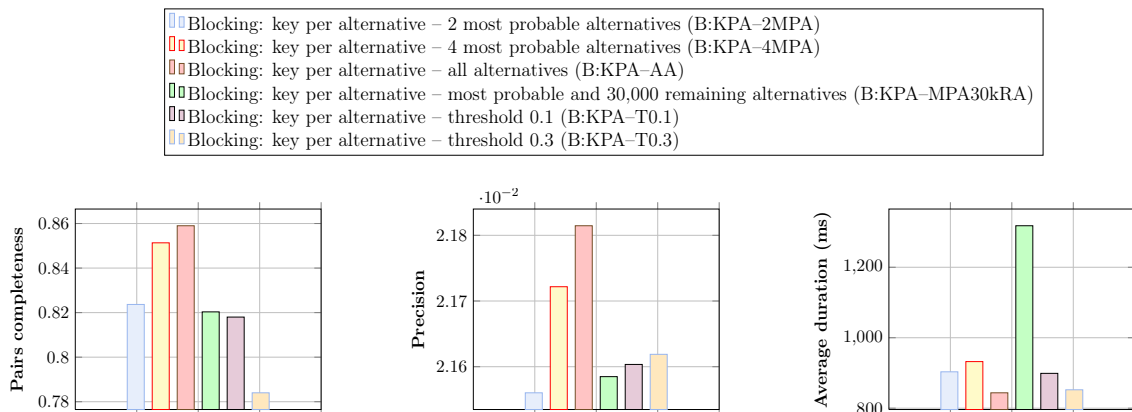


Figure 4.8.: Blocking: key per alternative experiment series.

An overview over the results of the Blocking experiment series is given in Figure 4.8. More details are presented in Appendix L.

The best results in terms of pairs completeness for the Sorted Neighborhood Method as well as for Blocking were achieved using all alternatives. The other SNM variants of this approach all showed a somewhat lower pairs completeness at a higher precision, while the other Blocking variants even had *worse* precision and average duration.

4.4.4. Probabilistic key experiments

In order to find out what results could be achieved using probabilistic key values, we repeated the SNM configuration experiment series described in Section 4.3.3 with probabilistic key values. The results are shown in Figure 4.9. For more details see Appendix M.

Using a title prefix and the production year as key and the minimal window size proved to be very ineffective with a pairs completeness of only 0.36 and a precision smaller than 0.01. Increasing the window size to 12 led to a ten-fold increase of the experiment’s average duration, while the precision decreased by factor 10 and the pairs completeness roughly doubled and thus was still far below 0.8.

Choosing the entire title concatenated with the production year as key and the window size 2 resulted in a pairs completeness of more than 0.8 and a precision over 0.02 where the average duration was comparable to using the title prefix and year. Setting the window size to 12 again had the same effect on average duration and precision as before, but increased the pairs completeness only by about 6 percent. We also tried the title alone as key, but the results were essentially the same.

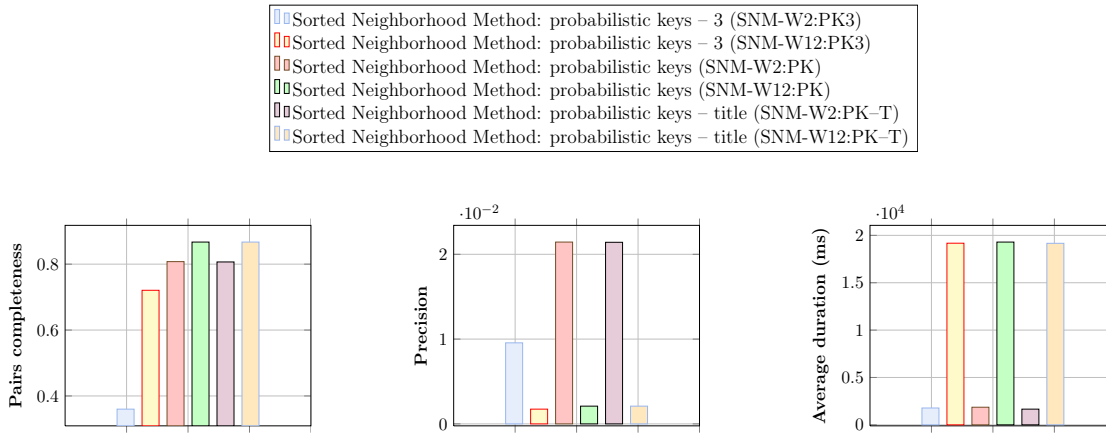


Figure 4.9.: Sorted Neighborhood Method: probabilistic key experiments.

Regarding the probabilistic key approach, we decided to use the same configuration as for the certain key strategies: the minimal window size and a key composed of the entire title and production year. We were able to achieve a pairs completeness of almost 0.81 at a precision of more than 0.02.

4.4.5. Experimental results

For a better evaluation of all the different SSR variants discussed above, we chose the best of every SSR strategy and made a direct comparison. Figure 4.10 illustrates the results, which are presented in more detail in Appendix N.

Regarding the Sorted Neighborhood Method, the best pairs completeness was achieved using all alternatives with the key per alternative strategy at a pretty high precision and a rather low experiment runtime. The same variant was clearly the best for Blocking, because it led to the best pairs completeness, precision and average duration.

In terms of pairs completeness, building highly dissimilar worlds was comparable to using all alternatives for both SSR techniques, but the precision was slightly lower and the average execution time was far worse. Although the key per tuple variants had an even

smaller average duration than using all alternatives, the precision was slightly and the pairs completeness by far smaller than with building highly dissimilar worlds. Finally, the probabilistic key approach did not work well at all in comparison to the other variants; the pairs completeness was the worst in the experiment series for SNM.

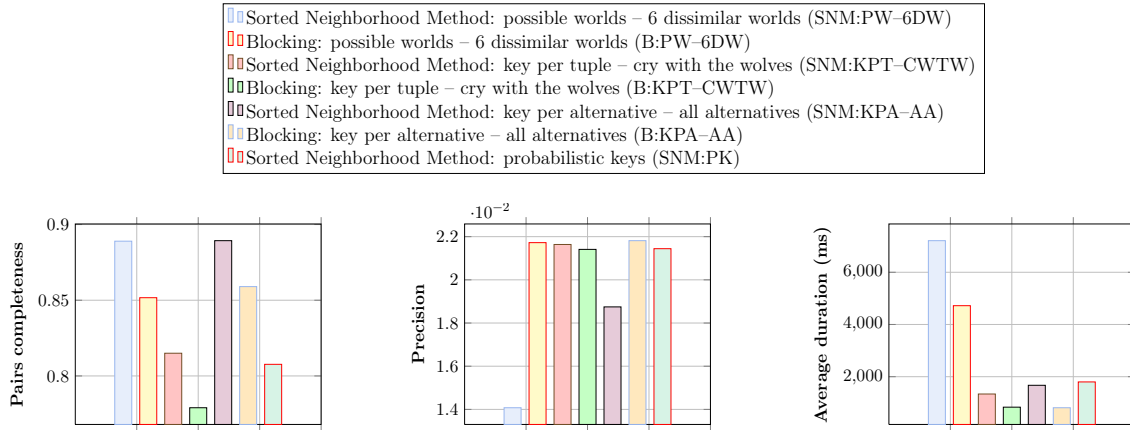


Figure 4.10.: The best adapted SSR variants in comparison.

In summary, using all alternatives is rather fast and results in a very good pairs completeness with an acceptable precision. Furthermore, we conclude that the Sorted Neighborhood Method seemingly outperforms Blocking in terms of pure pairs completeness. So, for single-pass search space reduction, SNM might be the better choice. With Blocking, on the other hand, an astonishingly high precision can be yielded – given that an appropriate key is available. As listed in Table 4.6, Blocking with all alternatives reached over 0.75 pairs completeness with a precision of over 0.98. Hence, an even better result than with the Sorted Neighborhood Method can perhaps be delivered by multi-pass Blocking, when more than one key is known by which a result with high precision can be achieved.

Since we did not do much optimisation of key design or window size, we expect the discussed implementations can all be improved further, for example, by phonetic encoding or other key functions or slightly different window sizes. Hence, our conclusions should only be considered tendencies as to which implementation might achieve the best result.

Comparison with Bigram Indexing

According to [BCC03], Bigram Indexing can lead to significantly improved pairs completeness and precision in comparison to Blocking or the Sorted Neighborhood Method. To verify this statement, we conducted further experiments: Appendix O shows the results of Bigram Indexing SSR in combination with the all alternatives variant. Note that we had to reduce the database size from 200,000 to 50,000 tuple alternatives, because our testing machine simply ran out of memory when executing Bigram Indexing on the large database. We performed the evaluation experiments of Blocking and SNM again on the smaller database (see Appendix P). The results were similar to the results on the

larger database.

We tested several combinations of key design and threshold in our Bigram Indexing experiments and observed a pairs completeness of 0.972 with a precision of more than 0.18 with a twelve-character title prefix as key and a threshold of 0.7. Using a threshold of 0.8 decreased the pairs completeness to less than 0.93, but increased the precision to over 0.30. It should also be mentioned that Bigram Indexing in our experiments took *a few hundred times more time* than any of the Blocking or SNM variants. We assume the actual average duration overhead of Bigram Indexing had been far less extreme, if our testing machine had been equipped with more memory.

Based on these experimental results we assume Bigram Indexing to have much more potential in terms of pairs completeness and precision. But performing Bigram Indexing also appears to require much more memory and seems to be much slower than performing Blocking or the Sorted Neighborhood Method.

5. Summary and future prospects

Duplicate detection is a computationally very expensive task that is a crucial part of the data integration process. Search space reduction techniques are used today to minimise the computational costs of duplicate detection in the process of certain data integration, but there are no SSR techniques available for the integration of probabilistic data yet.

In Chapter 2, we presented traditional probabilistic databases, lineage databases and uncertainty-lineage databases, described the process of certain data integration and went into detail with search space reduction where our focus lay on the SSR techniques Blocking and the Sorted Neighborhood Method.

We investigated how SSR techniques used in certain data could be adapted to the integration of *probabilistic* data in Chapter 3 by the example of the SSR techniques Blocking and the Sorted Neighborhood Method. In detail, we discussed three SSR strategies for probabilistic data using certain key values, namely *multi-pass over possible worlds*, *key per tuple* and *key per alternative*, and covered one SNM variant using probabilistic key values. We generated probabilistic test data from a certain data movie database, implemented our SSR strategies and then tested them on the generated (synthetic) probabilistic data. Chapter 4 covered the description of our testing framework and our test data. Furthermore, the achieved results were compared by the metrics pairs completeness, reduction ratio and precision as well as by their respective runtimes.

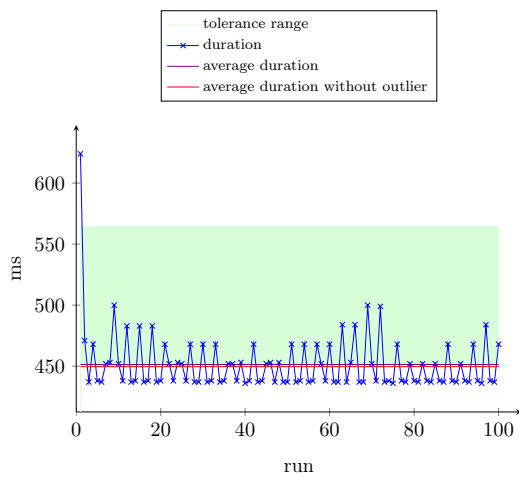
The comparison results indicated that the key per alternative strategy outperforms the other strategies, because it resulted in the best pairs completeness with an acceptable precision and was comparatively fast. Besides, the Sorted Neighborhood Method seems to achieve a higher pairs completeness, whereas a much higher precision seems achievable with Blocking. The best results could be achieved with Bigram Indexing, which appeared to have a longer runtime and much higher memory requirements.

One of the major obstacles we encountered was the total lack of standardised probabilistic benchmarking data. Although we were able to draw some conclusions from our experiments on the generated probabilistic data, it remains to be seen whether similar results can be achieved in a real scenario or what (absolute) results, e.g. what pairs completeness, can be achieved at all. Following studies may consider search space reduction in the context of duplicate detection and investigate to what extent the performance of duplicate detection and the quality of its result are affected. For example, it could be investigated whether the result of the duplicate detection can be improved by extending it to its transitive closure. Furthermore, future work may focus on other SSR techniques for probabilistic data, for example a Blocking variant using probabilistic key values or different multi-pass approaches.

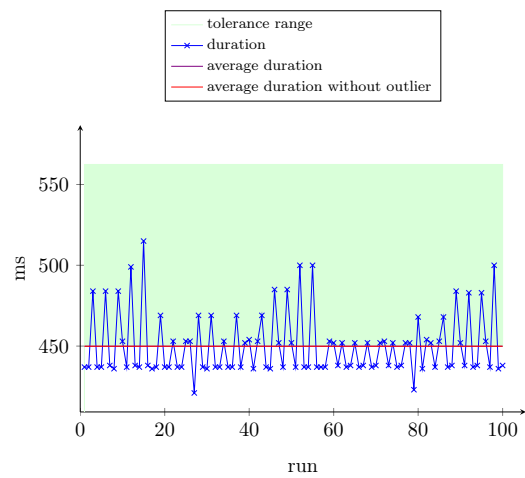
The possibilities of using clustering techniques for search space reduction in probabilistic data have also not been dealt with by us and may be investigated further.

Appendix

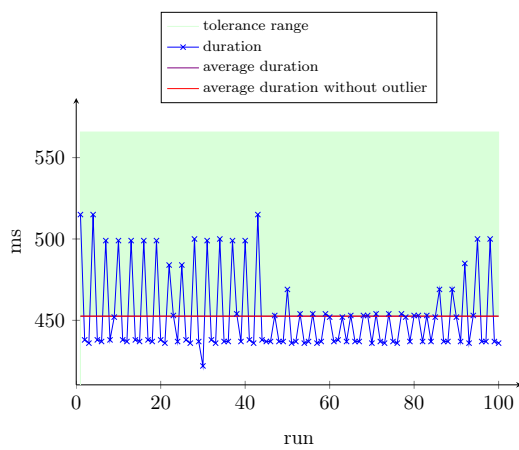
A. Duration experiment series



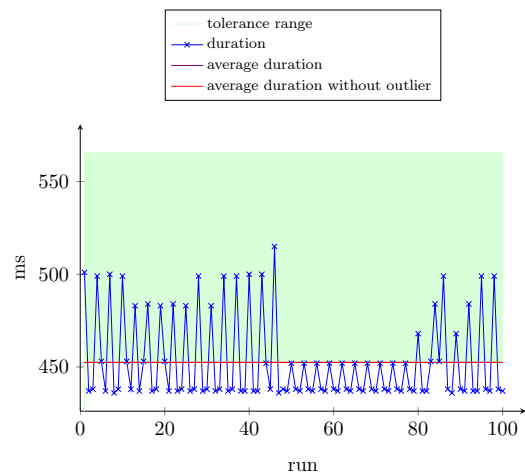
B:KPT-MPA - first experiment



B:KPT-MPA - second experiment

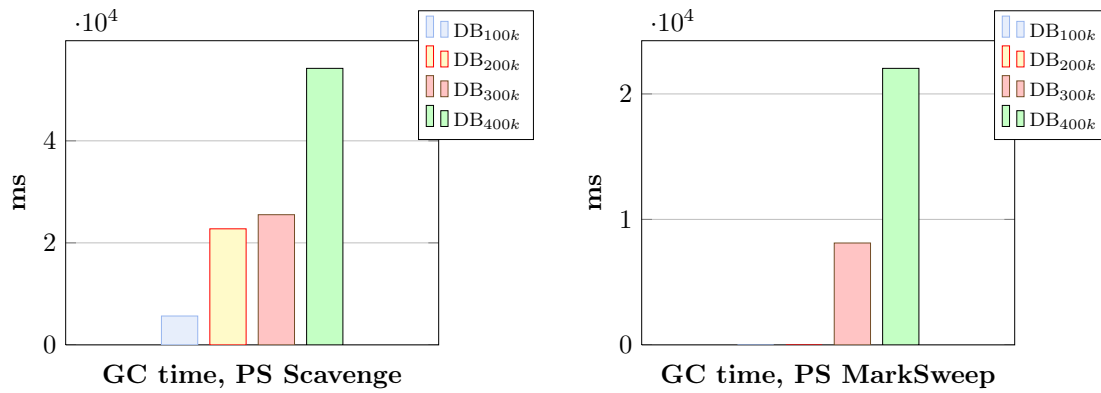


B:KPT-MPA - third experiment



B:KPT-MPA - fourth experiment

B. Database size experiment series



Used SSR variants

Sorted Neighborhood Method: possible worlds – dissimilar worlds (SNM:PW–DW)

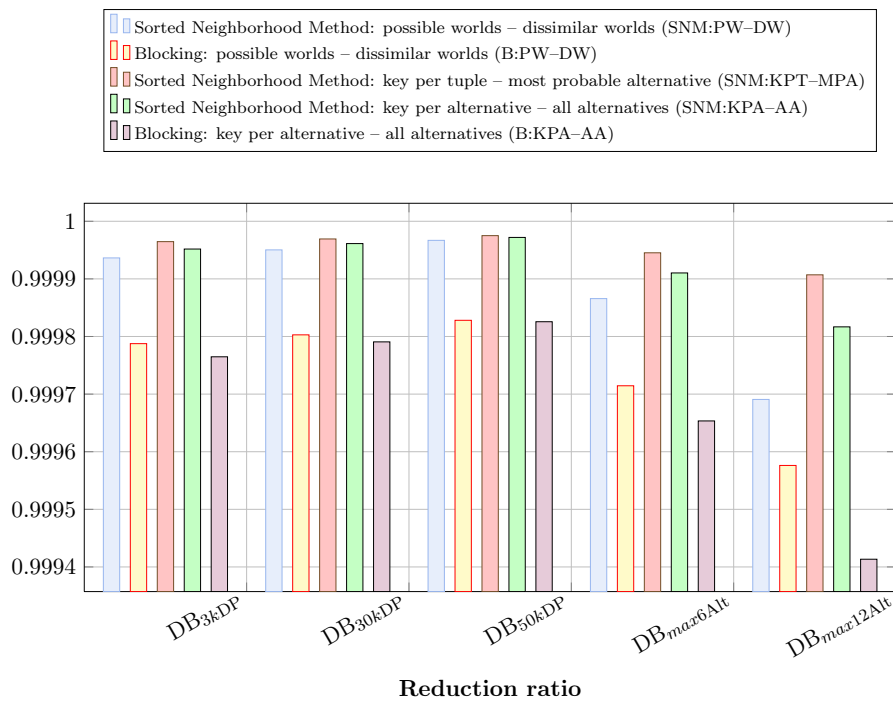
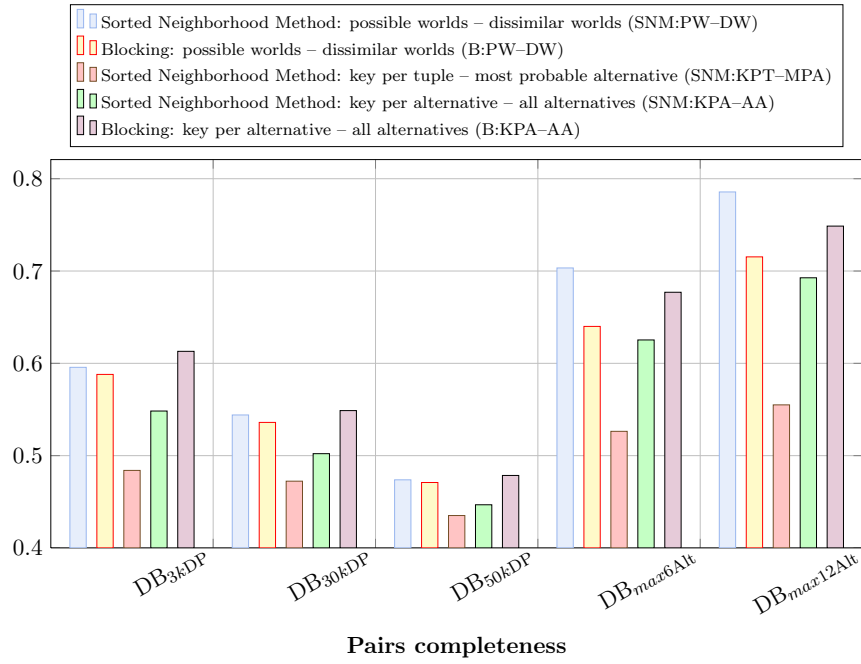
Blocking: possible worlds – dissimilar worlds (B:PW–DW)

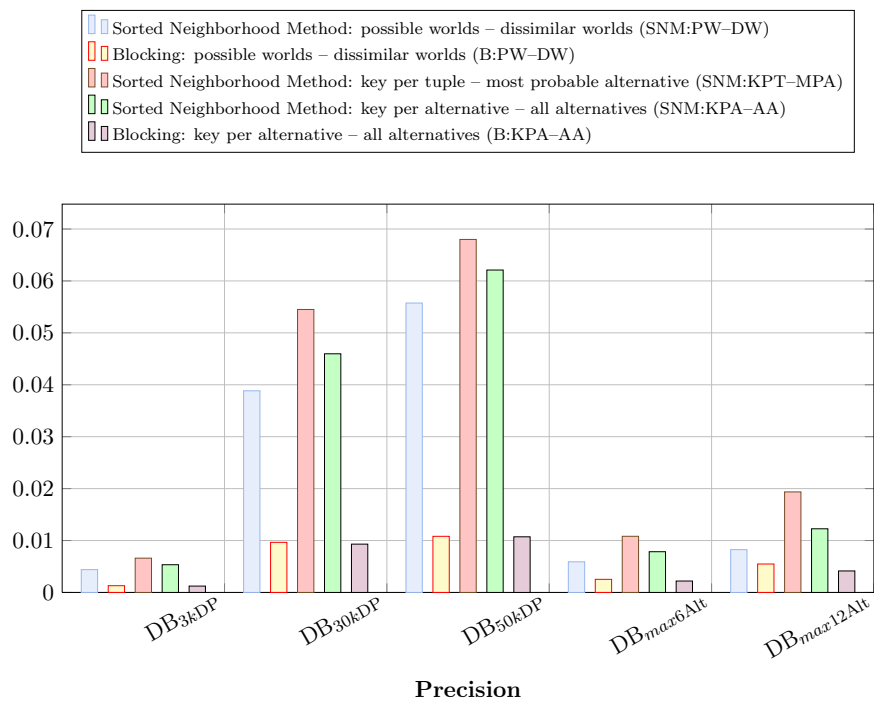
Sorted Neighborhood Method: key per tuple – most probable alternative (SNM:KPT–MPA)

Sorted Neighborhood Method: key per alternative – all alternatives (SNM:KPA–AA)

Blocking: key per alternative – all alternatives (B:KPA–AA)

C. Database configuration experiment series





D. SSR technique configuration experiment series

key function	title:prefix(3), year:attribute	title:prefix(3), year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute	title:attribute
window size	2	12	12	2	2	12
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	150,730	1,516,316	1,486,104	142,315	142,341	1,486,101
rejections	6,384,292,770	6,382,927,184	6,382,957,396	6,384,301,185	6,384,301,159	6,382,957,399
pairs completeness	0.387666666667	0.759333333333	0.924666666667	0.889333333333	0.889	0.924666666667
reduction ratio	0.999976391051	0.999762498329	0.999767230456	0.999977709099	0.999977705026	0.999767230926
precision	0.007715783188	0.001502325373	0.001866625754	0.018747145417	0.018736695681	0.001866629522
average duration (ms)	1,668.66	34,084.96	37,557.56	1,773.7	1,579.7	34,884.73

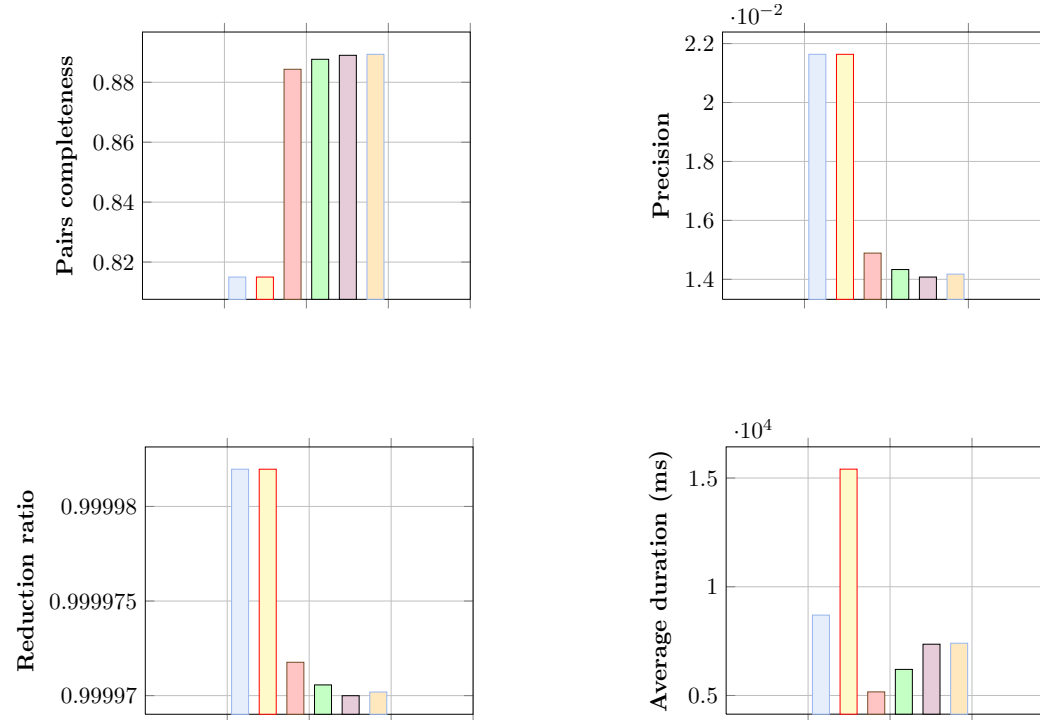
Sorted Neighborhood Method

key function	title:prefix(3), year:attribute	title:prefix(8), year:attribute	title:attribute, year:attribute	title:attribute	title:prefix(6)	title:prefix(8)
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	1,436,262	6,455	1,153	2,292	358,735	118,131
rejections	6,383,007,238	6,384,437,045	6,384,442,347	6,384,441,208	6,384,084,765	6,384,325,369
pairs completeness	0.539	0.465	0.384333333333	0.754333333333	0.889333333333	0.859
reduction ratio	0.999775037245	0.99998988949	0.99999819405	0.99999641002	0.999943811078	0.999981497056
precision	0.00125839157	0.21611541441	1	0.987347294939	0.007437244763	0.021814764964
average duration (ms)	3,303.53	910.03	1,004.13	711.36	1,180.86	857

Blocking

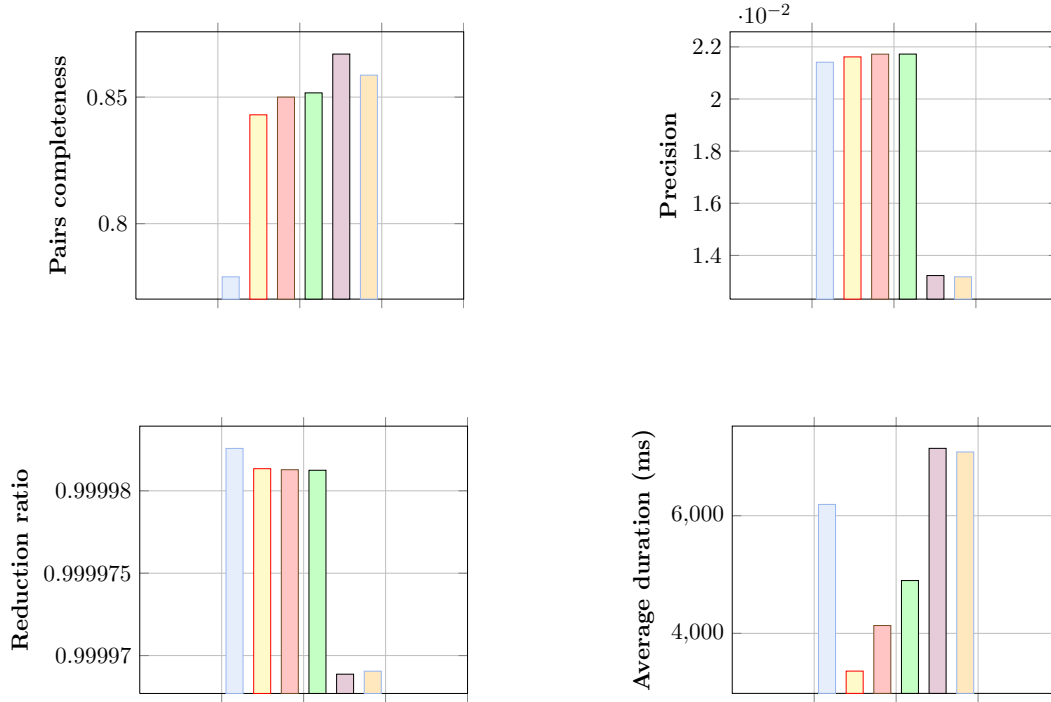
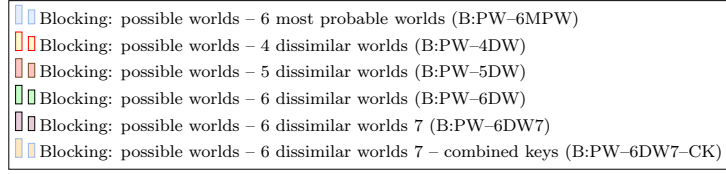
E. SNM: possible worlds experiment series

■	Sorted Neighborhood Method: possible worlds – 6 most probable worlds (SNM:PW-6MPW)
■	Sorted Neighborhood Method: possible worlds – 12 most probable worlds (SNM:PW-12MPW)
■	Sorted Neighborhood Method: possible worlds – 4 dissimilar worlds (SNM:PW-4DW)
■	Sorted Neighborhood Method: possible worlds – 5 dissimilar worlds (SNM:PW-5DW)
■	Sorted Neighborhood Method: possible worlds – 6 dissimilar worlds (SNM:PW-6DW)
■	Sorted Neighborhood Method: possible worlds – 6 dissimilar worlds – combined keys (SNM:PW-6DPW-CK)



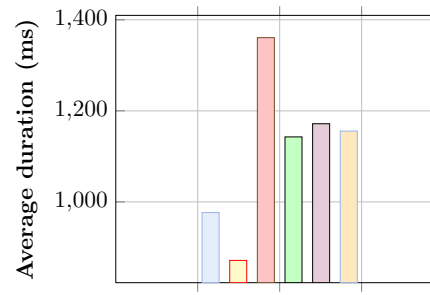
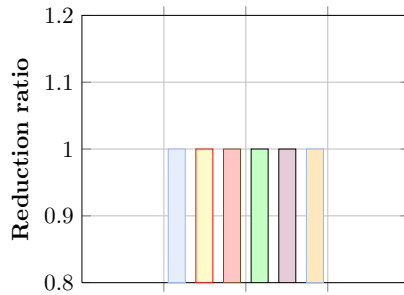
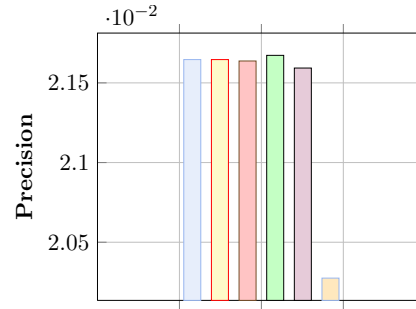
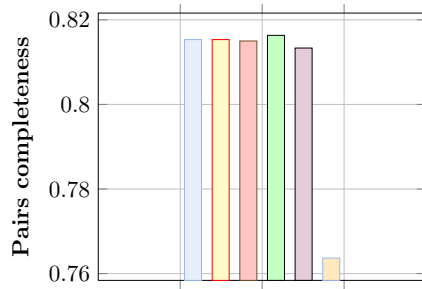
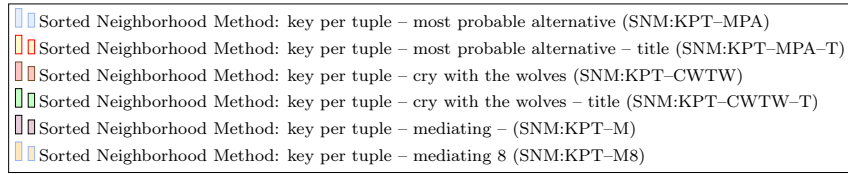
	SNM:PW-6MPW	SNM:PW-12MPW	SNM:PW-4DW	SNM:PW-5DW	SNM:PW-6DW	SNM:PW-6DPW-CK
key function	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute
window size	2	2	2	2	2	2
combined keys	yes	yes	no	no	no	yes
other information	6 worlds	12 worlds	4 worlds	5 worlds	6 worlds	6 worlds
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	113,006	113,009	178,171	185,803	189,454	188,208
rejections	6,384,330,494	6,384,330,491	6,384,265,329	6,384,257,697	6,384,254,046	6,384,255,292
false acceptances	110,561	110,564	175,518	183,140	186,787	185,540
false rejections	555	555	347	337	333	332
true acceptances	2,445	2,445	2,653	2,663	2,667	2,668
true rejections	6,384,329,939	6,384,329,936	6,384,264,982	6,384,257,360	6,384,253,713	6,384,254,960
pairs completeness	0.815	0.815	0.884333333333	0.887666666667	0.889	0.889333333333
reduction ratio	0.999982299789	0.999982299319	0.999972092947	0.999970897542	0.999970325683	0.999970520845
precision	0.021636019326	0.021635444965	0.014890189762	0.0143323843	0.014077295808	0.014175805492
average duration (ms)	8,720.85	15,408.82	5,217.4	6,253.8	7,377.94	7,403.68

F. Blocking: possible worlds experiment series



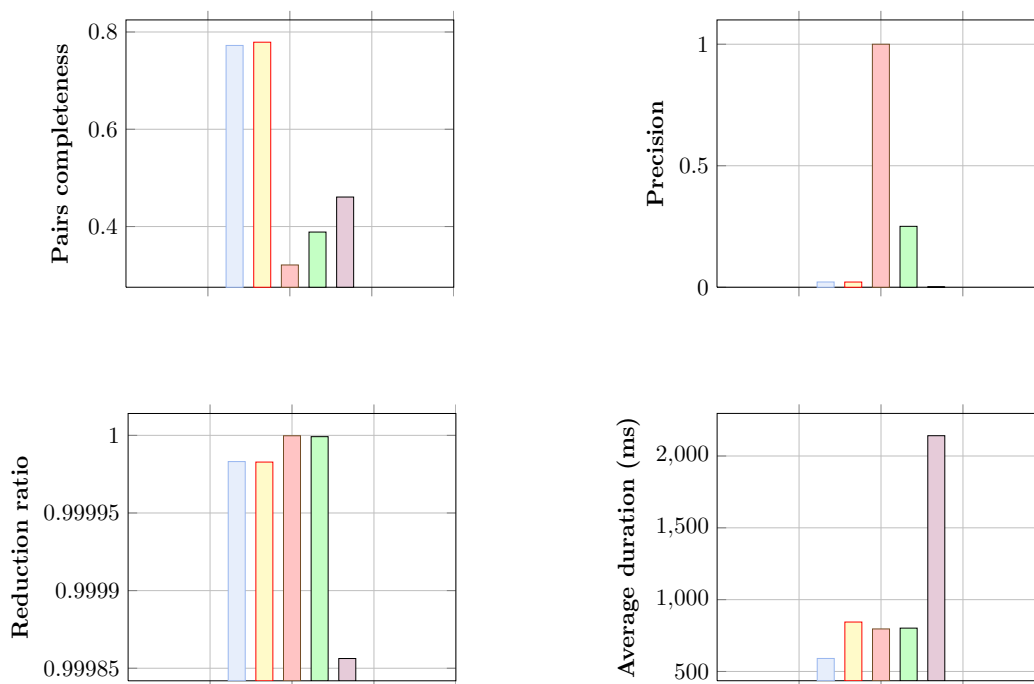
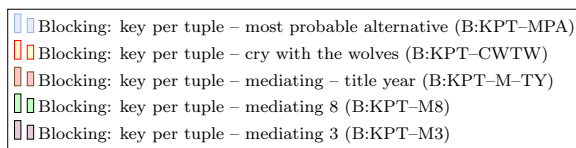
	B:PW-6MPW	B:PW-4DW	B:PW-5DW	B:PW-6DW	B:PW-6DW7	B:PW-6DW7-CK
key function	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(7)	title:prefix(7)
combined keys	no	no	no	no	no	yes
other information	6 worlds	4 worlds	5 worlds	6 worlds	6 worlds	6 worlds
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
actual matches	3,000	3,000	3,000	3,000	3,000	3,000
actual unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	109,151	117,004	117,402	117,617	196,634	195,467
rejections	6,384,334,349	6,384,326,496	6,384,326,098	6,384,325,883	6,384,246,866	6,384,248,033
false acceptances	106,814	114,475	114,852	115,062	194,033	192,891
false rejections	663	471	450	445	399	424
true acceptances	2,337	2,529	2,550	2,555	2,601	2,576
true rejections	6,384,333,686	6,384,326,025	6,384,325,648	6,384,325,438	6,384,246,467	6,384,247,609
pairs completeness	0.779	0.843	0.85	0.851666666667	0.867	0.858666666667
reduction ratio	0.9999829036	0.999981673579	0.99998161124	0.999981577564	0.999969201074	0.999969383863
precision	0.021410706269	0.021614645653	0.021720243267	0.021723050239	0.013227620859	0.013178695125
average duration (ms)	6,326.57	3,358.29	4,176.43	4,921.91	7,252.98	7,145.17

G. SNM: key per tuple experiment series









	SNM:KPT-MPA	SNM:KPT-MPA-T	SNM:KPT-CWTW	SNM:KPT-CWTW-T	SNM:KPT-M	SNM:KPT-M8
key function	title:attribute, year:attribute	title:attribute	title:attribute, year:attribute	title:attribute	title:attribute, year:attribute	title:prefix(8), year:attribute
window size	2	2	2	2	2	2
combined keys	no	no	yes	yes	title:yes	title:yes
other information					title:CWTW, year:EXP	title:CWTW, year:EXP
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
actual matches	3,000	3,000	3,000	3,000	3,000	3,000
actual unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	112,999	112,999	112,999	112,999	112,999	112,997
rejections	6,384,330,501	6,384,330,501	6,384,330,501	6,384,330,501	6,384,330,501	6,384,330,503
false acceptances	110,553	110,553	110,554	110,550	110,559	110,706
false rejections	554	554	555	551	560	709
true acceptances	2,446	2,446	2,445	2,449	2,440	2,291
true rejections	6,384,329,947	6,384,329,947	6,384,329,946	6,384,329,950	6,384,329,941	6,384,329,794
pairs completeness	0.815333333333	0.815333333333	0.815	0.816333333333	0.813333333333	0.763666666667
reduction ratio	0.999982300885	0.999982300885	0.999982300885	0.999982300885	0.999982300885	0.999982301198
precision	0.021646209258	0.021646209258	0.021637359623	0.021672758166	0.02159311443	0.020274874554
average duration (ms)	976.57	871.45	1,360.79	1,142.77	1,171.66	1,155.55

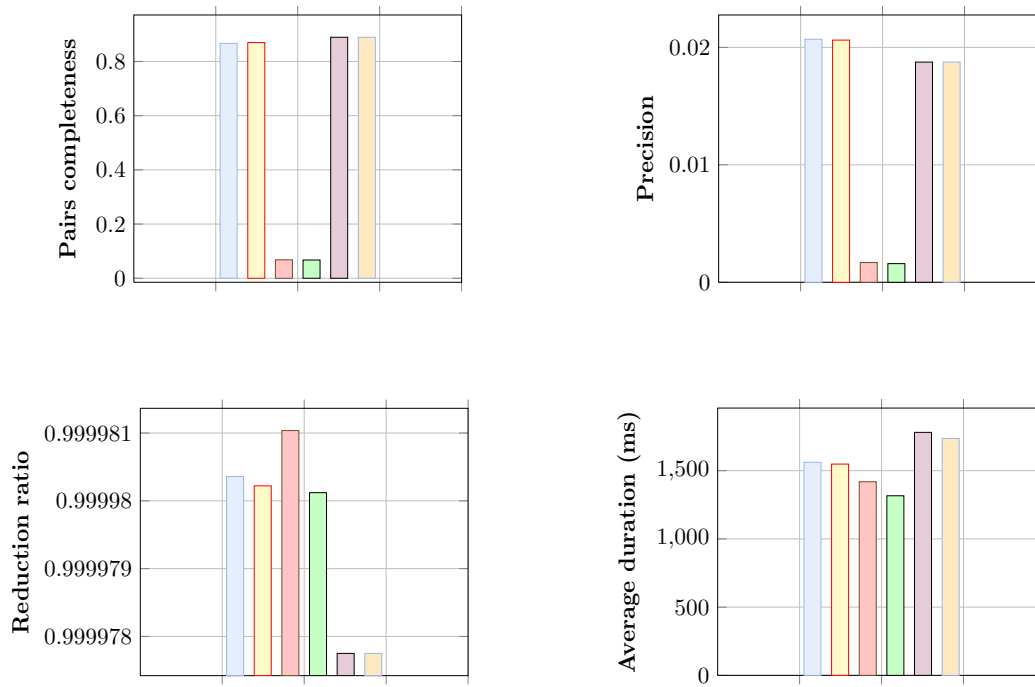
H. Blocking: key per tuple experiment series



	B:KPT-MPA	B:KPT-CWTW	B:KPT-M-TY	B:KPT-M8	B:KPT-M3
key function	title:prefix(8)	title:prefix(8)	title:attribute, year:attribute	title:prefix(8), year:attribute	title:prefix(3), year:attribute
combined keys	no	yes	title:yes	title:yes	title:yes
other information			title:CWTW, year:MEDIAN	title:CWTW, year:MEDIAN	title:CWTW, year:MEDIAN
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	107,124	109,152	963	4,652	916,993
rejections	6,384,336,376	6,384,334,348	6,384,442,537	6,384,438,848	6,383,526,507
false acceptances	104,807	106,815	0	3,486	915,611
false rejections	683	663	2,037	1,834	1,618
true acceptances	2,317	2,337	963	1,166	1,382
true rejections	6,384,335,693	6,384,333,685	6,384,440,500	6,384,437,014	6,383,524,889
pairs completeness	0.772333333333	0.779	0.321	0.388666666667	0.460666666667
reduction ratio	0.99998322109	0.999982903443	0.99999849165	0.99999271354	0.999856370724
precision	0.021629140062	0.021410510114	1	0.250644883921	0.001507099836
average duration (ms)	590.61	844.13	803.39	801.73	2,205.43

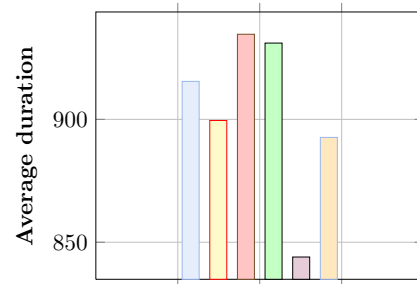
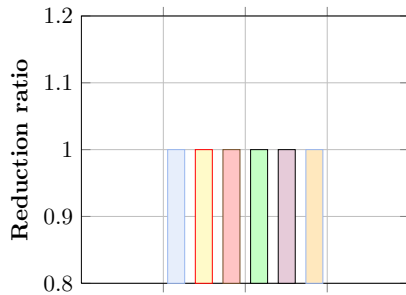
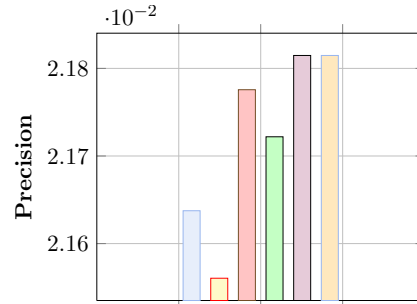
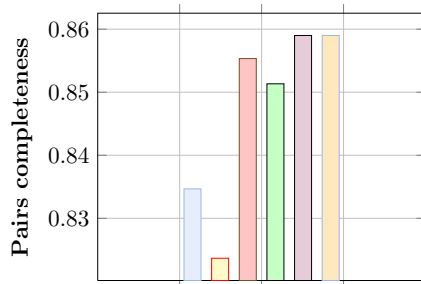
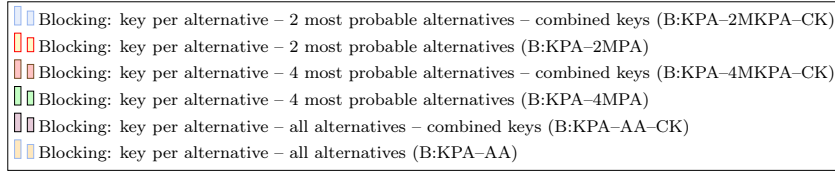
I. SNM: key per alternative – combined keys experiment series

	Sorted Neighborhood Method: key per alternative – 2 most probable alternatives (SNM:KPA-2MPA)
	Sorted Neighborhood Method: key per alternative – 2 most probable alternatives – combined keys (SNM:KPA-2MPA-CK)
	Sorted Neighborhood Method: key per alternative – 2 most probable alternatives 3 (SNM:KPA-2MPA3)
	Sorted Neighborhood Method: key per alternative – 2 most probable alternatives 3 – combined keys (SNM:KPA-2MPA3-CK)
	Sorted Neighborhood Method: key per alternative – all alternatives (SNM:KPA-AA)
	Sorted Neighborhood Method: key per alternative – all alternatives – combined keys (SNM:KPA-AACK)



	SNM:KPA-2MPA	SNM:KPA-2MPA-CK	SNM:KPA-2MPA3	SNM:KPA-2MPA3-CK	SNM:KPA-AA	SNM:KPA-AACK
key function	title:attribute, year:attribute	title:attribute, year:attribute	title:prefix(3)	title:prefix(3)	title:attribute, year:attribute	title:attribute, year:attribute
window size	2	2	2	2	2	2
combined keys	no	yes	no	yes	no	yes
other information	2 alternatives	2 alternatives	2 alternatives	2 alternatives		
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
actual matches	3,000	3,000	3,000	3,000	3,000	3,000
actual unmatched	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	125,660	126,539	121,331	127,178	142,314	142,314
rejections	6,384,317,840	6,384,316,961	6,384,322,169	6,384,316,322	6,384,301,186	6,384,301,186
false acceptances	123,060	123,930	121,127	126,976	139,646	139,646
false rejections	400	391	2,796	2,798	332	332
true acceptances	2,600	2,609	204	202	2,668	2,668
true rejections	6,384,317,440	6,384,316,570	6,384,319,373	6,384,313,524	6,384,300,854	6,384,300,854
pairs completeness	0.866666666667	0.869666666667	0.068	0.067333333333	0.889333333333	0.889333333333
reduction ratio	0.999980317783	0.999980180105	0.999980995838	0.999980080018	0.999977709255	0.999977709255
precision	0.020690752825	0.020618149345	0.001681351015	0.001588325025	0.018747277148	0.018747277148
average duration (ms)	1,561.32	1,554.6	1,418.81	1,316.08	1,864.87	1,834.87

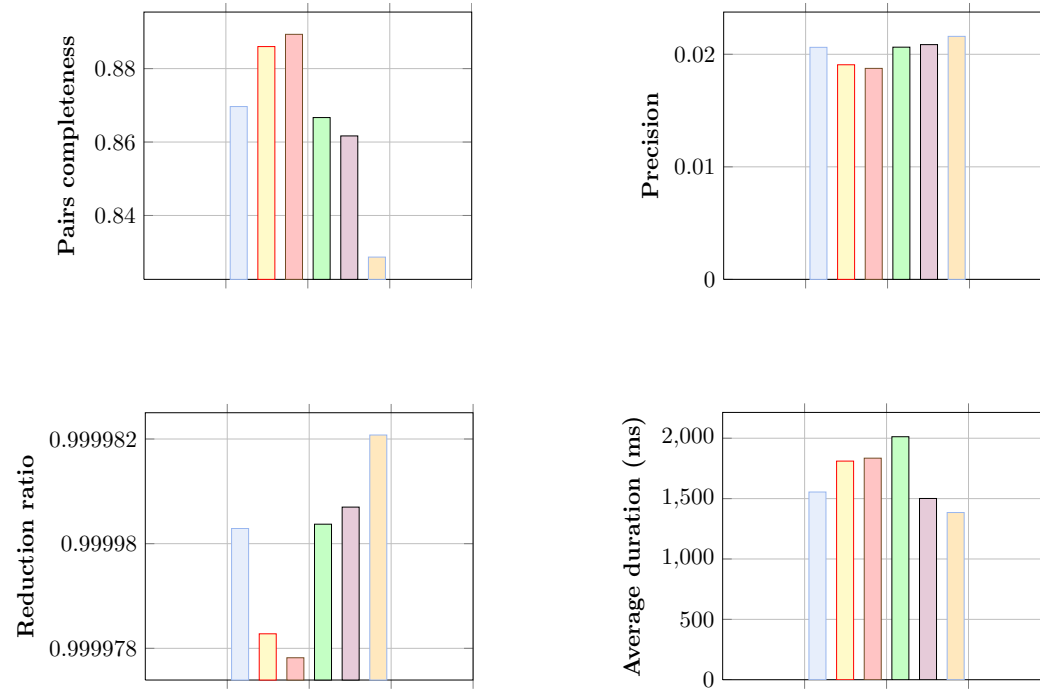
J. Blocking: key per alternative – combined keys experiment series



	B:KPA-2MKPA-CK	B:KPA-2MPA	B:KPA-4MKPA-CK	B:KPA-4MPA	B:KPA-AA-CK	B:KPA-AA
key function	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)
combined keys	yes	no	yes	no	yes	no
other information	2 alternatives	2 alternatives	4 alternatives	4 alternatives		
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	115,725	114,609	117,838	117,578	118,132	118,132
rejections	6,384,327,775	6,384,328,891	6,384,325,662	6,384,325,922	6,384,325,368	6,384,325,368
false acceptances	113,221	112,138	115,272	115,024	115,555	115,555
false rejections	496	529	434	446	423	423
true acceptances	2,504	2,471	2,566	2,554	2,577	2,577
true rejections	6,384,327,279	6,384,328,362	6,384,325,228	6,384,325,476	6,384,324,945	6,384,324,945
pairs completeness	0.834666666667	0.823666666667	0.855333333333	0.851333333333	0.859	0.859
reduction ratio	0.99998187391	0.999982048709	0.999981542949	0.999981583673	0.999981496899	0.999981496899
precision	0.0216375027	0.021560261411	0.021775658107	0.021721750668	0.0218145803	0.0218145803
average duration (ms)	915.45	903.28	934.63	932.29	892.66	843.7







K. SNM: key per alternative experiment series

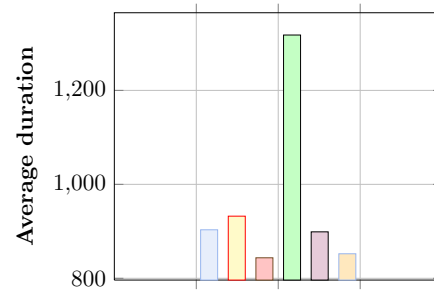
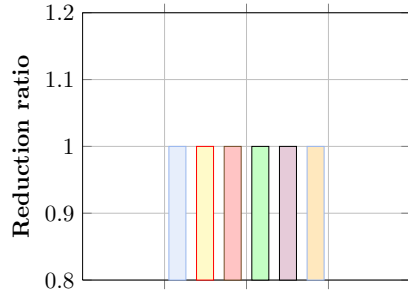
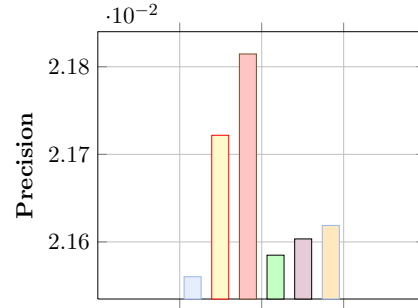
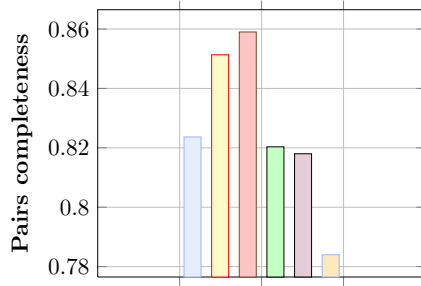
■	Sorted Neighborhood Method: key per alternative – 2 most probable alternatives (SNM:KPA-2MPA)
■	Sorted Neighborhood Method: key per alternative – 4 most probable alternatives (SNM:KPA-4MPA)
■	Sorted Neighborhood Method: key per alternative – all alternatives (SNM:KPA-AA)
■	Sorted Neighborhood Method: key per alternative – most probable and 30,000 remaining alternatives (SNM:KPA-MPA30kRA)
■	Sorted Neighborhood Method: key per alternative – threshold 0.1 (SNM:KPA-T0.1)
■	Sorted Neighborhood Method: key per alternative – threshold 0.3 (SNM:KPA-T0.3)



	SNM:KPA-2MPA	SNM:KPA-4MPA	SNM:KPA-AA	SNM:KPA-MPA30kRA	SNM:KPA-T0.1	SNM:KPA-T0.3
key function	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute, year:attribute
window size	2	2	2	2	2	2
combined keys	yes	yes	yes	yes	yes	yes
other information	2 alternatives	4 alternatives		30,000 remaining alternatives	threshold = 0.1	threshold = 0.3
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	126,539	139,388	142,314	126,012	123,925	115,129
rejections	6,384,316,961	6,384,304,112	6,384,301,186	6,384,317,488	6,384,319,575	6,384,328,371
false acceptances	123,930	136,730	139,646	123,412	121,340	112,643
false rejections	391	342	332	400	415	514
true acceptances	2,609	2,658	2,668	2,600	2,585	2,486
true rejections	6,384,316,570	6,384,303,770	6,384,300,854	6,384,317,088	6,384,319,160	6,384,327,857
pairs completeness	0.869666666667	0.886	0.889333333333	0.866666666667	0.861666666667	0.828666666667
reduction ratio	0.999980180105	0.999978167557	0.999977709255	0.999980262649	0.999980589538	0.999981967262
precision	0.020618149345	0.019069073378	0.018747277148	0.020632955592	0.020859390761	0.021593169401
average duration (ms)	1,554.6	1,810.63	1,834.87	2,041.66	1,501.14	1,384.75

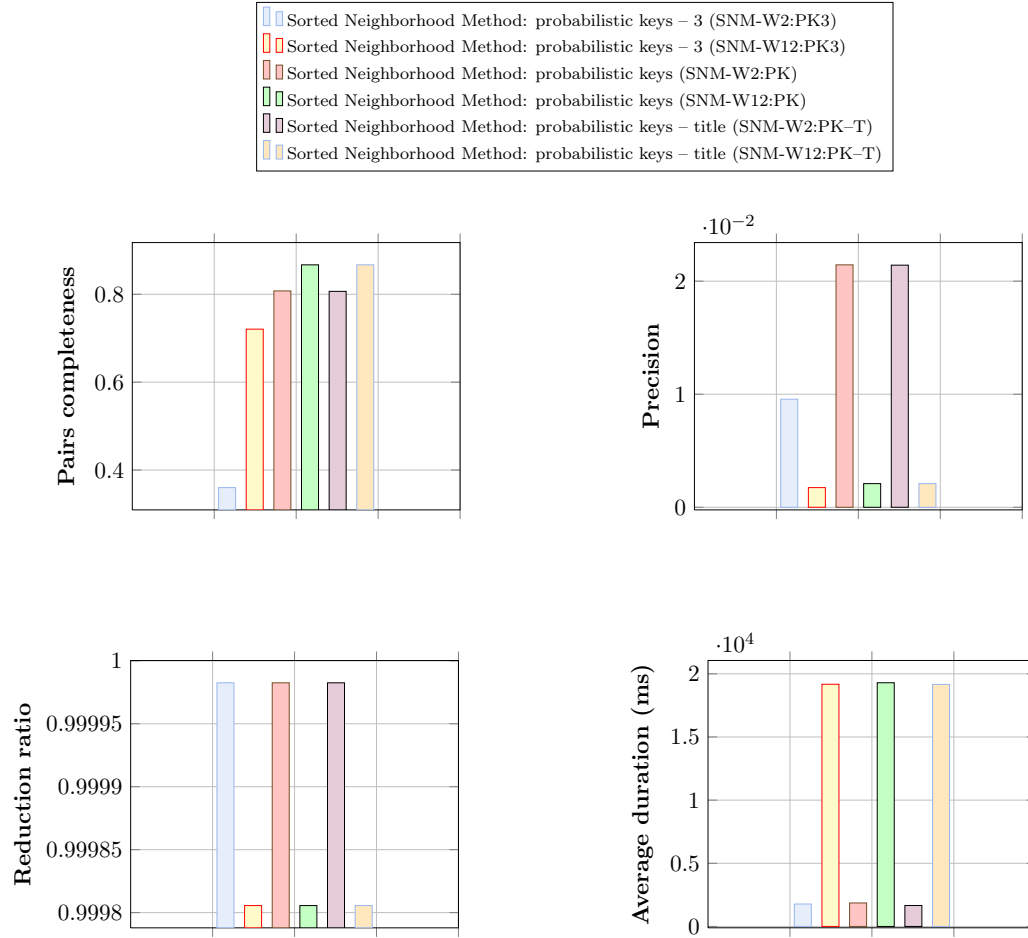
L. Blocking: key per alternative experiment series

	Blocking: key per alternative – 2 most probable alternatives (B:KPA-2MPA)
	Blocking: key per alternative – 4 most probable alternatives (B:KPA-4MPA)
	Blocking: key per alternative – all alternatives (B:KPA-AA)
	Blocking: key per alternative – most probable and 30,000 remaining alternatives (B:KPA-MPA30kRA)
	Blocking: key per alternative – threshold 0.1 (B:KPA-T0.1)
	Blocking: key per alternative – threshold 0.3 (B:KPA-T0.3)



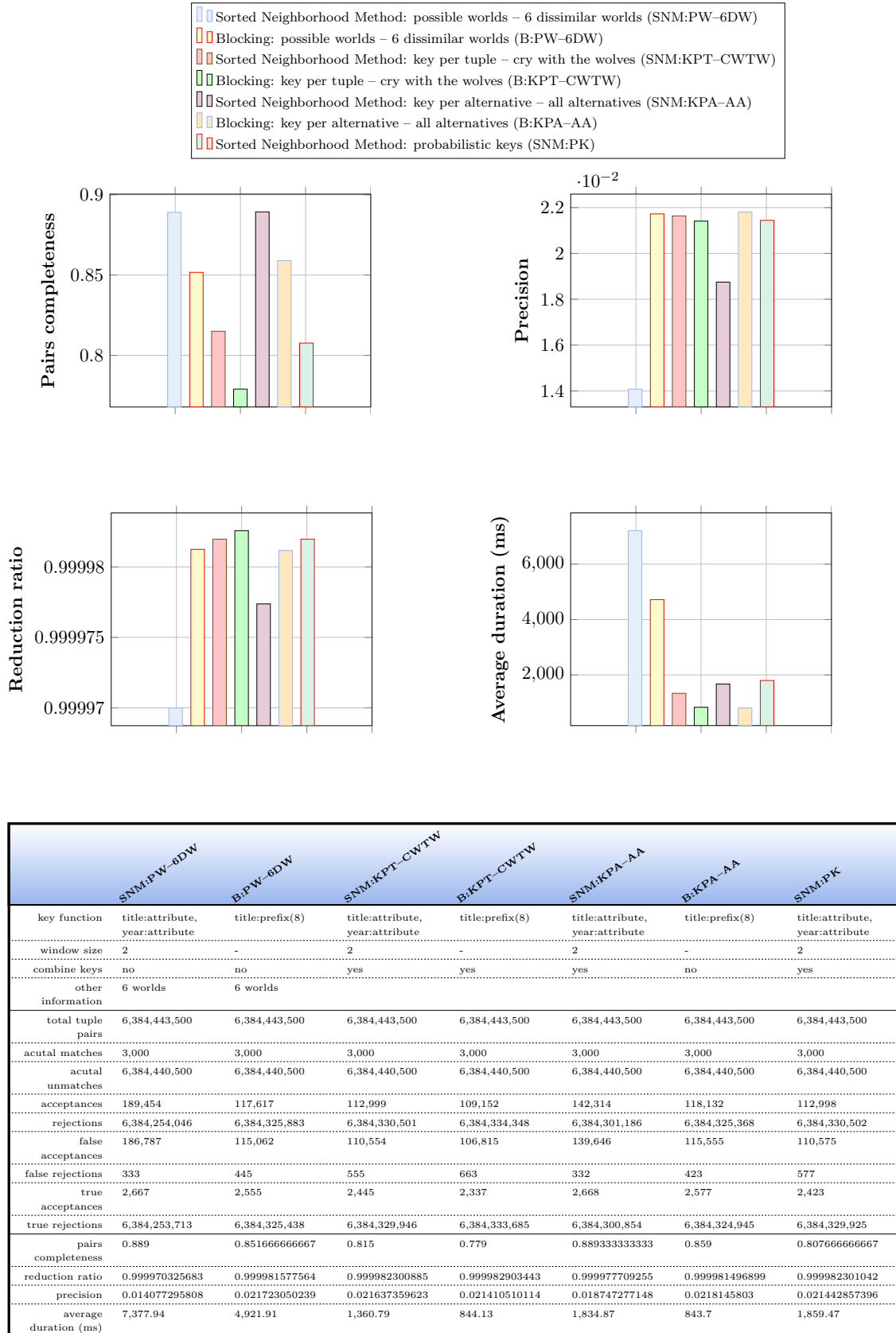
	B:KPA-2MPA	B:KPA-4MPA	B:KPA-AA	B:KPA-MPA30kRA	B:KPA-T0.1	B:KPA-T0.3
key function	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)	title:prefix(8)
combined keys	no	no	no	no	no	no
other information	2 alternatives	4 alternatives		30,000 remaining alternatives	threshold = 0.1	threshold = 0.3
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
actual matches	3,000	3,000	3,000	3,000	3,000	3,000
actual unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	114,609	117,578	118,132	114,015	113,593	108,794
rejections	6,384,328,891	6,384,325,922	6,384,325,368	6,384,329,485	6,384,329,907	6,384,334,706
false acceptances	112,138	115,024	115,555	111,554	111,139	106,442
false rejections	529	446	423	539	546	648
true acceptances	2,471	2,554	2,577	2,461	2,454	2,352
true rejections	6,384,328,362	6,384,325,476	6,384,324,945	6,384,328,946	6,384,329,361	6,384,334,058
pairs completeness	0.823666666667	0.851333333333	0.859	0.820333333333	0.818	0.784
reduction ratio	0.999982048709	0.999981583673	0.999981496899	0.999982141748	0.999982207846	0.999982959517
precision	0.021560261411	0.021721750668	0.0218145803	0.021584879183	0.021603443874	0.021618839274
average duration (ms)	903.28	932.29	843.7	1,321.69	898.91	852.26

M. SNM: probabilistic keys experiment series



	SNM-W2:PK3	SNM-W12:PK3	SNM-W2:PK	SNM-W12:PK	SNM-W2:PK-T	SNM-W12:PK-T
key function	title:prefix(3), year:attribute	title:prefix(3), year:attribute	title:attribute, year:attribute	title:attribute, year:attribute	title:attribute	title:attribute
window size	2	12	2	12	2	12
combined keys	yes	yes	yes	yes	yes	yes
total tuple pairs	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500	6,384,443,500
acutal matches	3,000	3,000	3,000	3,000	3,000	3,000
acutal unmatches	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500	6,384,440,500
acceptances	112,999	1,242,669	112,998	1,242,666	112,998	1,242,665
rejections	6,384,330,501	6,383,200,831	6,384,330,502	6,383,200,834	6,384,330,502	6,383,200,835
false acceptances	111,919	1,240,507	110,575	1,240,065	110,578	1,240,064
false rejections	1,920	838	577	399	580	399
true acceptances	1,080	2,162	2,423	2,601	2,420	2,601
true rejections	6,384,328,581	6,383,199,993	6,384,329,925	6,383,200,435	6,384,329,922	6,383,200,436
pairs completeness	0.36	0.720666666667	0.807666666667	0.867	0.806666666667	0.867
reduction ratio	0.999982300885	0.999805359856	0.999982301042	0.999805360326	0.999982301042	0.999805360483
precision	0.009557606704	0.0017398036	0.021442857396	0.002093080522	0.021416308253	0.002093082206
average duration (ms)	1,777	19,946.67	1,859.47	19,292.57	1,657.53	19,161.43

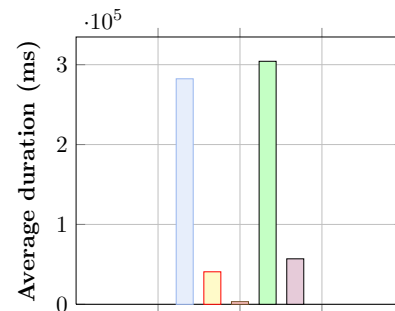
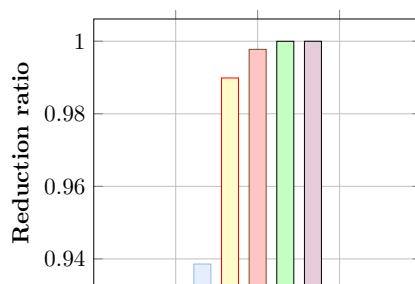
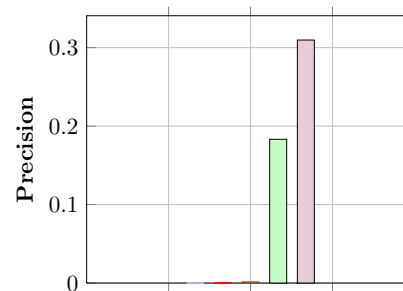
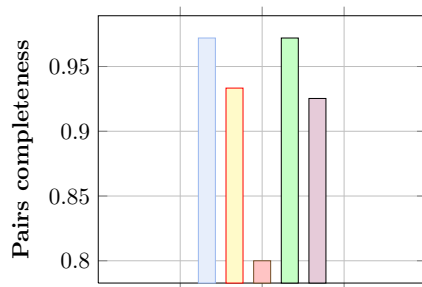
N. Evaluation experiment series



O. Bigram Indexing: all alternatives experiment series

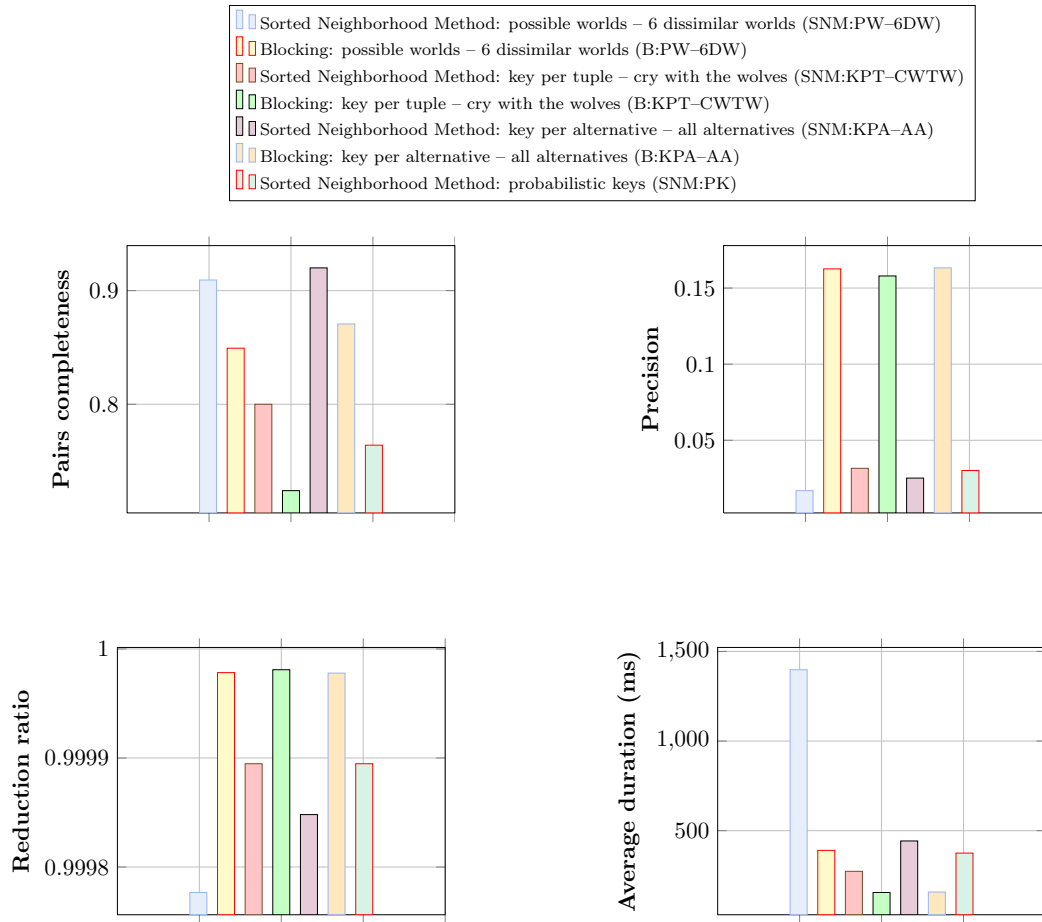
DB _{50k}	
overall tuples	19,000
overall alternatives	50,000
duplicate pairs	750
min. alternatives	1
max. alternatives	6
smallest tuple confidence	0.080618
biggest tuple confidence	1.0
smallest alternative confidence	0.08
biggest alternative confidence	1.0

Bigram Indexing 0.5: key per alternative – all alternatives 3 (BI0.5:KPA-AA3)
Bigram Indexing 0.6: key per alternative – all alternatives 3 (BI0.6:KPA-AA3)
Bigram Indexing 0.8: key per alternative – all alternatives 3 (BI0.8:KPA-AA3)
Bigram Indexing 0.7: key per alternative – all alternatives 12 (BI0.7:KPA-AA12)
Bigram Indexing 0.8: key per alternative – all alternatives 12 (BI0.8:KPA-AA12)



	B10.5:KPA-AA3	B10.6:KPA-AA3	B10.8:KPA-AA3	B10.7:KPA-AA12	B10.8:KPA-AA12
key function	title:prefix(3), year:attribute	title:prefix(3), year:attribute	title:prefix(3), year:attribute	title:prefix(12)	title:prefix(12)
threshold	0.5	0.6	0.8	0.7	0.8
combined keys	no	no	no	no	no
total tuple pairs	180,490,500	180,490,500	180,490,500	180,490,500	180,490,500
acutal matches	750	750	750	750	750
acutal unmatches	180,489,750	180,489,750	180,489,750	180,489,750	180,489,750
acceptances	11,083,419	1,827,637	404,065	3,980	2,241
rejections	169,407,081	178,662,863	180,086,435	180,486,520	180,488,259
false acceptances	11,082,690	1,826,937	403,465	3,251	1,547
false rejections	21	50	150	21	56
true acceptances	729	700	600	729	694
true rejections	169,407,060	178,662,813	180,086,285	180,486,499	180,488,203
pairs completeness	0.972	0.933333333333	0.8	0.972	0.925333333333
reduction ratio	0.938592784662	0.989874054313	0.997761294916	0.999977948978	0.999987583834
precision	0.000065773928	0.000383008223	0.001484909606	0.183165829146	0.309683177153
average duration (ms)	306,271.33	62,703.73	3,365.73	362,788	127,042.87

P. Evaluation experiment series 50k



	SNM:PW-6DW	B:PW-6DW	SNM:KPT-CWTW	B:KPT-CWTW	SNM:KPA-AA	B:KPA-AA	SNM:PK
key function	title:attribute, year:attribute	title:prefix(8)	title:attribute, year:attribute	title:prefix(8)	title:attribute, year:attribute	title:prefix(8)	title:attribute, year:attribute
window size	2	-	2	-	2	-	2
combined keys	no	no	yes	yes	yes	no	yes
other information	6 worlds	6 worlds					
total tuple pairs	180,490,500	180,490,500	180,490,500	180,490,500	180,490,500	180,490,500	180,490,500
acutal matches	750	750	750	750	750	750	750
acutal unmatches	180,489,750	180,489,750	180,489,750	180,489,750	180,489,750	180,489,750	180,489,750
acceptances	40,312	3,918	18,999	3,438	27,417	4,000	18,999
rejections	180,450,188	180,486,582	180,471,501	180,487,062	180,463,083	180,486,500	180,471,501
false acceptances	39,630	3,281	18,399	2,895	26,727	3,347	18,426
false rejections	68	113	150	207	60	97	177
true acceptances	682	637	600	543	690	653	573
true rejections	180,450,120	180,486,469	180,471,351	180,486,855	180,463,023	180,486,403	180,471,324
completeness	0.909333333333	0.849333333333	0.8	0.724	0.92	0.870666666667	0.764
reduction ratio	0.999776653065	0.999978292486	0.999894736842	0.999980951906	0.999848097268	0.999977838169	0.999894736842
precision	0.016918039294	0.162582950485	0.031580609506	0.157940663176	0.025166867272	0.16325	0.030159482078
average duration (ms)	1,400.85	390.03	273.35	155.21	442.64	157.67	376.36

Bibliography

- [BCC03] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage, 2003.
- [Bil06] Mikhail Bilenko. Adaptive blocking: Learning to scale up record linkage. In *In Proceedings of the 6th IEEE International Conference on Data Mining (ICDM-2006)*, pages 87–96, 2006.
- [BN08] Jens Bleiholder and Felix Naumann. Data fusion. *ACM Comput. Surv.*, 41(1):1–41, 2008.
- [BS06] Carlo Batini and Monica Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006.
- [BSH⁺08] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.
- [CGL01] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Description logics for information integration. In *In Computational Logic: From Logic Programming into the Future (In honour of Bob Kowalski), Lecture Notes in Computer Science*, pages 41–60. Springer, 2001.
- [Chr08a] Peter Christen. *Febrl - Freely Extensible Biomedical Record Linkage Manual*. The Australian National University, 0.4.1 edition, 2008.
- [Chr08b] Peter Christen. Febrl – an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1065–1068, New York, NY, USA, 2008. ACM.
- [DJ03] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, NY, USA, 2003.
- [dK08] Ander de Keijzer. *Management of Uncertain Data - towards unattended integration*. PhD thesis, University of Twente, Enschede, February 2008.
- [dKvK07] A. de Keijzer and M. van Keulen. User feedback in probabilistic integration. In *Second International Workshop on Flexible Database and Information System Technology (FlexDBIST 2007), Regensburg, Germany*, pages 377–381, Los Alamitos, September 2007. IEEE Computer Society Press.

- [DS04] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases, 2004.
- [dVKCC09] Timothy de Vries, Hui Ke, Sanjay Chawla, and Peter Christen. Robust record linkage blocking using suffix arrays. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 305–314, New York, NY, USA, 2009. ACM.
- [EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, 2007.
- [FHH⁺09] Ronald Fagin, Laura M. Haas, Mauricio A. Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236, 2009.
- [FS69] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64:1183–1210, 1969.
- [HS95] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 127–138, New York, NY, USA, 1995. ACM.
- [HS98] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2:9–37, 1998.
- [Jar89] Matthew A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84:414–420, 1989.
- [LF05] Patrick Lehti and Peter Fankhauser. A precise blocking method for record linkage. In *DaWaK*, pages 210–220, 2005.
- [LN06] Ulf Leser and Felix Naumann. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt, 2006.
- [ME97] Alvaro Monge and Charles Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *SIGMOD workshop on data mining and knowledge discovery*, May 1997.
- [MF03] Heiko Müller and Johann-Christoph Freytag. Problems, methods, and challenges in comprehensive data cleansing. Technical Report 164, Humboldt University Berlin, 2003.
- [MM08] Matteo Magnani and Danilo Montesi. Uncertainty in data integration: current approaches and open problems, 2008.

-
- [MNU00] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, New York, NY, USA, 2000. ACM.
- [New67] Howard B. Newcombe. Record linking: The design of efficient systems for linking records into individual and family histories, 1967.
- [NK62] Howard B. Newcombe and James M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, 1962.
- [NKAJ59] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130:954–959, October 1959.
- [PvKdKR09] Fabian Panse, Maurice van Keulen, Ander de Keijzer, and Norbert Ritter. Duplicate detection in probabilistic data. In *Proceedings of the 2nd International Workshop on New Trends in Information Integration (NTII 2010)*, number TR-CTIT-09-44 in CTIT technical report series, Enschede, December 2009. Centre for Telematics and Information Technology, University of Twente. Extended version of NTII2010 workshop paper.
- [RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.
- [STW08] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Data modifications and versioning in trio. Technical Report 2008-5, Stanford InfoLab, 2008.
- [vKdK09] Maurice van Keulen and Ander de Keijzer. Qualitative effects of knowledge rules and user feedback in probabilistic data integration. *The VLDB Journal*, July 2009.
- [vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.
- [Wid08] Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. Springer, 2008.
- [WM89] Y. Richard Wang and Stuart E. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 46–55, Washington, DC, USA, 1989. IEEE Computer Society.
- [YLyKG07] Su Yan, Dongwon Lee, Min yen Kan, and C. Lee Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *INTERNATIONAL CONFERENCE ON DIGITAL LIBRARIES*, pages 185–194. ACM, 2007.

List of Figures

2.1. A traditional probabilistic database (PDB)	12
2.2. A lineage database (LDB)	14
2.3. An uncertainty-lineage database	15
2.4. Data integration in four steps	17
2.5. Schema management	18
2.6. Search space reduction during duplicate detection	21
2.7. The trade-off between PC and PR	23
2.8. Blocking	25
2.9. The Sorted Neighborhood Method	27
2.10. Q -gram Indexing	30
3.1. Multi-pass over possible worlds	32
3.2. Most probable worlds with (normalised) confidence values	33
3.3. Building highly dissimilar possible worlds	35
3.4. Key per tuple strategy applied to SNM	35
3.5. Combining variants of the key per tuple strategy	37
3.6. Key per alternative strategy applied to SNM	38
3.7. Combining key values	39
3.8. Probabilistic key approach applied to SNM	40
4.1. Computing the average duration of an experiment	42
4.2. Generating probabilistic test data	43
4.3. SNM: possible worlds experiment series	49
4.4. Blocking: possible worlds experiment series	50
4.5. SNM: key per tuple experiment series	51
4.6. Blocking: key per tuple experiment series	51
4.7. SNM: key per alternative experiment series	52
4.8. Blocking: key per alternative experiment series	53
4.9. SNM: probabilistic key experiments	54
4.10. The best adapted SSR variants in comparison	55

List of Tables

4.1. The average duration of a repeated experiment	42
4.2. Databases of different size	45
4.3. Database configuration experiment series – databases	46
4.4. The database for the SSR experiments	47
4.5. SNM: configuration experiment series	47
4.6. Blocking: configuration experiment series	48

Statutory Declaration

We hereby declare that we have developed and written the enclosed thesis entirely by ourselves and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such. Individual responsibilities for certain sections are as follows:

<i>chapter</i>	<i>Steffen Friedrich</i>	<i>Wolfram Wingerath</i>
1	1.2, 1.4	1–1.1, 1.3
2	2.2, 2.3.1, 2.3.2, 2.3.4	2–2.1, 2.3, 2.3.3
3	3–3.1, 3.1.2, 3.1.4	3.1.1, 3.1.3, 3.2
4	4.3, 4.4.2, 4.4.4, 4.4.5	4–4.2, 4.4–4.4.1, 4.4.3
5	5	

This thesis was not submitted in the same or in a substantially similar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere.

We agree that a copy of this thesis may be made available in the Informatics Library of the University of Hamburg.

Steffen Friedrich

Wolfram Wingerath

Hamburg, November 25th, 2010