Duplicate-based Schema Matching

von Diplom - Informatiker

Alexander Bilke
aus Berlin

von der Fakultät IV – Elektotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:
Vorsitzende: Prof. Dr. Sabine Glesner
Berichter: Prof. Dr. Bernd Mahr
Berichter: Prof. Dr. Herbert Weber
Berichter: Prof. Dr. Felix Naumann
Berichter: Prof. Dr. Erhard Rahm

Tag der wissenschaftlichen Aussprache: 20. 12. 2006

Berlin 2007
D 83

Zusammenfassung

Die Integration unabhängig voneinander entwickelter Datenquellen stellt uns vor viele Probleme, die das Ergebnis verschiedener Arten von Heterogenität sind. Eine der größten Herausforderungen ist Schema Matching: der halbautomatische Prozess, in dem semantische Beziehungen zwischen Attributen in heterogenen Schemata erkannt werden.

Verschiedene Lösungen, die Schemainformationen ausnutzen oder spezifische Eigenschaften aus Attributwerten extrahieren, wurden in der Literatur beschrieben. In dieser Dissertation wird ein neuartiger Schema-Matching-Algorithmus vorgestellt, welcher 'unscharfe' Duplikate, also unterschiedliche Repräsentationen der gleichen Realwelt-Entität, ausnutzt.

In dieser Arbeit wird der *DUMAS table matcher*, welcher Attributkorrespondenzen zwischen zwei Tabellen herstellt, beschrieben. Das Auffinden der Duplikate, die dann für das Schema Matching benutzt werden können, ist eine herausfordernde Aufgabe, weil die semantischen Beziehungen zwischen den Tabellen nicht bekannt sind und somit bekannte Duplikaterkennungsverfahren nicht angewandt werden können. Das neue Problem der Duplikaterkennung zwischen nicht angeglichenen Tabellen und ein Algorithmus, der die Top-k Duplikate findet, wird beschrieben. Die Attributkorrespondenzen zwischen den beiden Tabellen werden in einem folgenden Schritt aus den Duplikaten extrahiert.

Der *DUMAS schema matcher* erweitert den duplikat-basierten Matching-Ansatz auf komplexe Schemata, welche aus mehreren Tabellen bestehen. Das Auffinden von Korrespondenzen zwischen komplexen Schemata wirft neue Probleme auf, die bei einzelnen Tabellen nicht auftreten. Somit ist die direkte Anwendung des DUMAS table matcher nicht möglich. Stattdessen werden Heuristiken benutzt, mit deren Hilfe entschieden werden kann, ob einem Matching zwischen zwei Tabellen vertraut werden kann. Basierend darauf wird ein Algorithmus entwickelt, der Attributkorrespondenzen zwischen komplexen Schemata findet.

Die beiden bisher beschriebenen Algorithmen sind auf einfache (1:1) Korrespondenzen beschränkt. Weil komplexe (1:n oder m:n) Korrespondenzen in der Praxis vorkommen, wurde der *DUMAS complex matcher* entwickelt. Dieser Matcher benutzt das Ergebnis des DUMAS table matcher und verbessert das Ergebnis, indem einzelne Attribute kombiniert werden. Auf diese Weise werden komplexe Korrespondenzen gebildet. Weil der Raum der möglichen komplexen Matchings sehr groß ist, wurden Heuristiken entwickelt, mit deren Hilfe die Anzahl der zu betrachtenden Attributkombinationen eingeschränkt werden.

Abstract

The integration of independently developed data sources poses many problems, which are the result of several types of heterogeneity. One of the most daunting challenges is schema matching, which is the semi-automatic process of detecting semantic relationships between attributes in heterogeneous schemata.

Various solutions that exploit schema information or extract specific features from attribute values have been described. In this thesis we propose novel schema matching algorithms that exploit fuzzy duplicates, i.e., different representations of the same real-world entity.

We describe the *DUMAS table matcher*, whose goal is to establish attribute correspondences between two tables. Finding the duplicates that can be used for schema matching is a challenging task because the semantic relationships between the tables are unknown, and thus, existing duplicate detection solutions cannot be applied. We discuss the novel problem of duplicate detection in unaligned relations and describe an algorithm that is able to detect the top-k duplicates. The attribute correspondences between the two tables are extracted from those duplicates in a subsequent step.

The *DUMAS schema matcher* extends the duplicate-based matching approach to complex schemata consisting of multiple tables. Finding attribute correspondences between complex schemata poses several new challenges that do not occur when single tables are to be matched, and thus, complicate the application of the table matcher. We describe heuristics used to determine if a table matching can be trusted, and develop an algorithm that exploits multi-table duplicates to detect correspondences between complex schemata.

The previous two algorithms are restricted to simple (i.e, 1:1) correspondences. Because complex (i.e., 1:n or m:n) do occur in practice, we developed the *DUMAS complex matcher*. The matcher uses the result of the DUMAS table matcher and improves the matching by merging certain attributes, and thus, detecting complex correspondences. Because the space of possible complex matchings is very large, we devised several heuristics to decrease the number of attribute combinations that have to be considered.

Acknowledgements

First of all I thank my supervisors for their constant support and advice during my dissertation years. Prof. Herbert Weber (TU Berlin & Fraunhofer ISST) gave me the opportunity to carry out out my PhD work in his department. Felix Naumann (Hasso-Plattner-Institute Potsdam) has influenced my work throughout the years by providing valuable ideas in many interesting discussions.

I would also like to thank the Computer-based Information Systems group at Technical University Berlin. Dr. Ralf-Detlef Kutsche has introduced me to the field of data integration. He also gave me the opportunity to hold a seminar on that topic. I very much enjoyed the interesting discussions with Dr. Susanne Busse, Thomas Kabisch, and Dr. Alexander Löser in our Journal Club. Martin Konitzer and Dennis Dietrich supported my work by developing new ideas on duplicate-based schema matching in their diploma theses.

The idea for our Journal Club I was happy to 'import' from the Information Integration Group at Hasso-Plattner-Institute Potsdam. I thank Melanie Weis, Jens Bleiholder, and their colleagues for many interesting ideas on data integration and schema matching. I very much enjoyed our development of the Humboldt Merger.

The Berlin-Brandenburg Graduate School on Distributed Information Systems has supported my work by providing the financial basis and valuable feedback in various seminars. I particularly thank those graduate students who made those three years very interesting not only in research. I also thank Prof. AnHai Doan for his invitation to his group at University of Illinois Urbana-Champaign. Finally, I thank Prof. Erhard Rahm and Prof. Bernd Mahr for their evaluation of my thesis.

Finally, I thank my parents, who have always helped me by all means. The constant support of my family has given me the motivation to begin and finish my PhD work.

Contents

List of Figures

Part I

Mapping Overlapping Databases

Chapter 1

Putting Together Pieces of Information

When autonomously developed data sources are to be integrated, one inevitably has to deal with heterogeneity. While standard solutions for the resolution of conflicts arising from technical and data model heterogeneity exist, structural and semantic heterogeneity is still an open issue despite several decades of research. One of the most daunting challenges in data integration is *schema matching*, which is the semi-automatic process of detecting correspondences between semantically related attributes. This thesis describes a novel schema matching algorithm that exploits extensional overlap (i.e., duplicates) between data sources.

Before discussing the schema matching problem and our duplicate-based solution, this chapter provides a general introduction to data integration. We show that *schema mappings* are an integral part in most integrated information systems. The creation of such mappings is guided by attribute correspondences, which are usually manually established. Schema matching algorithms extract correspondences in a semi-automatic process, and thus, reduce the cost of developing an integrated information system.

1.1 Introduction

Many companies gather a large amount of data about customers, products, sales, suppliers, etc. Much of those data is stored in relational databases, which provide a uniform interface (or schema) that is used by multiple software application to access the information. Unless standard off-the-shelf products are used, the database schemata can be designed to suit the specific requirements of the company: The schemata can closely reflect the view of the company on the business domain and be optimized with respect to their applications.

While the liberty the database designers have may benefit each individual company, it creates additional problems when databases need to be integrated:

Assume a merger of two companies with the goal of generating positive synergy effects. Although some departments might be unaffected, the company can benefit from the merging of many operative areas, e.g., customer relationship or procurement. This process also involves the integration of existing databases, which is known to be an expensive process due to various forms of heterogeneity, which are the result of autonomous development of the data sources.

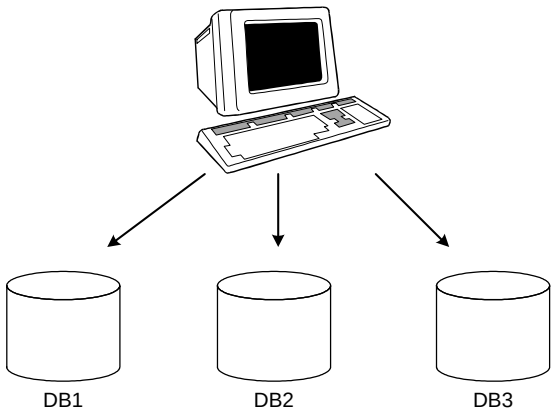


Figure 1.1: Single application using many data sources.

One way to bring together the information from the databases is to write new software that uses the existing data sources (Fig. 1.1). That application would query each data source individually and combine the retrieved results. While this solution is technically feasible, it is not desirable because in most IT infrastructures several software applications use the databases. If another application that requires data from all databases has to be developed, a lot of the integration work (i.e., the resolution of heterogeneity) has to be redone. In addition, inconsistencies may arise if the applications are run concurrently. To avoid those problems, an *integrated information system* (IIS) is placed between the applications and existing data sources (Fig. 1.2). An integrated information system presents a uniform interface, which can be used by several applications to access the sources. The creation of such applications is not as complex as in the previous case, because only the IIS and not all sources need to be understood by the developers. In addition, the IIS can provide regular DBMS services, e.g., concurrency control.

In the following we discuss different forms of heterogeneity that can be observed in such scenarios. We describe different ways of integrating information sources, and show that schema matching is an important step in the development of such systems.

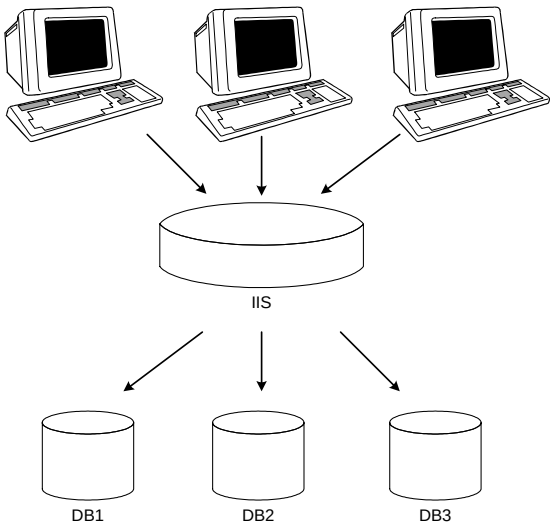


Figure 1.2: An integrated information system.

1.2 Heterogeneity and Conflicts in Data Integration

Data integration has been a research area in computer science for several decades under various names: multidatabase systems [LMR90], federated database systems [SL90], mediator-based systems [Wie92], data warehouses [CD97], and (more recently) peer database management systems [HIST03]. While different types of integrated information systems differ with respect to their level of coupling, materialization, etc., all of them have to deal with *heterogeneity*. Heterogeneity arises when data sources are autonomously developed. In order to avoid heterogeneity, widely accepted standards must be followed. However, in most domains such standards do not exist or are not sufficient, thus, requiring the developers to customize the standard to their specific needs or create their own domain model. As a consequence, one inevitably has to deal with heterogeneity in the creation of an integrated information system.

Various classifications of heterogeneity can be found in the literature [BKLW99, KS91]. We distinguish four types of heterogeneity:

1. Technical heterogeneity,
2. Data model heterogeneity,
3. Structural heterogeneity,
4. Semantic heterogeneity.

Technical heterogeneity is concerned with issues of technical nature, e.g.

hardware, operating system, or networking infrastructure. Conflicts resulting from technical heterogeneity can be solved with existing middleware technology.

Furthermore, *data model heterogeneity* has to be resolved by the integrators when data is stored in various formats. An integrated information system should be able to integrate not only structured information (e.g. relational or hierarchical data model), but also semi-structured data (e.g. XML) or unstructured documents. Conflicts resulting from data model heterogeneity are usually resolved by wrapping the data sources. Each source wrapper translates between the data model of the IIS and the data model of the information source. While wrapping of Web sources is an active research topic (e.g., [CMM01]), wrapping of structured data can be considered solved. Note that this thesis only considers structured data in the relational data model.

Structural heterogeneity derives from the fact that, given a data model, information can be structured in various ways. Problems arising from structural heterogeneity include data-metadata conflicts (a piece of information that is an attribute value in one data source is an attribute or table name in the other) or partitioning (different groupings of attributes into tables). Various other sorts of conflicts occur due to *semantic heterogeneity*[1]. Even when the data is structured in the same way, the schemata under consideration might be different, e.g., attributes might have varying labels. Naming conflicts result from the use of homonyms (same word having a different meaning) and synonyms (different words having the same meaning) when defining attribute names. We subsume structural heterogeneity and semantic heterogeneity that results in differences between the schemata under *schematic heterogeneity*. Data conflicts, which are also a result of semantic heterogeneity, occur when the same information is represented in different ways, e.g. different formats, abbreviations, or acronyms. In addition, some tables might contain erroneous, contradictory, or missing data, which increases the complexity of data integration.

Fig. 1.3 depicts an extract of an integration scenario with a single source table R and an integrated schema S, which consists of two tables S_1 and S_2. In the following, the integrated schema is called target schema. Source table R contains data about employees, including the name, salary, age, department, and the head of the department. Similar information can be found in the target schema, but structured as two tables S_1 and S_2. Data about the departments, including the name and head of each department, is shown in table S_2. Employees are represented in table S_1 by their name, salary, and a foreign key to the tuple representing the department they work for. The dashed arrow represents that foreign key, while the solid arrows indicate semantic relationships between attributes of the two schemata.

Several examples for structural heterogeneity can be found in the example: Firstly, employee information is structured as a single table in the source, but the target schema contains two tables. Secondly, the name of each employee is represented in two attributes in R, while the target table S_1 has a single

[1]The term "semantic heterogeneity" has been heavily used in the literature with varying meaning. As stated by Öszu and Valduriez, "semantic heterogeneity is a fairly overloaded term without a clear definition" [ÖV99].

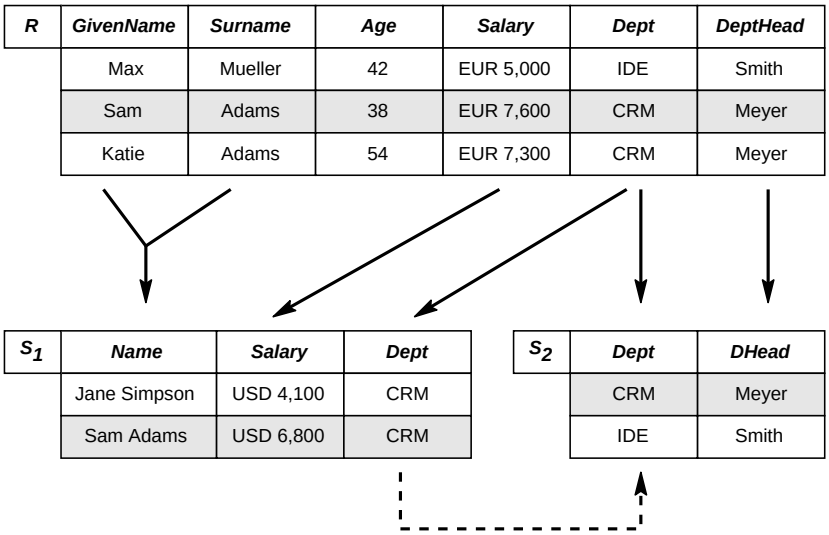


Figure 1.3: Structural and semantic heterogeneity.

attribute containing both given name and surname. Thirdly, the source table R has an attribute Age, which does not have a corresponding partner in the target database.

Semantic heterogeneity occurs both on the schema and instance levels. The attribute representing the head of the department is labelled $DeptHead$ in R and $DHead$ in S_2. In the process of developing the integrated information system, $DeptHead$ and $DHead$ must be identified as synonyms. Both R and S_1 contain an attribute $Salary$ representing the salary of employees. However, both attributes use a different currency: The salary is shown in Euro in R, while S_1 uses US dollars. Finally, data conflicts in duplicate tuples are a result of semantic heterogeneity: One can see that the employee Sam Adams is represented both in the source and the target database. However, after translating the salary in the target database into Euro one notices that the salary of Sam Adams is much lower than in the source database. There are several possible reasons for this conflict, e.g., Sam Adams has received a major pay raise that is not shown in the target database, or the salary in either of the database has been misprinted, etc.

The resolution of heterogeneity is a labor-intensive process. Fortunately, standard solutions exist for some types of heterogeneity. As stated above, middleware applications can be used to bridge different technologies. Data model heterogeneity can be resolved by existing wrapper technology. Unfortunately, there is no automatic solution for structural and semantic heterogeneity. However, two research areas are concerned with those forms of heterogeneity: schema mapping and duplicate detection.

The goal of *schema mapping* is the semi-automatic construction of map-

pings, which describe the structural and semantic relationships between two schemata. A mapping is constructed in two steps: schema matching (i.e., the detection of attribute correspondences) and query discovery (i.e., the generation of a mapping based on the detected correspondences). Attribute correspondences (depicted as solid arrows in Fig. 1.3) connect semantically related attributes. Schema matching solutions, whose goal is to find such attribute correspondences, have to deal with structural and semantic heterogeneity. Based on the detected attribute correspondences, the query discovery process determines a schema mapping. The process is challenging, because several interpretations of a set of correspondences are possible, and ambiguities have to be resolved.

Duplicate detection is the process of identifying different representations of the same real-world entity. An example for such a duplicate are the two representations of Sam Adams in Fig. 1.3. Duplicate detection is complicated by data conflicts as described above. This thesis is concerned with the problem of schema matching, because it is one of the most daunting challenges in data integration. Our proposed solution exploits extensional overlap, and thus, we also have to deal with duplicate detection.

1.3 Types of Integrated Information Systems

As stated above, different types of integrated information system exist. In the following, we describe mediator-based systems, data warehouses, peer data management systems, and multi-database systems. We also show that *schema mappings*, which resolve structural and semantic heterogeneity, are an important part in most of those systems. This section informally describes what mappings are and how they are used by the different systems. In the following section, we discuss two general approaches to the creation of mappings.

1.3.1 The Mediator-based System (MBS)

When developing an integrated information system, one needs to decide if the data is to be stored in the data integration layer, or if the integrated information system uses the sources to answer a user query. The mediator architecture (Fig. 1.4), which was originally described by Wiederhold [Wie92], is an example for *virtual integration*: Instead of storing the data in a database at the data integration layer, the mediator uses the data sources to answer queries. When the user sends a query, the mediator translates the user query into several queries that are sent to the sources. The retrieved results are merged and presented to the user.

The sources can be of different types, e.g., relational or XML database systems, web sites, etc. As stated above, data model heterogeneity is resolved by wrappers, which receive a query from the mediator and translate it into a query that is understood by the source. Afterwards the wrapper retrieves the result and translates it into the canonical data model, i.e., the data model that is used by the mediator.

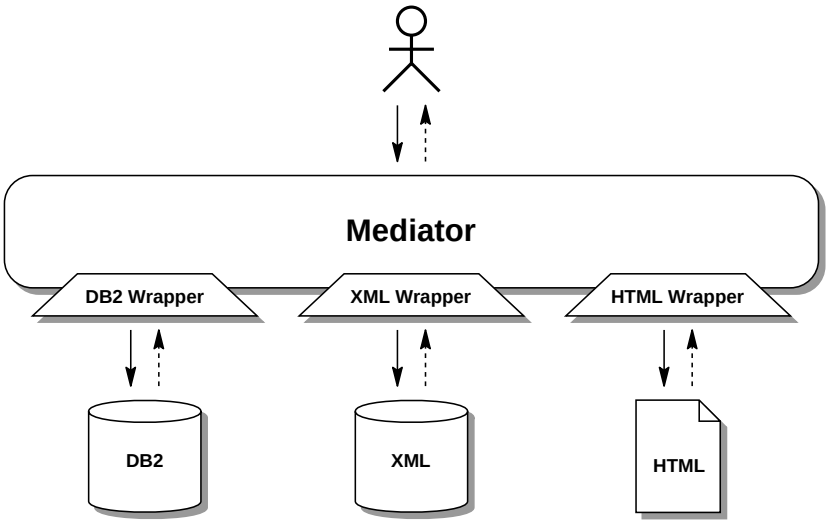


Figure 1.4: The mediator architecture.

Like all integrated information systems, mediator-based systems have the advantage that the user only needs to understand the schema and data model of the mediator. The mediator layer completely hides the details of the sources from the user. In addition, the sources are unaffected by the mediator: They can be used in the same way as they have been used before the mediator was in place. Essentially, the mediator acts as another database application, and the autonomy of the data sources is retained. Unfortunately, this raises additional challenges: Whenever a data source is modified, the mediator and its mappings needs to be adapted. This holds for modifications of the schema, data model, or technical infrastructure of the data source. Automatic detection of such changes is an active research area [LMK03, MAL+05]. In contrast, adding, removing, or modifying data does not affect the mediator. Instead, the use of virtual integration ensures that the user always receives an up-to-date result.

Query Processing in MBS

Query processing in mediator-based systems is driven by mappings, which describe the semantic relationship between the source schemata and the mediator schema [Len02]. Those relationships are usually described as declarative specifications of the data transformation between a source and the mediator. The specification language depends on the mediator and its data model: E.g., TSIMMIS is based on the semi-structured Object Exchange Model (OEM) and uses the Mediator Specification Language (MSL) to specify mappings [GMPQ+97, PAGM96], while Information Manifold integrates relational data and specifies semantic relationships as conjunctive queries [LRO96a, LRO96b].

Another distinguishing feature is the direction of the mapping, which de-

termines the query processing strategy [Ull97]. *Global-as-view* (GaV) systems describe mediator relations as as views over the source schemata. To translate the user query, which is formulated in terms of the mediator schema, into one or several source queries, a technique called *view expansion* (or *query unfolding*) is used. As the name suggests, the mediator relations in the user query are replaced by their definitions in the mappings, resulting in a query containing only source relations. In *local-as-view* (LaV) systems, the sources are described as views over the mediator. E.g., Information Manifold describes each source relation as a conjunctive query involving one or several mediator relations. Query processing resolves to *answering queries using views* [Hal01, PL00].

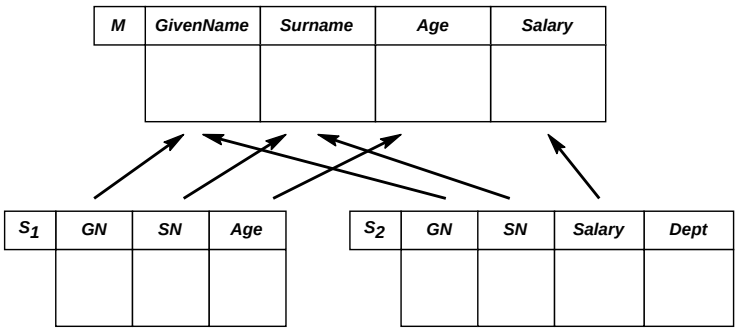


Figure 1.5: Matching between sources and the mediator.

In the following we illustrate GaV query processing using the example depicted in Fig. 1.5. The figure shows a single mediator relation M and two source relations S_1 and S_2. Arrows depict semantic relationships between attributes. To simplify the description, we use conjunctive queries to describe a mapping. In the above example, the GaV mapping would be:

$$M(GN, SN, A, S) \text{ :- } S_1(GN, SN, A), S_2(GN, SN, S, D)$$

which represents a join of S_1 and S_2 on the respective first attribute (GN) and the respective second attribute (SN). The schema of the mediator table does not contain an attribute for the department *Dept*. User queries are formulated in terms of the mediator relation M. E.g., if the user asked for the salary of thirty-year old people, the query would be:

$$Q(S) \text{ :- } M(GN, SN, A, S), A = \text{“}30''.$$

The mediator would expand the view M with its definition, resulting in a query

$$Q(S) \text{ :- } S_1(GN, SN, A), S_2(GN, SN, S, D), A = \text{“}30''.$$

Based on that expanded query, the mediator produces a query plan, which describes how and in what order data sources are accessed. A description of the planning phase in TSIMMIS can be found in [PAGM96].

1.3.2 The Data Warehouse

The management of a large enterprise requires decision support systems to obtain a global view on the company. To extract relevant information about the performance of the enterprise, Online Analytical Processing (OLAP) applications are used [CD97]. Such applications are able to process large amounts of data and extract hidden knowledge, e.g., purchasing patterns of customers. The data used by OLAP tools comes from a *data warehouse*, which in turn receives data from very many operative databases and applications within the company.

In contrast to mediator-based systems, data warehouses perform *materialized integration*: Due to the huge amount of data — the largest data warehouses store several tera bytes of data — accessing the sources every time a user sends a query clearly does not scale. Instead, new data is copied in regular intervals from the sources to the warehouse, and user queries are processed on that local database. In the extract-transform-load (ETL) process, data is extracted from the operative databases and transformed into the warehouse schema before it is loaded into the data warehouse. Because the ETL process is time-consuming and requires a lot of resources, it is usually performed when the load on the operative databases is very low, e.g., at night. Afterwards, the user can send queries even when the operative databases are heavily used, because the data warehouse uses its own copy of the data to produce a query result. Note that the warehouse data is not always up-to-date. However, that does not affect the quality of the query answers because of the nature of OLAP queries: The user is usually interested in aggregates, e.g., sales per month, and the difference between the last warehouse update and the current state of the operative databases has only a negligible effect on the query result.

The ETL process usually involves complex schema transformations, because the schema design of operative databases and warehouse database follows different design principles. On the one hand, the schemata of the operative databases are designed and optimized with respect to their local applications: They closely reflect the application domain and follow normalization rules to avoid redundancy. On the other hand, the warehouse schema usually follows a typical design pattern, e.g., star schema or snowflake schema [CD97]. The ETL process also involves schema mappings that describe the transformation of source data such that the data can be used in later stages of the process.

1.3.3 Other Types of Integrated Information Systems

Peer-to-peer (P2P) systems have become a very active research area in the last decade [DGMY03, ATS04]. While the predominant purpose of P2P networks is file sharing, several efforts have been taken to use apply the peer-to-peer concept in data management [HIST03, HHL+03, NOTZ03, RNHS06]. In a P2P network, there is no distinction between client and sever. Instead, all nodes in the network act as both consumer and provider of information.

While research on P2P file sharing considers very large networks (i.e., hundreds or thousands of nodes), *peer data management (PDMS)* is also conceivable

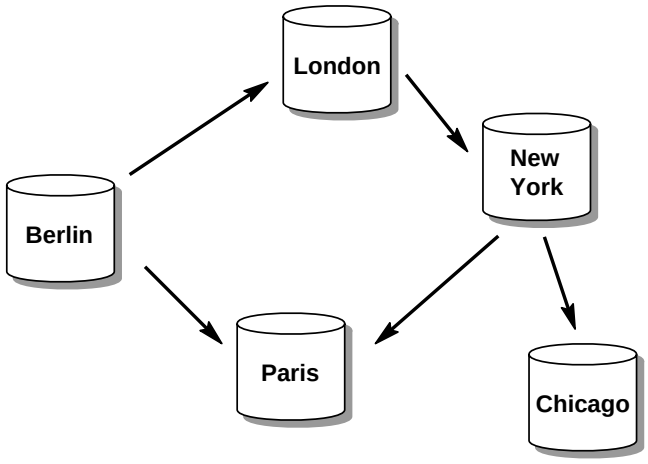


Figure 1.6: A peer data management system.

on a smaller scale. Fig. 1.6 depicts a schematic representation of a PDMS network of a global company: To facilitate information exchange between different locations, five databases are connected in a PDMS network. Each node acts both as a client and a server, i.e., a user in Berlin can send a query based on its local schema, and the result also contains data from the other four locations. The arrows in the figure indicate schema matchings, which are used for query rewriting. Note that although the mappings are directed, queries can be forwarded in any direction using either GaV or LaV query rewriting [HIST03, TH04].

Techniques developed in the context of mediator-based systems, e.g., GaV or LaV query rewriting, can also be applied in PDMS networks. However, additional problems arise due to the more general architecture. If a query has to take several hops from the query node to the network node that contains relevant data, then the mappings on the path need to be combined [MH03, FPKT04]. Some relevant attributes might be lost in the composed mapping, because the schema of an intermediate node does not contain those attributes. This affects query routing: In mediator-based systems, the path the query takes from the user to the source is predefined. In a PDMS network, several distinct paths are possible. Finding a good path that avoids information loss is a challenging problem [ACMH03]. Beside mapping composition and query routing, other problems arise when the network is assumed to be unstable, i.e., if node can dynamically enter and leave the network. Such a system would require (semi-)automatic creation of semantic mappings every time a node enters a network.

Multi-database systems differ from the previously discussed architectures because they do not require predefined mappings [LMR90]. Instead, they merely provide a query language that allows the user to send a single query involving multiple sources [LSS01]. The architecture is similar to the scenario depicted in Fig. 1.1: The user does not use a global schema, but interacts with each

source. In contrast to querying Internet sources individually and merging the results manually, the multi-database system provides a query language that can be used to describe this process in a single expression. This requires the user to know the semantic relationships between the sources. In other words, the user needs a mental model of the mappings in order to specify a reasonable query. One might argue that multi-database systems are no variant of integrated information systems, because they do not provide an integration layer. However, we have added them in this section because multi-database systems are part of data integration research, and the research results are also applicable in other types of IIS.

In the following we describe how mappings are created. Although the discussion is focused on mediator-based systems, the general ideas are also applicable to data warehouses and (to some extent) peer data management systems. Multi-database systems are not of our interest because those systems do not contain schema mappings, but relay the problem of resolving structural and semantic heterogeneity to the user.

1.4 Building An Integrated Information System

Schema mappings, which describe the semantic relationship between heterogeneous schemata, are an integral part of many types of integrated information system. The creation of mappings is closely bound to the principle approach in which a database integration system is built. In this section, we describe two general processes: schema mapping and schema integration.

1.4.1 Schema Mapping: The Top-down Approach

In the top-down approach, the mediator schema is designed independently of the underlying sources. Instead, the schema should closely reflect the user's view on the domain and his information need. After the mediator schema has been defined, relevant data sources need to be added to the system. This process involves the creation of wrappers, which translate the data model of the sources into the data model of the mediator, and the definition of schema mappings, which are used by the mediator for query rewriting. We assume that the former task can be solved by existing standard solutions, and restrict our discussion to the resolution of schematic conflicts in the mapping step.

Sec. 1.3.1 shows a simple example with two schemata and six correspondences (Fig. 1.5). Based on those correspondences, the mapping between those schemata can be specified in a very short period in time. In real-world scenarios, where developers have to deal with dozens or hundreds of relations, the specification of mappings is a time-consuming and error-prone process: Li and Clifton mention a project at GTE, whose aim was the integration of 27,000 data elements [LC00]. It required an average of 4 hours to extract and document one matching element if that task was performed by a person other than the data

owner. Hence, it is absolutely essential to provide tools that aid the user in the mapping process.

The creation of schema mappings is performed in two steps:

1. *Schema Matching*: Semi-automatic detection of attribute correspondences
2. *Query Discovery*: Mapping definition based on detected correspondences.

The first step is the detection of attribute correspondences, which relate semantically related attributes (depicted as arrows in Fig. 1.5). To create a matching, conflicts resulting from structural and semantic heterogeneity must be resolved. This process is considered to be semi-automatic, i.e., interaction with the user is always required, because available information is usually not sufficient. Most algorithms rely on schema metadata or instance information. Documentation rarely exists in a format that can be exploited by automatic tools. Hence, the schema matching detected by automatic tools is usually imperfect and needs to be adapted by the user. Schemata and instances do not provide enough information to determine the desired matching with 100% accuracy, because the intention of the developer is not fully specified in them. Related work on schema matching is described in Sec. 2.3.

Schema mappings are created in the second step based on the detected correspondences. Because in most cases different interpretations of the same matching are conceivable, the user might have to interact with the tool: E.g., the correct mapping of the above example defines a join of S_1 and S_2. However, based on the attribute correspondences depicted in Fig. 1.5, a union of the two relations is also conceivable, which results in the following mapping:

$$\begin{aligned} M(GN, SN, A, \texttt{null}) &\text{ :- } S_1(GN, SN, A) \\ M(GN, SN, \texttt{null}, S) &\text{ :- } S_2(GN, SN, S, D) \end{aligned}$$

where `null` is a special value stating that the value is unknown. In such a case, the user needs to decide which of the mappings is correct. Query discovery techniques are discussed in Sec. 2.4.2.

1.4.2 Schema Integration: The Bottom-Up Approach

In the example of Fig. 1.5, the mediator does not contain all source data because the attribute *Dept* is missing. Some applications require all source data to be part of the integrated information system. E.g., when two companies merge and their databases are to be integrated, no data must be lost in the process. In such a scenario, a top-down approach is not advisable. Instead, the system should be built bottom-up using a schema integration process [BLN86, SPD92, Bus02, PB03].

Fig. 1.7 depicts two source schemata S_1 and S_2, and the schema M that is the result of integrating the two source schemata. The schema integration proceeds in two steps:

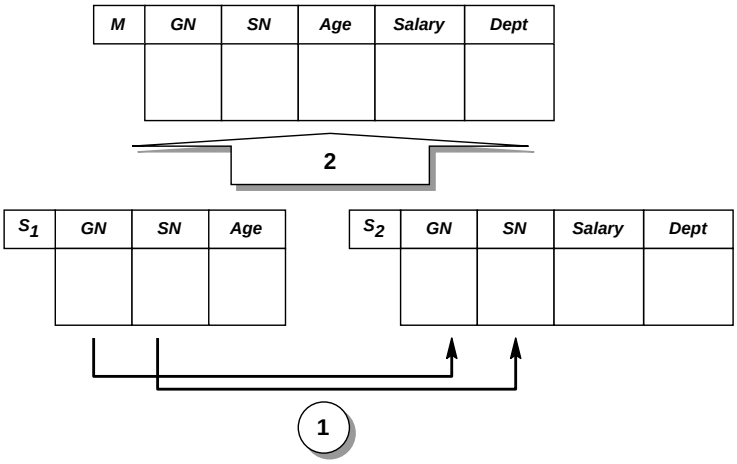


Figure 1.7: Integration of two schemata.

1. *Description of Inter-schema Correspondences*: Common schema elements are identified and their relationships are specified.

2. *Schema Integration*: The schemata are merged based on the specified inter-schema correspondences. The result is an integrated schema and mappings that describe the semantic relationships between the source schemata and the integrated schema.

In the first step, the user needs to determine the semantic overlap between the databases, which is formally described as inter-schema correspondences. Note that inter-schema correspondences in schema integration are different than attribute correspondences in schema matching: While the former provide a very specific description of the relationship between schemata both on the intensional and the extensional level, the latter only represent a relationship between attributes without well-defined semantics.

Although schema matching is usually discussed as a first step in schema mapping, we believe that it can be a viable tool in schema integration as well. Based on the detected attribute correspondences, the semantic relationships between the source schemata can be declaratively described in a correspondence language like MoCA [Bus02]. Using the generic language defined in [SPD92] and assuming that the two tables overlap in their extension (i.e., some people are represented in both tables), the inter-schema correspondences of the above example are:

$$S_1.FN \cap S_2.FN$$
$$S_1.GN \cap S_2.GN.$$

The predicate $\cap$ shows that the extensions of those attributes intersect. In the second step, the integrated schema is created based on the input schemata

and the inter-schema correspondence assertions. Because the FN attributes are related, they are represented as a single attribute in the integrated schema M. The GN attributes are handled likewise. The schema mappings between the integrated schema and the source schema are created along the way [SPD92].

1.5 Duplicate-based Schema Matching

We have shown that mappings play a major role in many integrated information systems. The creation of mappings is a time-consuming process, but the developers can be aided by semi-automatic tools. One of the most daunting challenges in the generation of mappings is the detection of attribute correspondences. To support the developers in that process, various schema matching solutions have been proposed. Each of them use different kinds of information to detect correspondences, e.g., metadata contained in the database schema or attribute-specific features extracted from attribute values. The schema matcher described in this thesis exploits the fact that in many data integration scenarios an extensional overlap exists, i.e., some real-world entities are represented in both databases. The *DUMAS* (duplicate-based matching of schemata) algorithm is able to detect such duplicates and extract attribute correspondences from them.

1.5.1 Example Scenarios

While relying on duplicates for schema matching appears to be too restrictive at first glance, we emphasize that in many data integration scenarios duplicates exist:

Customer Relationship Management (CRM). In large enterprises, different departments maintain their own data about customers. The customer information can be available in databases, spreadsheets, or unstructured files. Such an infrastructure is very inefficient when the management needs all information about their customer base. Customer Relationship Management (CRM) systems are established to avoid searching all existing data sources for the required information.

The main motivation for establishing a CRM system is the fact that information about the same customer is spread over multiple sources. Hence, we can assume that duplicates exist in such a scenario. Detecting those duplicates is no trivial task, because customer data might change over time (e.g., because the customer has moved or has received a new phone number), and some departments have not updated their databases. In addition, we have to deal with schematic heterogeneity, because some information about a customer might be represented in only a single source, e.g., because that information is only relevant for one department. However, we can assume that every data source contains attributes that can be used for identifying a customer (e.g., name, address, phone number, etc.).

Comparison Shopping. Many retailers provide stores on the Internet, which can be easily queried by the user to search for a given product. As in traditional stores, the prize for a given product differs from store to store. Because Internet stores are easily accessible, comparison shopping agents have been created that search for the best price of a given product.

Adding a store to a comparison shopping agent involves many tasks beside schema matching, e.g., the creation of wrappers, which extract information from Web pages and translate the data into relations. To find correspondences between the store and the agent, one could exploit sample data that has been extracted from the Web pages by crawling the store's site. Note that duplicates can be expected in a comparison shopping scenario: If a given product were not available in several stores, searching for the lowest prize would not be an issue. In all domains, the description of a product contains enough information for the user to identify a product: E.g., books usually have a title, an author, a publisher, and an edition. Books also have an ISBN number, which is globally unique. However, such an identifying attribute cannot be expected in all domains.

Catalogue Integration. When two retailers merge, they want their customers to have access to all available products. Unless the two companies are kept separate from each other, the product databases need to be integrated.

The process of combining two catalogues is similar to finding a best price for a given product: In both cases, one has to identify different representations of the same product. One also has to resolve schematic differences, because some information can only be found in one of the catalogues. The main difference is that catalogue integration can be performed once, and new products can be added to the integrated catalogue. In contrast, comparison shopping agents cannot modify the source databases, and thus, have to regularly query the sources to get the latest price.

The data integration problems described above show some characteristics of duplicate-based schema matching scenarios: Two or more data sources exist, and some real-world entities are represented in several sources. The attributes used to describe the entities differ, because only information that is required by local applications is kept in the databases. However, there is sufficient data to identify an entity: While globally unique identifiers (e.g., ISBN number) cannot be expected, the identity of a real-world object can be deduced from a few attributes that exist in all sources with sufficient accuracy.

Detecting duplicates is no trivial task even when descriptive attributes exist, because the attribute values for the same entity might differ. One reason for different values is the fact that the same information can be represented in different ways: E.g., "Microsoft" and "Microsoft Inc." both refer to the same company. Data quality issues are another reason for differing values: Information might be out-of-date as in the CRM scenario, or some attribute values are erroneous. Due to such inconsistencies, finding duplicates with very high accuracy is a challenging problem.

1.5.2 The DUMAS Approach

$\boldsymbol{R}$	$FirstName$	$LastName$	Sex	$Phone$	Fax
r_1	John	Doe	m	(408) 7573339	(408) 7573338
r_2	Joe	Smith	m	(249) 3615616	(249) 2342366
r_3	Suzy	Klein	f	(358) 2436321	(358) 2436321
r_4	Sam	Adams	m	(541) 8127100	(541) 8121164
r_5	Mark	Spitz	m	(901) 8319311	(901) 8612382
r_6	Jim	Beam	$\perp$	(782) 1238957	(781) 1883744
r_7	Kate	Moss	f	(124) 9654565	$\perp$
r_8	Sam	Wong	f	(124) 4955670	(999) 9999999
r_9	John	Dean	m	(369) 3663624	(367) 3663625

$\boldsymbol{S}$	LN	Acc	Tel	OS
s_1	Douglas	jdouglas	(408) 9182043	XP
s_2	Dean	jd	(369) 3663624	XP
s_3	Klein	littlesue	(358) 2436321	UNIX
s_4	Adams	sam	(541) 8127100	W2000
s_5	Wong	kate	(923) 6363443	Linux
s_6	Kurz	itsme	$\perp$	UNIX

Figure 1.8: Relations R and S with intensional and extensional overlap.

To illustrate the general idea, consider the example depicted in Fig. 1.8. Both relations R and S contain information about people, but are differently structured: Relation R has attributes for the first name, last name, gender, phone number, and fax number, while relation S contains attributes representing last name, user account, phone number, and operating system. The goal of schema matching is to detect correspondences between semantically related attributes. In the example scenario only the following correspondences exist:

$$LastName \longrightarrow LN$$
$$Phone \longrightarrow Tel$$

Various semi-automatic solutions exploiting different kinds of 'hints' from the available data have been proposed (see Sec. 2.3). Schema-based algorithms use metadata extracted from the database schemata, e.g., attribute names or data types. Those approaches suffer from the fact that autonomously developed databases use different names to describe attribute names. In the above example one might guess that LN is an acronym for "last name", but inferring that Tel is a synonym for $Phone$ requires more sophisticated means than string comparison. Note that in some cases database designers chose to use names that are not related to the semantics of the attributes, which renders schema-based matchers useless.

Instance-based matchers do not suffer from the above drawbacks because they use actual data from the tables. Most proposed algorithms extract characteristic features from the values of each attribute and match attributes that have similar features. This approach works well if a good set of distinguishing features is chosen, which is a challenging problem itself. Unfortunately, some attributes are indistinguishable even when a reasonable feature set is used: Relation R in the above example contains attributes for phone number and fax number. The values of those attributes have the same structure. In fact, even a human observer is unlikely to be able to distinguish phone and fax numbers without knowing the respective attribute labels. Thus, it seems impossible to determine which of them matches with Tel in relation S.

The problems described above can be solved by following the DUMAS approach, which exploits duplicate tuples for schema matching. Different representations of the same real-world object are called duplicates. In the example in Fig. 1.8, tuples r_3, r_4, and r_9 in R represent the same entities as tuples s_3, s_4, and s_2 in S, respectively. Those duplicates provide valuable information that can be used for schema matching: E.g., the tuple pairs representing the same entity always have the same value in $LastName$ and LN, thus indicating that the two attributes correspond. Two duplicates, namely (r_4, s_4) and (r_9, s_2), also provide hints that help in distinguishing $Phone$ and Fax: Both r_4 and r_9 have different values for $Phone$ and Fax, and the $Phone$ values equal the Tel values in the respective matching tuple. In this thesis we show how to extract attribute correspondences from duplicate tuple pairs.

In principle, any duplicate-based schema matching must proceed in two steps:

1. *Duplicate detection*: A few duplicate tuples are found in the databases, and

2. *Matching*: Attribute correspondences are extracted from the duplicates.

The goal of the first step is to detect a few duplicates. As will be shown in this thesis, the actual number of duplicates required for matching can be quite small and is far from the total number od duplicates in the data set. Those duplicates are used to discover attribute correspondences in the second step.

Contributions of this Thesis

Although this general process appears to be simple and has been considered in related work (see Sec. 2.3.4), some important aspects (e.g., duplicate detection in unaligned schemata or multi-table duplicates) have not been tackled at all, while other problems are solved only to a small extent. With this thesis we make the following contributions:

Duplicate detection in unaligned schemata. The problem of duplicate detection has been considered under various names, e.g., entity identification, deduplication, or record linkage. Several solutions have been proposed, but all of them have one property in common: They require the attribute correspondences

to be known. Unfortunately, finding attribute correspondences is the goal of schema matching, and thus, existing solutions cannot be used in the duplicate detection step of our approach.

Hence, we have to solve the problem of finding duplicates when correspondences are not known, which has not been considered before. We show the challenges associated with that problem and describe an algorithm that is able to detect at least a few duplicates even when the semantics of the attributes are unknown. The experiments indicate that the proposed method works well even in difficult cases where only few duplicates exist and when only few attributes correspond.

Matching single tables based on duplicates. The goal of the second step is to use the detected duplicates to establish semantic correspondences. In other words, intensional similarities have to be deduced from extensional 'hints', which are provided by attribute values. For that purpose, we define a domain-independent similarity function that is used to compare attribute values. Based on the similarity scores of several duplicates, we extract a set of 1:1 correspondences. The experimental evaluation indicates that it is possible to extract a high-quality matching from only a few duplicates. It also shows that the algorithm is robust with respect to false duplicates.

Matching complex schemata based on duplicates. Matching schemata consisting of multiple tables raises several new challenges. The algorithm for matching single tables assumes that the tables are semantically related and contain duplicates. Those assumptions do not hold when comparing two arbitrary tables in two schemata, and thus, might produce false correspondences. To solve this problem, we have devised heuristics used to determine if a matching can be trusted. Those heuristics are part of the schema matching algorithm, which detects duplicates spanning multiple tables and extracts correspondences from them. It has to be noted that the problem of such *multi-table duplicates* also has not been considered before. The matching algorithm is designed to compare only relevant tables to reduce computation cost.

Finding complex matchings. The problem of complex (i.e., m:n) matchings has only recently been approached in research. The space of possible complex matchings is much larger than the the solution space of simple (i.e., 1:1) matchings, and thus, efficiency is a major concern apart from the correctness of the detected correspondences. We propose a novel solution to the problem of complex matchings which is based on the simple matching detected by our algorithm. The complex matching algorithm improves the matching by merging certain attributes, and thus, producing complex correspondences. The performance issue is tackled by only considering promising attribute combinations. Our experimental evaluation shows that the proposed algorithm is able to detect complex correspondences with high accuracy.

The DUMAS approach was first proposed in [Bil04]. The DUMAS table matching algorithm, which includes duplicate detection in unaligned tables and the extraction of simple correspondences from them, is described in [BN05].

The DUMAS table matcher has also been incorporated into the *Humboldt Merger* (HumMer) [BBB+05]. HumMer is a semi-automatic data integration

and cleansing tool: Given two or several source tables, the tool first discovers attribute correspondences. Based on those correspondences, the table schemata are merged into a single integrated schema. After inserting the source data into the integrated table, duplicates are detected and removed. The user can interact within the different stages by altering the schema matching, guiding the duplicate detection process, or defining conflict resolution rules.

Structure of this Thesis

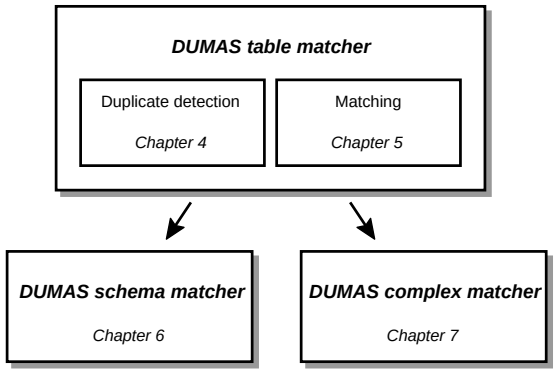


Figure 1.9: DUMAS algorithms and related chapters.

The DUMAS algorithms and the chapters that describe them are shown in Fig. 1.9. Before specifying the DUMAS matchers, we discuss the problem of schema matching in Chap. 2. We lay down the formal basis for the remainder of the thesis and review related work on schema matching. In addition, we describe how schema mappings are generated based on attribute correspondences.

Chap. 3 describes the DUMAS approach to schema matching. Because the DUMAS matcher exploits duplicates, we discuss the duplicate detection problem and review related work in that area.

Our solution to the problem of duplicate detection in unaligned tables is presented in Chap. 4. We discuss the challenges associated with the problem, describe an algorithm to detect the top-k duplicates, and we evaluate the quality of the detected duplicates using both synthetic and real-world data. The process to extract attribute correspondences from those duplicates in described in the following Chap. 5. We show experimentally that the matching step produces a good result even in the presence of errors.

The following two chapters are concerned with extensions of the DUMAS table matcher. Chap. 6 describes the DUMAS schema matcher, which detects multi-table duplicates and uses them to extract attribute correspondences between complex schemata consisting of several tables. We show that duplicate-based schema matching in complex schemata cannot be done by mere application of the DUMAS table matcher on each table combination, because several underlying assumptions do not always hold. We describe under what conditions

a matching can be trusted and how known attribute correspondences can be used to detect further attribute correspondences. The DUMAS schema matcher uses these heuristics to detect correspondences in schemata involving multiple tables.

The DUMAS complex matcher, which detects complex correspondences between two tables, is presented in Chap. 7. Finding complex matchings is a challenging problem, which has not received great attention in the literature. We propose an algorithm that is based on the DUMAS table matcher: The simple matching is used as input, and attributes are merged if that improves the matching. The experiments show that the resulting complex matching is of high quality. We conclude with a discussion of the DUMAS schema matcher in Chap. 8.

Chapter 2

The Schema Matching Problem

In the previous chapter problems associated with translating data between heterogeneous schemata are informally described. The goal of this chapter is to lay down the formal foundation for the following chapters. We start by defining the notation used throughout this thesis. As the proposed schema matching solution is based on relational schemata, this chapter will include a description of necessary relational concepts. Afterwards the problem of schema matching is formally defined, and related work in that area is discussed. We also discuss related theoretical and practical work on schema mapping.

2.1 Basic Relational Concepts

The schema matching approach described in this thesis is based on relational schemata. Consequently, the notation for relational concepts used in this thesis needs to be defined. Note that only basic relational concepts are required to define schema matching; an in-depth examination of database theory can be found in [AHV95].

As mentioned above, only a few basic relational concepts are required to define schema matching. A *database schema* or, simply, a schema, is a finite set $\mathbf{R} = \{R_1, \ldots, R_k\}$ of relation schemata. A *relation schema* $R_i = \langle a_1, \ldots, a_l \rangle$ is a sequence of attributes. If the relation schema is clear from the context, we omit the subscript i. Each *attribute* a_j has an associate domain denoted by $Dom(a_j)$ consisting of constants and the special `null` value, which we denote by $\perp$. Furthermore, we define $att(\mathbf{R})$ and $att(R)$ to be the set of attributes in the database schema $\mathbf{R}$ and relation schema R, respectively.

A *database instance* $I(\mathbf{R}) = \{I(R_1), \ldots, I(R_k)\}$, i.e., an instance of a database schema, is a set of instances of its constituent relation schemata. An instance $I(R) = \{r_1, \ldots, r_m\}$ of a relation schema R, or *relation* for short, is a set of tuples. A tuple $r_i \in Dom(a_1) \times \ldots \times Dom(a_l)$ over R is a sequence of values

$\langle v_1, \ldots, v_l \rangle$, where the value v_j for attribute a_j is an element of $Dom(a_j)$.

2.2 Problem Description

2.2.1 Schema Matching and Attribute Correspondences

Although schema matching is applied in various application areas, the basic problem can always be formulated intuitively as follows: Given a source schema and a target schema, find the attribute correspondences that describe how both schemata are related. One such *attribute correspondence* connects a set of source attributes with a set of target attributes, and implies that data from the source attributes is used to generate data in the target attributes.

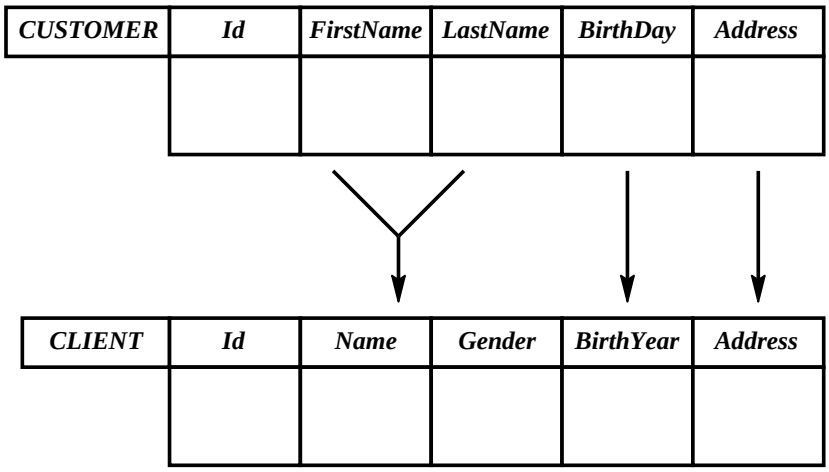


Figure 2.1: A schema matching example.

An example is depicted in Fig. 2.1. Each of the two schemata consists of a single relation containing information about customers. Attribute correspondences are depicted as arrows between the source and the target schema. It is important to note that the generation of target data can involve difficult transformations. In the example, $FirstName$ and $LastName$ should be concatenated to produce values for the attribute $Name$ in the target schema, while the year in the dates of birth in $BirthDay$ needs to be extracted to form values for $BirthYear$. The work described in this thesis is only interested in finding correspondences between attributes and does not consider transformation functions as part of the schema matching output.

Also note that the desired schema matching result is very subjective. In the example, both relations contain an Id attribute, which represents an identifier that is unique within its relation. However, there is no correspondence between the two attributes, i.e., the developer has chosen not to use identifiers of the source relations to generate identifiers for the target table. There are

several possible reasons for this decision: The two databases could use different numbering schemes for creating identifiers, or the source databases uses simple numbers while the target database uses 'semantic' identifiers based on other attribute values. However, despite the differences, the developer could also have chosen to use source identifiers in the target table as long as those values do not violate constraints on the target.

This subjectivity of the desired schema matching result is one reason why a schema matching can be established only semi-automatically. In all but the simplest scenarios, the developer needs to step in, either by adjusting the matching or by interacting with the schema matching program.

2.2.2 Formal Problem Description

As stated above, the desired schema matching is very subjective and might vary in different scenarios. Thus, given two databases, it is impossible to define the desired matching between them. However, we can define the structure of the input and the output of a schema matching component.

The minimal input for the schema matching problem is a *source schema* $\mathbf{R} = \{R_1, \ldots, R_k\}$ and a *target schema* $\mathbf{S} = \{S_1, \ldots, S_l\}$. The schemata are always required because the output of the schema matching process is defined in terms of the schemata's attributes. Additional input, e.g., instances of the schemata, metadata, or documentation, has been shown to improve the schema matching result. As described in Sec. 2.3, different schema matching solutions exploit different kinds of input.

Given a source schema $\mathbf{R}$ and a target schema $\mathbf{S}$, the output is a schema matching between $\mathbf{R}$ and $\mathbf{S}$. A *schema matching* $\mathcal{M} = \{(a, b) | a \in 2^{att(\mathbf{R})} \wedge b \in 2^{att(\mathbf{S})}\}$, or matching, is a set of *attribute correspondences* (a, b), such that a is a set of attributes in the source schema (i.e., it is a set of source attributes), b is a set of attributes in the target schema (i.e., it is a set of target attributes), and values of a are used to generate values of b in the desired mapping. We say that a group of attributes a *matches* attributes b if a correspondence (a, b) is part of the matching. We denote the number of attributes in a by $|a|$. The process of creating such a matching is called schema matching, too.

Note that other definitions of the schema matching output are also conceivable. *Structure-level matching* considers correspondences both between atomic schema elements (e.g., attributes in a relational schema) and between higher-level elements (e.g., relations) [RB01]. This is of particular interest in the case where the data model allows multiple levels of structuring. For instance, beside matching leaf elements of two XML schemata, correspondences between intermediate XML nodes can be of interest, too. However, as our approach works on relational schemata, and because the mapping generation methods described in Sec. 2.4.2 only use correspondences between atomic schema elements, we do not consider structure-level matching.

Matchings can be classified based on their cardinality (Table 2.1). *Simple matchings* are 1:1 matchings, i.e., each attribute has zero or one corresponding

partner. Thus, the attribute correspondences contained in the matching are constrained to pairs of singleton attribute groups. This restriction does not hold for *complex matchings*: In an m:n matching, each attribute can be part of an attribute group that corresponds to any number of other attributes. The group of 1:n matchings also falls into the category of complex matchings, but has the constraint that a group of target attributes corresponds to at most one source attribute. Note that n:1 matchings are defined accordingly, and we use the term 1:n matching when direction is irrelevant. Analogous to the above definitions, a correspondence (a, b) is a *simple correspondence* if both a and b are singletons, or a *complex correspondence* if a or b contain more than one attribute.

Matching class	Cardinality	Constraint on $\mathcal{M}$
Simple matching	1:1	$\forall(a,b) \in \mathcal{M} : \lvert a\rvert = 1 \wedge \lvert b\rvert = 1$
Complex Matching	1:n	$\forall(a,b) \in \mathcal{M} : \lvert a\rvert = 1$
	n:1	$\forall(a,b) \in \mathcal{M} : \lvert b\rvert = 1$
	m:n	—

Table 2.1: Classification of schema matchings based on their cardinality

2.3 An Overview of Schema Matching Approaches

Finding attribute correspondences both for schema mapping or for schema integration is a tedious and error-prone process that is mostly done manually [LNE89]. Semi-automatic tools that support the user can reduce development time and cost. The schema matching problem has received increased attention in the past decade, and various solutions have been proposed by the research community. In the following, we classify different approaches and describe their relationship to our algorithms.

2.3.1 Classification of Schema Matching Approaches

This classification of schema matching approaches is based on the excellent survey by Rahm and Bernstein [RB01]. They define several, largely orthogonal criteria, which they use to describe schema matching algorithms. In this thesis, we restrict ourselves to two criteria, which were used in [RB01], and add a third classification criterion for instance-based matchers (Fig. 2.2):

1. Individual matcher vs. combining matchers,
2. Schema-based vs. instance-based (applicable for single matchers), and
3. Vertical vs. horizontal (applicable for instance-based matchers).

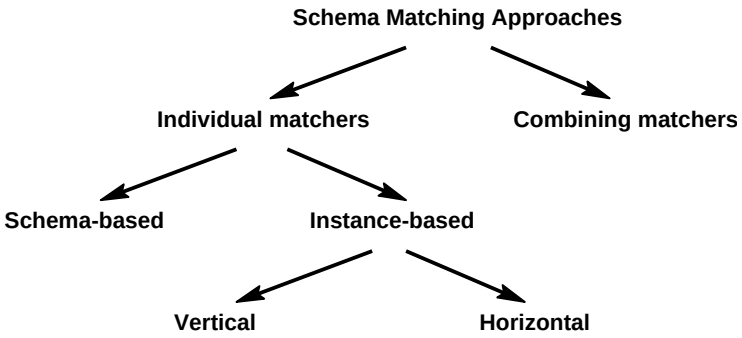


Figure 2.2: A classification of schema matching approaches.

Individual matchers exploit only one type of information, e.g., schema or instances, to detect attribute correspondences. One advantage of this work is the opportunity to identify potentials and restrictions of using a specific type of information for schema matching. Although they achieve reasonably good results, it is generally agreed that best results can be achieved by *combining matchers*, which will be discussed in Sec. 2.3.5.

Individual matchers can be further organized into schema-based and instance-based matchers. *Schema-based* matchers only work on available metadata, such as attribute labels, data type, or schema structure (see Sec. 2.3.2). *Instance-based* matchers exploit available data (Sec. 2.3.3). In addition to the criteria defined in [RB01], we further distinguish instance-based matchers which perform *vertical* matching from those which perform *horizontal* matching. Because our duplicate-based approach falls into the category of horizontal matchers, related work in that area is discussed in detail in Sec. 2.3.4.

Before describing schema matching algorithms, we point out that most solutions have been tested with small data sets. However, scaling schema matching to large schemata has received additional attention in the recent past [BMPQ04, RDM04].

In the following, we give a brief overview of related work in schema matching. We also include work in the area of *web interface matching*, which is a challenging problem related to the integration of information sources on the World Wide Web. The goal of web interface matching is to find correspondences between fields of query interfaces. Such fields are usually text input areas, but can also include a predefined list of possible values or radio buttons. Thus, web interfaces matching is similar to schema matching, but instead of attribute correspondences the goal is to find correspondences between web interface elements.

2.3.2 Schema-based Matchers

As the name suggests, schema-based matchers exploit available schema information, which can be extracted from a database's metadata repository [MBR01,

MGMR02, MZ98, BCV99]. The general approach of schema-based matchers is to compare schema elements (i.e., attributes) using a similarity measure, and to regard attribute pairs whose similarity is above a given threshold as corresponding. The similarity measure can take into consideration different kinds of metadata: It could incorporate similarity of attribute labels, data type, etc. More advanced approaches also apply external, 'semantic' metadata to improve the schema matching. For instance, MOMIS uses the WordNet thesaurus, which relates terms through synonym and broader term / narrower term relationships, to identify correspondences between attributes whose name is lexicographically different, but whose semantic is assumed to be similar [BCV99].

For structure-level matching, as described in Sec. 2.2.2, where in addition to atomic schema elements higher-level elements are to be matched, taking the structure of the schema into consideration can greatly improve the matching result. Matching algorithms like Cupid [MBR01] or Similarity Flooding [MGMR02] are based on the following intuition: If two elements are considered similar, their neighbors (i.e., connected schema elements) are also likely to be similar. In the example in Fig. 2.1, we can identify many corresponding attributes. This indicates a correspondence on the upper level, i.e., between the tables $CUSTOMER$ and $CLIENT$, although their names are very dissimilar.

Most web interface matchers are also schema-based, because the data is hidden behind the web interface and can only be accessed by querying. Thus, methods that apply the above ideas have also been proposed in the context of web data integration [HC03, HCCH04]. In contrast to the work on schema matching, which deals with only two or a few schemata, integration of web data sources is targeted at hundreds or even thousands of sources. On the one hand, this requires the matching algorithms to be much more scalable. But on the other hand, the large number of interfaces facilitates the application of data mining algorithms. E.g., DCM matches schemata holistically: The algorithm does not compare two schema at a time, but all of them at once [WDM04]. Two attributes or groups of attributes are assumed to be synonyms, and thus corresponding, if they never appear together in the same schema. Attributes that co-occur in many schemata are assumed to form a group. The authors developed a correlation measure that takes into consideration the specific characteristics of query interfaces on the Web, and used it to mine complex matchings from a corpus of web interfaces.

Schema-based approaches work if attributes have names that unambiguously define their semantics. Unfortunately, this is not always the case. In general, the database developer has several options how to design the database schema. Thus, if two schemata have been developed independently, most likely different design decisions have been made. Beside different ways to structure information, attributes can be given different names. A common problem that has to overcome by schema-based matchers is the occurrence of *homonyms* and *synonyms*: Semantically different attributes can be given the same name (homonym), while corresponding attributes may have different names (synonym). As discussed above, external metadata like ontologies can be used to tackle this problem, but in many cases attribute names do not follow any standards.

2.3.3 Instance-based Matchers

In cases where schema-based approaches fail, exploiting actual data instead of or in addition to metadata can result in a better matching. Most instance-based matchers perform vertical matching: They extract features from each attribute in isolation. These features act as a description of the attribute's semantics. In other words, the description of an attributes is automatically extracted as opposed to manually specified as attribute names. Consequently, attributes with similar features are matched. We discuss such matchers in the following. Another group of matchers performs horizontal matching: They try to detect duplicates, i.e., different tuples that represent the same real-world entity, and match attributes using the detected duplicates. Such algorithms are discussed in Sec. 2.3.4.

Examples for vertical matchers are Automatch [BM02], Clio [NHT+02], SEMINT [LC94, LC00], and the content matcher of LSD [DDH01, DDH03], which we call CM in the following. Those solutions consider schema matching as a classification problem: The target attributes are considered as classes, to which each source attribute is to be assigned. To classify source attributes, they employ supervised learning methods. Thus, they start with a training phase, in which a model for classification problem is learned. To do so, they use data from previously matched sources. In the following matching phase, additional sources can be matched to the target schema.

SEMINT extracts 20 features from each attribute, e.g., data length, standard deviation, and average value. These features are used to train a neural network. After the training phase, the neural network can be used to match other sources to the same target schema. CM employs a naïve Bayes learner that is trained on the actual attribute values. After the training phase, CM would give a large similarity score to attributes that have similar values.

Instance-based schema matchers that employ vertical matching as described above are not affected by different names of schema elements. However, a problem similar to homonyms or synonyms in attribute names can occur in the attribute descriptions that are extracted from the data. An example for the *homonym* case is depicted in Fig. 2.3. The figure contains two tables describing persons by name, place of birth, and place of residence in R and by name and place of birth in S. Birthplace and place of residence both contain names of places. Thus, the features extracted from them will look alike, although their semantics is different. The *synonym* problem occurs if the descriptions of two attributes are different although they correspond. Such a case can occur if the same information is represented in different ways: E.g. the gender of a person might be abbreviated as "m" or "f" in one database, while it is represented as "0" and "1" in another database.

In addition to synonyms and homonyms, the instance-based matchers described here suffer from a problem related to their underlying learning approach, namely feature selection. Deciding which features to extract from the underlying data is a problem that has been extensively studied in machine learning [GE03]. Each of the above mentioned algorithms uses a different set of features, and

R	Name	Birthplace	Residence
	John Doe	London	Berlin
	Max Mustermann	Berlin	NYC
	Sam Adams	NYC	London
	⋮	⋮	⋮

S	Name	POB
	John Doe	London
	Suzy Klein	Berlin
	Sam Adams	NYC
	⋮	⋮

Figure 2.3: An example for columns with similar features but different semantics.

it seems highly unlikely that a 'best' set of features that produces an optimal schema matching in varying scenarios will be devised in the future.

We emphasize that different instance-based matching approaches that do not fit into our classification hierarchy are conceivable. E.g., the matcher describe by Kang and Naughton [KN03] neither learns attribute characteristics nor compares duplicate tuples. Instead, it determines fuzzy dependencies between attributes within a table using various information-theoretic tools. The result is a dependency graph for each table. Attribute correspondences are derived by comparing the dependency graphs of the tables that are to be matched.

2.3.4 Duplicate-based Matching Approaches

As will be shown in Chap. 3, instance-based matchers that perform horizontal matching are potentially able to distinguish semantically different attributes that have similar features. The basic idea of duplicate-based matchers is to search for fuzzy duplicates and use those duplicates to infer attribute correspondences. We are aware of three schema matchers that exploit duplicates to some extent.

The Internet Learning Agent (ILA) can be considered a schema matcher although it is targeted towards information sources on the Web [PE95]. Unlike web interface matchers, which establish correspondences between query interfaces, ILA's goal is to establish correspondences between its internal model of the application domain and the model of an information source on the Internet. The model of the information source includes all information that is contained in a Web page that is returned as the result of a query. To establish a matching, ILA chooses objects from its internal model and queries the source using a keyword query interface. The returned result are used to learn the semantics of the source's attributes. This process is based on the correspondence heuristic, which has two components: (i) if an attribute value of the internal object is equal to a value of the result, the two attributes are assumed to be related, and (ii) the meaning of an attribute is consistent across all individuals.

One drawback of this approach is related to the fact that it is a web-based tool: Because the source's data is hidden behind a interface, the detection of duplicates becomes a matter of sending keyword queries. To achieve a reasonably good result, the internal model and the source's model must have a large

overlap in the real-world objects that they represent. In addition, the algorithm only considers equal values as a match, which is too restrictive in heterogeneous environments, where data is dirty or differently represented.

The matcher iMap combines multiple individual matchers [DLD+04] (see Sec. 2.3.5). In cases where an extensional overlap exists, it uses special 'overlap modules'. These modules assume that duplicate tuples are provided by the user, and thus, completely ignore the problem of duplicate detection. In most cases, the overlap module simply uses its non-overlap counterpart to find an initial matching, then re-evaluates the matches using the duplicates.

Horizontal matching according to Chua et al.

Chua et al. propose a matching algorithm that is most closely related to our solution, and thus, we discuss their approach in more detail [CCL03]. The algorithm produces a set of complex correspondences between two tables in three phases:

1. *Classification of attributes and formation of attribute groups*: Each attribute is heuristically assigned to a predefined domain class. Based on this classification, attributes are grouped together.

2. *Measurement of correspondence score*: Attribute groups are compared with each other and a correspondence score, which reflects the similarity of the attribute groups, is computed.

3. *Matching attribute groups*: Source attribute groups are uniquely assigned to target attribute groups. Each assignment represents a complex correspondence.

Domain classes are organized into three domain hierarchies: KEY, STRING, and CODE. The KEY hierarchy includes classes CANDIDATE KEY and FOREIGN KEY. The STRING hierarchy is used to classify alphanumeric attributes. It contains classes ALPHABETIC, ZERONINE, and MIXED. The CODED domain class organizes mathematical/statistical properties. Example classes are ORDINAL (representing values that can be ordered), NOMINAL (values with a nominal scale), and DATE.

In the first phase of the algorithm, attributes are classified by assigning each attribute a domain class. This process is done automatically using various techniques: E.g., key attributes are determined by accessing the data dictionary, while coded domain classes require the application of heuristic rules and data analysis. Afterwards, attributes are grouped such that a valid domain class can be assigned to the group: E.g., an attribute with CODED domain class can only be assigned to a group if all attributes of the group have CODED class.

Correspondence scores between source and target attribute groups that reflect their similarity are computed in the second phase. Note that only attribute groups with a class from the same hierarchy are compared. The similarity function used to determine the score depends on the domain class: E.g., STRING

attributes are compared using normalized edit distance (see Sec. 4.3.2). CANDIDATE KEY attributes are given a high score if their values in the duplicate tuples match, while FOREIGN KEY attribute require the use of Goodman and Kruskal's Lambda to determine a score. Twelve statistical similarity functions are defined for CODED values. Which function is applied depends on the concrete classes of the attribute groups. The result of the second phase is a matrix for each class hierarchy that describes the similarity of source and target attribute groups in that hierarchy.

In the third phase of the algorithm, attribute groups are assigned to each other such that each attribute group has at most one matching attribute group. The Hungarian algorithm is used to compute an assignment that maximizes the sum of the correspondence scores. The resulting alignment of attribute groups represents the matching between the two tables.

While the algorithm has shown to be successful in the experimental evaluation, it has several drawbacks. Firstly, the duplicate detection problem is largely ignored. The authors assume that attributes, which uniquely describe the real-world identity of the entity represented by a tuple, exist in both tables and have been manually matched. Without the correspondence between identifying attributes, the matcher is unable to detect duplicates. We point out that in many scenarios, there is no key attribute that describes the identity across databases. Secondly, the algorithm is restricted to single tables, which reduces its applicability in real-world scenarios. We believe that matching single tables is a relevant problem, but the majority of matching tasks involves complex schemata with several tables. Thirdly, the first phase of the algorithm creates overlapping groups, which can lead to unintended results in the third phase.

To illustrate the third problem, assume a table with attributes for contact ($C1$), which contains the name and address of a person, and signature (S), and another table with an attribute for contact ($C2$) and an attribute representing the degree of the person (D). Note that the respective contact attributes contain strings that are much longer than the values in S and D. All of those attributes have a domain class in the STRING hierarchy, and thus, can be combined. The matrix in Fig. 2.4 shows the resulting groups.

	$\{C2\}$	$\{D\}$	$\{C2, D\}$
$\{C1\}$	**0.9**	0.0	0.8
$\{S\}$	0.0	0.0	0.0
$\{C1, S\}$	0.8	0.0	**0.8**

Figure 2.4: The problem of overlapping attribute groups.

The matrix values represent correspondence scores. The bold numbers indicate the correspondences as determined by the Hungarian algorithm whose score is above the threshold of 0.7. The actual correspondence scores are fictitious, but closely reflect similarity scores determined by normalized edit distance: The score for $\{C1\}$ and $\{C2\}$ is 0.9, because the values perfectly match in most duplicates. In contrast, the correspondence scores involving S are 0 because S does

not have a matching partner. However, one notes that $\{C1, S\}$ and $\{C2, D\}$ have a very high correspondence score that is above the threshold although S and D unrelated. These two attribute groups have a high score because the contact values are much longer strings than the signature and degree, and thus, the similarity of $C1$ and $C2$ has a larger effect on the score than the dissimilarity of S and D.

The underlying problem is that attribute groups overlap. The Hungarian algorithm does not take that into consideration: Its goal is to find a best matching between two sets of elements (i.e., attribute groups). The structure of those elements is irrelevant. In Chap. 7 we present an alternative matching approach that avoids the problem of overlapping attribute groups.

2.3.5 Combining multiple matchers

The previous sections indicate that schema matching solutions have to overcome two general problems:

1. *Homonyms*: Semantically different attributes are described similarly. Homonyms can result in false correspondences.
2. *Synonyms*: Corresponding attributes are described differently. Synonyms can lead to missed correspondences.

The "description" of an attribute can be produced manually as metadata or be extracted from available data. Because the above two problems occur in both types of input, it is generally agreed that no single indicator, data or metadata, can produce an optimal result. Thus, several types of information or several schema matchers should be used in the matching process. *Hybrid matchers* exploit different matching approaches in a single algorithm. This approach has the advantage that its constituent matchers can cooperate closely, and thus, produce a result very quickly. A disadvantage is its inflexibility: One cannot another matching approach without rewriting the hybrid algorithm.

In contrast, *composite matchers* run each of their constituent matchers in isolation and merge their results to produce a final matching. Thus, it is possible to add additional matchers if required. In the following, we describe the general architecture of a composite matcher, which is similar to LSD [DDH01], COMA [DR02], COMA++ [ADMR05], and by Madhavan et al. [MBDH05].

The composite matcher architecture has three major components: a set of base matchers, a match combiner, and a constraint handler (Fig. 2.5). The matching process starts by extracting information from the source that is relevant to the base matchers, e.g., schema information, instance statistics, etc. The base matchers process the type of information they require to produce their output, which is a similarity matrix containing similarity scores for each attribute pair[1]. The similarity matrices are then combined by the match combiner, which produces a single similarity matrix. This matrix is then processed

[1] If a given schema matcher does not produce a similarity matrix but a set of correspondences, these correspondences can be used to create a matrix by setting scores for corresponding attributes to 1 and other scores to 0.

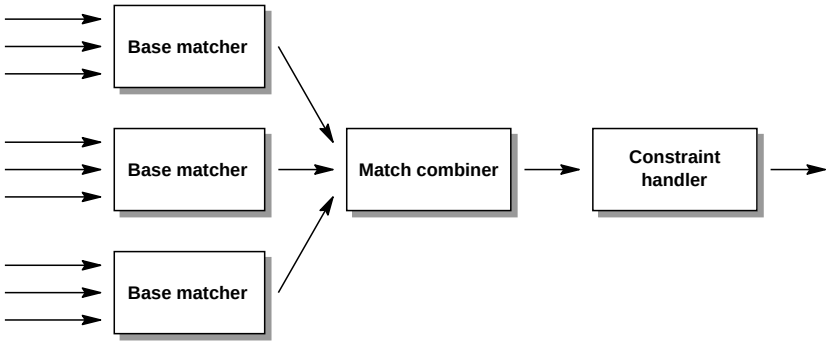


Figure 2.5: Architecture of a composite matcher.

by the constraint handler to produce a final matching. The constraints applied in this step can be domain-independent (e.g., the cardinality of the matching can be restricted to 1:1) or domain-dependent (e.g., each person only has one name).

2.4 Schema Mappings

The work described in this thesis is concerned with schema matching, i.e., the detection of attribute correspondences. However, to overcome schematic heterogeneity, those attribute correspondences have to be used to create schema mappings. In the following, we will discuss schema mappings and describe query discovery.

2.4.1 What is a Schema Mapping?

As pointed out in the previous chapter, schema mappings are required in various applications. Schema Mappings have been examined both from a theoretical and a practical perspective. A recent survey of theoretical work on schema mappings can be found in [Kol05]. In short, schema mappings are a form of tuple generating dependency between a source and a target schema. Languages for specifying such constraints and their properties is an active research area. However, in this thesis we resort to a more practical definition of mappings: A *schema mapping* is a declarative specification of a data transformation. In other words, a mapping describes how data conforming to the source schema can be transformed such that it satisfies the constraints of the target schema. Although a data transformation can also be achieved by a programm, we only consider declarative specifications here. Such specifications are mostly described as a query, whose language depends on the data model being used, e.g. MSL descriptions in TSIMMIS [GMPQ$^+$97] or conjunctive queries in Information Manifold [LRO96b].

Recall from Chap. 1 that schema mappings can be semi-automatically created in a two step process: (i) schema matching and (ii) query discovery. The schema matching problem and related work in that area are described above. In Sec. 2.4.2 we will examine query discovery solutions, which have been developed in the Clio project [HMH01].

2.4.2 Schema Mapping Generation

Early work on query discovery by Miller et al. considered the detection of mappings between relational schemata [MHH00]. They deviated slightly from the above two-step process by using value correspondences (as opposed to attribute correspondences) as input. A *value correspondence* v_i relates one or several source attributes with one target attribute (i.e., only 1:1 and n:1 relationships are considered). In contrast to attribute correspondences, it also contains a function f_i, which describes how values are to be transformed, and a filter describing which source values should be used. The query discovery process works in four steps, which are depicted in Fig. 2.6.

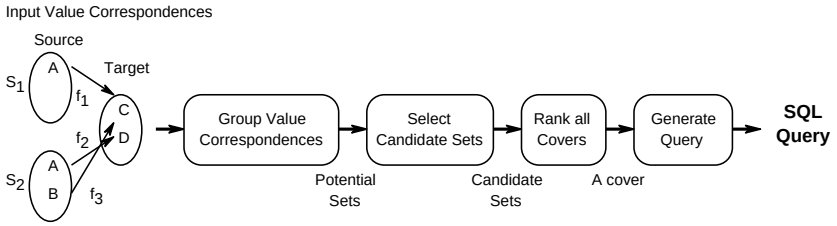


Figure 2.6: The query discovery process (Source: [MHH00]).

The goal of the query discovery algorithm is to construct a query for each target relation. If more than one target relation exists, the algorithm is invoked for each target relation. In the first phase of the query discovery process, potential candidate sets c_i are created, which contain a subset of the set of value correspondences $\mathcal{V}$ such that there is at most one correspondence for each target attribute in the candidate set. Each potential candidate set represents one possible way of mapping the attributes in target relation T. These potential candidate sets do not need to be complete, i.e., a single set does not need to contain a correspondence for every target attribute. The example in Fig. 2.6 has candidate sets $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\{v_1\}$, $\{v_2\}$, and $\{v_3\}$.

In the second phase, potential candidate sets that cannot be mapped into a good query are pruned. In particular, a potential candidate set cannot be mapped into a query if its correspondences involve multiple source relations, and no join path connecting those source relations exists. Various techniques are used to determine meaningful join paths, e.g., exploiting the data dictionary or schema discovery. The remaining sets are called candidate sets. In the example, we assume that source tables S_1 and S_2 can be joined, and thus, all potential candidate sets are candidate sets.

The goal of the third phase is to find a cover Γ, which is a set of candidate sets that cover all value correspondences, i.e., each value correspondence must appear in at least one candidate set. In the above example, possible covers include $\Gamma_1 = \{\{v_1\}, \{v_2, v_3\}\}$ and $\Gamma_2 = \{\{v_1, v_2\}, \{v_2, v_3\}\}$ since all defined value correspondences appear at least once. If there are several possible covers, Clio prefers the one which contains fewer candidate sets, i.e., one that produces a simpler mapping. In the case of a draw, covers that produce less null values in the target are used.

In the final step, a query is constructed based on the preferred cover. Because every candidate set in the cover represents a possible mapping, the produced query is the union of the queries described by a candidate set. Given that Γ_2 is chosen as he best cover, the resulting query is:

```
CREATE VIEW T(C,D) AS
  SELECT f1(S1.A), f2(S2.A)
  FROM S1,S2
  WHERE S1.K = S2.FK
UNION
  SELECT f3(S2.B), f2(S2.A)
  FROM S2
```

Miller et al. also present a incremental query discovery algorithm, which is not shown here. As described above, query discovery is a semi-automatic process: The user can interact by adding correspondences, running the algorithm, and then adding or removing correspondences again. The incremental algorithm takes user interaction into consideration and only considers those correspondences that have been added or removed by the user after the previous run of the algorithm. As a result, the performance of the algorithm increases because previous decisions are reused.

In subsequent work, Popa et al. developed an algorithm for query discovery based on the nested relational model, which is applicable both for relational and XML data [PVM+02]. In contrast to Miller et al., they use attribute correspondences as described in Sec. 2.2 as input. Consequently, they are able to include any existing schema matcher into their framework as long as the matcher produces attribute correspondences. They also consider constraints on the target schema, which were ignored in the work by Miller et al. Thus, they try to interpret the attribute correspondences in a way that is consistent with the semantics of both the source and the target schema. In a process called *semantic translation*, the semantics encoded in the schemata is used to construct a *logical mapping*. This step involves grouping of correspondences such that they do not violate any constraints. The second phase is data translation, where issues of generating data for target attributes that do not participate in a correspondence and grouping of nested elements are addressed. For the sake of brevity, we omit details of the algorithm and refer the interested reader to [PVM+02].

Chapter 3

From Duplicates To Schema Matching

Duplicates provide helpful information to find attribute correspondences where schema-based or vertical instance-based matchers fail. In this chapter we describe the duplicate-based matching approach, describe the problem of detecting duplicates, and review related work in that area.

3.1 Why Duplicates Can Help in Schema Matching

In Chap. 1 we claimed that extensional overlap, i.e., the existence of duplicates, can help in the process of detecting a schema matching. Existing solutions, which are reviewed in Sec. 2.3.4, suggest the same. In particular, we believe that duplicate-based approaches succeed where others fail, e.g., when attribute names are 'cryptic' and semantically different attributes have similar features.

To motivate the duplicate-based approach, we use the example in Fig. 3.1, which is an adaptation of the example used in [BN05]. The source relation R contains attributes $FirstName$, $LastName$, Sex, $Phone$, and Fax, whose semantics can easily be determined by a human user. The schema of S contains less readable attribute names: LN for last name, Acc for user account, Tel for phone number, and OS for operating system. The correct matching contains the correspondences $(\{LastName\}, \{LN\})$ and $(\{Phone\}, \{Tel\})$.

Schema-based schema matchers are very unlikely to produce good results due to the use of acronyms (e.g., LN) and different abbreviations (Tel instead of $Phone$) in relation S. Usually, string similarity measures are used in schema-based approaches to determine the similarity of attribute names, and attributes with highly similar names are assumed to correspond. In this scenario, names of matching attributes do not have a higher similarity than names of unrelated attributes. Consequently, schema-based matchers will fail.

$\boldsymbol{R}$	$FirstName$	$LastName$	Sex	$Phone$	Fax
r_1	John	Doe	m	(408) 7573339	(408) 7573338
r_2	Joe	Smith	m	(249) 3615616	(249) 2342366
r_3	Suzy	Klein	f	(358) 2436321	(358) 2436321
r_4	Sam	Adams	m	(541) 8127100	(541) 8121164
r_5	Mark	Spitz	m	(901) 8319311	(901) 8612382
r_6	Jim	Beam	$\perp$	(782) 1238957	(781) 1883744
r_7	Kate	Moss	f	(124) 9654565	$\perp$
r_8	Sam	Wong	f	(124) 4955670	(999) 9999999
r_9	John	Dean	m	(369) 3663624	(367) 3663625

$\boldsymbol{S}$	LN	Acc	Tel	OS
s_1	Douglas	jdouglas	(408) 9182043	XP
s_2	Dean	jd	(369) 3663624	XP
s_3	Klein	littlesue	(358) 2436321	UNIX
s_4	Adams	sam	(541) 8127100	W2000
s_5	Wong	kate	(923) 6363443	Linux
s_6	Kurz	itsme	$\perp$	UNIX

Figure 3.1: Relations R and S with intensional and extensional overlap.

Also note that there are cases in Web data integration where attribute names are incorrect or unavailable. Assume that the web site of a retailer (e.g., Amazon) has been crawled and a large collection of Web pages containing information about books has been downloaded. Current wrapper generation techniques are able to extract structural information from such a large corpus of pages [LRNdST02]. However, the semantics of the structural elements cannot be determined. To give the field of a web page a meaning, some approaches try to extract labels from Web pages [ACMM03]. However, these approaches do not always produce a good result. To complicate the issue, in several cases labels are missing or even misleading. Fig. 3.2 shows product details of the book "Readings in Database Systems" by Joseph M. Hellerstein and Michael Stonebraker. One might assume that the text in front of the colon, which is distinctively written in bold face, represents the attribute label, while the following text is the actual data. But that is only partially true. In the first line of the details, 'Paperback' does not represent an attribute name, but the format of the book. The value after the colon represents the number of pages. The following line contains information about the edition and publishing date without a corresponding label.

Instance-based schema matchers are not affected by the issues described above, because they ignore schema information and exploit available data. Nevertheless, instance-based matchers that perform vertical matching will have difficulties with the scenario in Fig. 3.1. They might be able to determine the cor-

Figure 3.2: Product details from `www.amazon.com`.

respondence $(\{LastName\}, \{LN\})$, depending on which features they extract. Determining the second correspondence $(\{Phone\}, \{Tel\})$ is more difficult, because Tel is also very similar to Fax. Even for a human user it is impossible to distinguish phone numbers from fax numbers just by looking at each attribute in isolation. In general, vertical matchers fail to distinguish attributes that have similar features but are semantically different: They cannot tell phone numbers from fax numbers (Fig. 3.1) or place of residence from birthplace (Fig. 2.3).

Close examination of the above scenario reveals an extensional overlap: Tuple pairs (r_3, s_3), (r_4, s_4), and (r_9, s_2) are fuzzy duplicates, because both tuples in each pair represent the same person. We claim that such duplicates can be used to increase matching accuracy. Instance-based matchers which perform horizontal matching follow that idea by detecting duplicates and extracting attribute correspondences from them. Each of the duplicates indicate a certain matching, which can be determined by comparing attributes values. E.g., (r_4, s_4) has similar attribute values that indicate correspondences $(\{LastName\}, \{LN\})$ and $(\{Phone\}, \{Tel\})$. Some duplicates might suggest false correspondences, but by aggregating the results of several duplicates, the effect of a few false indications diminishes.

In contrast to vertical matchers, duplicate-based matchers are able to distinguish phone number from fax number in the above example: By comparing the tuple pairs determined to be duplicates, we can see that values for $Phone$ and Tel are always equal, and thus, can deduce that these attributes correspond. Similarly, when comparing duplicates 'John Doe' and 'Sam Adams' in Fig. 2.3 on page 30, we can see that attributes $Birthplace$ and POB match.

3.2 The DUMAS Approach

As described in the previous section, duplicates provide information that can be used to detect correspondences. We now sketch out an algorithm that exploits extensional overlap. Consistent with related work on horizontal matching (Sec. 2.3.4), the general algorithm for duplicate matching of schemata (DUMAS)

proceeds in two steps:

1. Duplicate detection,

2. Matching.

The goal of the first step is to detect fuzzy duplicates, which will be used in the second step. While it might not be necessary to detect all duplicates, the tuple pairs identified as duplicates should in fact be duplicates, i.e., false duplicates must be avoided. In the above example, the perfect result is the set of tuple pairs $\{(r_3, s_3), (r_4, s_4), (r_9, s_2)\}$. The duplicate detection step will be discussed in detail in the following sections. In particular, we define what a duplicate is, and examine related work in duplicate detection. We show that existing solutions are not applicable, and define the problem of detecting duplicates for schema matching in Sec. 3.5.

Consistent with the definition in Sec 2.2, the goal of the matching step is to produce a set of attribute correspondences. The duplicates detected in the previous step are to be used here as input. Other information, such as metadata or documentation, will be ignored. Intuitively, each of the tuple pairs can be seen as an indicator for the correct schema matching: If the value of an attribute in one tuple is the same as or very similar to the value of an attribute in the other tuple, we could deduce that those two attributes are related.

While this is usually true, two attribute values can coincidentally match, thus misleading schema matching, or several attributes are very similar, resulting in uncertainty. Fig. 3.3 shows tuples r_4 and s_4, with highly similar attribute values connected by arrows. The two solid arrows denote true correspondences, while the dashed arrow shows that the duplicate has similar values in attributes that do not match. Such a case can mislead schema matching: If only this duplicate were used, the result would include a false correspondence. The case of several possible matches occurs in tuple pair (r_3, s_3), because Suzy Klein's phone number and fax number are equal. Hence it is impossible to decide which attribute corresponds to Tel when using only duplicate (r_3, s_3). The solution to both problems is the same: Use several duplicates instead of just one, thus, reducing the effect of a few coincidental value similarities or uncertain matches. Consequently, one must aggregate the different indications of several duplicates to produce a final matching. A solution to this problem is presented in Chap. 5.

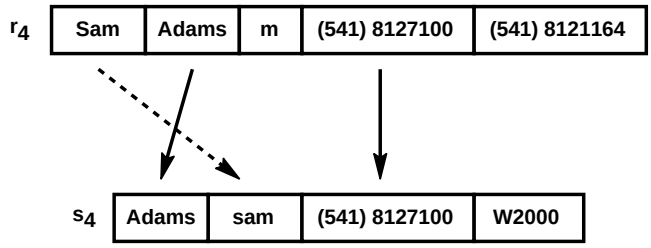


Figure 3.3: Misleading similarity of attribute values.

Note that this two-step process can also be made iterative (Fig. 3.4). After detecting a few correspondences in the matching step, it should be determined if this matching can be trusted. If the correspondences are certain, they should be presented to the user. If some correspondences are uncertain. The schema matching process should resume. However, in the second iteration the correspondences that are trusted can be used in the duplicate detection step.

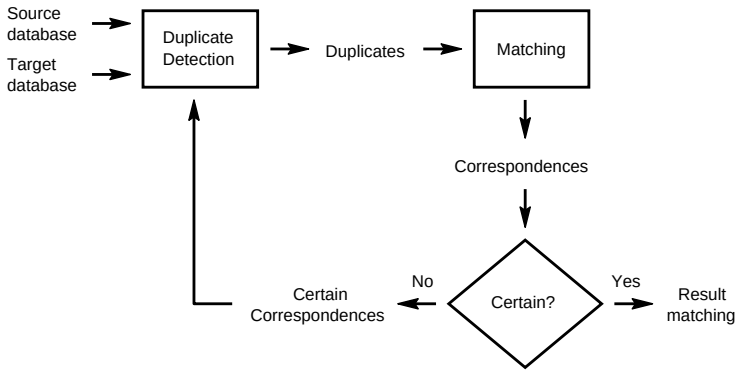


Figure 3.4: The duplicate-based schema matching process

3.3 Duplicates and Duplicate Detection

The problem of detecting duplicates has been studied for several decades under various names, including record linkage, data cleansing, and entity identification [RD00]. In all but a few publications, the described goal is to find duplicates in a single table or two matching tables. The latter implies that it has been established that the two tables represent the same entity type. A *fuzzy duplicate*, or duplicate for short, is defined to be different representations of the same real-world entity.

Duplicates are called fuzzy because they are not exact copies of one another. Even when schematic heterogeneity is not an issue, e.g., when looking for duplicates in a single table, the same information can be represented in different ways. This can be done deliberately when no standard representations exist: "Technische Universität Berlin" can be abbreviated "TU Berlin" or even "TUB", a second given name of a person can be fully written or be replaced by a middle initial, and different scales for certain measures can be used in different database entries. In addition to deliberately different representation, misspellings and missing information make duplicate detection a hard problem.

Duplicate detection is the process of searching for fuzzy duplicates, which can be represented as tuple pairs. More formally, the goal of *duplicate detection in a single relation* R is to find tuple pairs $(r_i, r_j) \in R \times R$, where r_i and r_j represent the same real-world entity. We say that r_j is a duplicate or duplicate record of r_i. Because only a single table is considered, all tuples are equally

structured, and thus, there is an inherent matching relating each attribute to itself.

The problem of *duplicate detection in aligned relations* is similarly defined: Given to tables R and S and the schema matching $\mathcal{M}$ between R and S, find all tuple pairs $(r_i, s_j) \in R \times S$, such that r_i and s_j represent the same real world entity. The correspondences are required, because a duplicate detection algorithm needs to compare values of related attributes. In contrast to the problem of duplicate detection in a single relation, schematic heterogeneity may occur, i.e., the schemata of the relations can have a different structure, and not all attributes must have a corresponding attribute in the other schema. Note that both problem definitions do not contain cardinality restrictions: A database entry can have zero, one, or several duplicate records.

The *accuracy* of the duplicate detection result is measured in terms of *precision* and *recall*, which are computed as follows:

$$Precision = \frac{|D \cap R|}{|R|} \qquad Recall = \frac{|D \cap R|}{|D|}$$

where D is the set of true duplicates and R is the set of retrieved duplicates. A good precision is achieved if only few false duplicates are in the result set. On the other hand, the user wants the duplicate detection algorithm to find all or most of the duplicates, which is measured as recall.

In most cases, the user has to make a tradeoff decision: By allowing more errors, one will retrieve more tuple pairs and increase recall, but also increase the chance of detecting false duplicates. Consequently, recall is likely to increase, but as more false duplicates enter the result set, precision will drop.

Beside the quality measures precision and recall, *efficiency* of duplicate detection is a major issue. If a single table contains n tuples, then there are n^2 possible tuple pairs which have to be compared. Analogously, if a source table contains m tuples and a target table contains n tuples, there are $m \cdot n$ tuple pairs which have to be checked in the case of duplicate detection in aligned relations. Because that many comparisons are infeasible in all but the smallest relations, methods for reducing the number of tuple comparisons have to be applied.

3.4 Related Work on Duplicate Detection

As discussed above, duplicate detection algorithms must produce a result with both (i) high accuracy and (ii) good efficiency. In the following, we review related work on duplicate detection and describe their approach to achieve these two goals.

3.4.1 Record Linkage

Record linkage is a statistical approach that uses user-provided samples of duplicate tuple pairs and non-duplicate tuple pairs to derive duplicate detection rules [NKAJ59, FS69, Win95, EVE02]. The record linkage method classifies

tuple pairs into 'duplicate' (A_1), 'non-duplicate' (A_3), and 'possible duplicate' (A_2) based on the comparison vector $\gamma = (\gamma_1, \ldots, \gamma_n)$ for each tuple pair. The method to create the comparison vector is not defined by record linkage. Usually, one compares values of the same attribute using an attribute-specific comparison function to determine a score for each γ_i: E.g., in a table describing books, one could determine a comparison score for the ISBN attribute with the following function [NL00]:

$$f_1(ISBN_1, ISBN_2) := \begin{cases} 0 : ISBN_1 = ISBN_2 \\ 1 : ISBN_1, ISBN_2 \text{ are missing} \\ 2 : \text{otherwise.} \end{cases}$$

To determine if two tuples match, the likelihood ratio λ of the tuple pair must be computed, which is defined as:

$$\lambda = \lambda(\gamma) = \frac{P(\gamma|D)}{P(\gamma|N)} \tag{3.1}$$

where D is the set of duplicate tuple pairs and N is the set of non-duplicates. The conditional probabilities $P(\gamma|D)$ and $P(\gamma|N)$ can be estimated from the user-provided samples. We point out that in practice the computation of the probabilities is simplified by assuming that the elements of γ are statistically independent.

A lower bound γ_l and an upper bound γ_u is also derived from the samples [FS69]. Using those bounds, one can determine the class of a given tuple pair using the decision function $\delta(\lambda)$:

$$\delta(\lambda) := \begin{cases} A_1 : \lambda > \lambda_u \\ A_2 : \lambda_l \leq \lambda \leq \lambda_u \\ A_3 : \lambda < \lambda_l. \end{cases} \tag{3.2}$$

To make record linkage scalable, the number of tuple comparisons is decreased in a process called *blocking*: Domain-specific criteria are used to partition the set of tuples into blocks, and only tuples within a single block are compared [FS69]. All other tuple pairs are implicitly considered non-duplicates.

3.4.2 The Sorted Neighborhood Method

The *Sorted Neighborhood Method* (SNM) is a duplicate detection approach that has been developed in the database community [HS95, HS98]. It is based on the process used by database management systems to remove exact duplicates: Instead of comparing all tuples, the content of a relation is sorted by tuple content. This process brings equal tuples together, i.e., tuples with the same content are neighbors. Consequently, only neighboring tuples need to be compared to find exact duplicates.

Because of the previously discussed data quality issues, the procedure to find exact duplicates is not applicable for fuzzy duplicates: If the same piece

of information is represented in different ways, sorting a table might place two tuples representing the same real-world entity in different regions of the table. The sorted neighborhood method tackles that problem in three steps:

1. *Create key*: Compute a key for each record in the relation by extracting relevant attribute values or portions of attribute values.
2. *Sort data*: Sort the records in the table using the key of step 1.
3. *Merge*: Move a fixed size window through the ordered list of records limiting the comparisons for matching records to those records in the window.

The goal of the first step is to define a key that places duplicate tuples close to each other in the sorting phase even when the data suffers from data quality issues. The key is usually comprised of several attribute values or portions thereof: E.g., a good key for relation R in Fig. 3.1 could be constructed from the first two letters of the $FirstName$ value and the first two letters of the $LastName$ value. The definition of a reasonable key requires the user to have knowledge of the domain and of typical errors contained in the data set. The creation of the above key could be driven by the experience that errors in names usually occur at the end of the string.

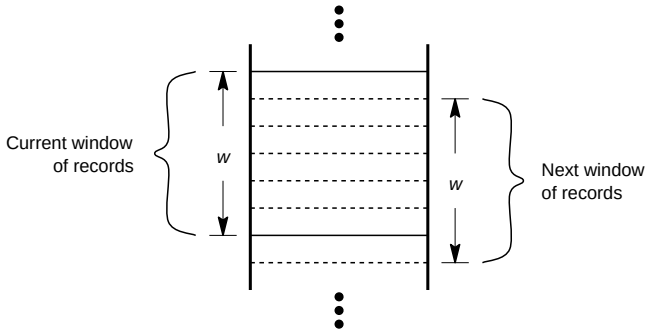


Figure 3.5: Sliding a window through a sorted table (Source: [HS95]).

The key defined in step 1 is used in the second step to sort the table. The sorted table is searched for duplicates using the 'sliding window' approach: A window of size w (i.e., with a capacity of w records) is moved through the sorted table. Every time the window has been moved one tuple ahead, the tuple that has entered the window is compared with the other $w - 1$ tuples presently inside the window. Obviously, that approach decreases the number of tuple comparisons: Instead of comparing all tuple pairs, which has a complexity of $O(N^2)$, comparing only tuples within the sliding window has a time complexity of $O(wN)$. This justifies the additional effort for creating a key for each tuple ($O(N)$) and sorting the table ($O(N \log N)$).

Note that the process to find exact duplicates is a special case of the sorted neighborhood method, where the sort key is the whole tuple content and the

window size is 2. Comparing tuples to find exact duplicates is straightforward: The contents of the attributes must be equal in exact duplicates. When searching for fuzzy duplicates, data quality issues must be taken into consideration: Hernández and Stolfo propose to compare tuples using user-defined rules [HS95], e.g.

```
Given two records, r1 and r2
IF the last name of r1 equals the last name of r2,
   AND the first names differ slightly,
   AND the address of r1 equals the address of r2
THEN
   r1 is equivalent to r2.
```

The implementation of "differ slightly" is based on a string similarity function, i.e., two first names differ slightly if their string similarity is above a given threshold. Many rules similar to the above one are usually defined to allow for different kinds of errors. However, the sorted neighborhood method is not limited to such rules: The primary purpose of the sliding window technique is to reduce the number of tuple comparisons, and various tuple comparison measures can be used to detect duplicates.

The duplicate detection rules should be defined such that only true duplicates are considered as duplicates, i.e., false duplicates must be avoided. However, even with a good set of rules it is possible that duplicates are missed because they are placed far apart in the sorting phase. A possible measure to avoid missed duplicates is to increase the size of the window w, and thus, increase the number of tuple comparisons. While this action improves the accuracy of duplicate detection, it also degrades performance. Hernández and Stolfo have shown that the *multi-pass approach* results in much better duplicate detection accuracy: Instead of increasing w, the sorted neighborhood method is applied several times using varying keys. Afterwards, the transitive closure for the duplicates is computed, i.e., if (r_1, r_2) is detected as a duplicate in one run and (r_2, r_3) is detected in another run, then (r_1, r_3) must be considered a duplicate, too.

3.4.3 Other Duplicate Detection Approaches

Various duplicate detection methods that use domain-independent string similarity measures have been proposed. Monge and Elkan compare attribute values using the Smith-Waterman algorithm, which is a variant of the Levenshtein edit distance that applies different weights for different characters [ME96, ME97]. Chaudhuri et al. define a fuzzy similarity function, which determines the similarity of two attribute values based on the operations needed two transform one value into the other [CGGM03]. Three operations are defined: token replacement, token insertion, and token deletion. The cost of token insertion and token deletion depends on the inverse document frequency of the inserted or deleted token, while token replacement also takes the edit distance into consideration[1].

[1] See Sec. 4.3.2 for a description of edit distance and inverse document frequency.

Chaudhuri et al. also describe a probabilistic procedure to reduce the number of tuple comparisons when searching for duplicates.

The string similarity measures applied in the above duplicate detection systems are domain-independent to facilitate their use in different scenarios. Recent research has shown that machine learning techniques can be used to automatically adapt these measures to a given domain in order to improve their accuracy. Bilenko and Mooney describe the MARLIN system, which exploits user-provided samples to learn domain-specific costs for edit distance and TFIDF cosine similarity [BM03]. Experiments show that the accuracy of TFIDF does not always increase, but the edit distance costs learned by Expectation Maximization (EM) always improve the result. In a similar fashion, Tejada et al. use decision trees to learn weights for various string transformation functions [TKM02].

The duplicate detection methods described here only take corresponding attributes in consideration when determining if tuples represent the same entity. A notable exception is PROM, which also exploits information represented in only a single source [DLLH03]. After possible duplicates have been detected on the basis of shared attributes, PROM performs a sanity check. The sanity check uses various application-specific constraints, which can also relate to unmatched attributes: E.g., assume two tables describing people, where the income is only represented in the first table, while the age of a person is only shown in the second table. A reasonable rule for this scenario would be "If a person's age is less than 18, then the income cannot be larger than USD 10,000."

We point out that duplicates are also an issue outside of the relational world. Hence, duplicate detection algorithms have also been proposed for XML [WN05], data warehouses [ACG02], and spatial data [BKSS04].

The algorithms described in this section are designed to detect duplicates with high precision and high recall, i.e., the goal is to find all duplicates and to not produce false duplicates. To achieve that goal, they require a lot of information by the user: attribute correspondences, sample duplicates and non-duplicates, duplicate detection rules, etc. As will be shown in the following, finding duplicates for schema matching has relaxed quality goals, but cannot expect as much user input as classic duplicate detection approaches.

3.5 Finding Duplicates For Schema Matching

The duplicate detection approaches described above use different techniques, but all of them have one property in common: They require the complete schema matching to be known, so they can restrict attribute comparisons to related attributes. This is contrary to our problem of detecting duplicates for schema matching, where the goal is to establish a matching *after* duplicate detection. This section discusses the problem of detecting duplicates without known correspondences.

3.5.1 Single-Table Duplicates

When matching two tables using our duplicate-based approach, the problem definitions in Sec. 3.3 cannot be applied. Instead, the problem of *detecting duplicates in unaligned relations* has to be solved, because attribute correspondences do not exist in the first step. The definition of this problem is equivalent to the definition of duplicate detection in aligned relations, except that no correspondences are known. We call the detected duplicates *single-table duplicates* because their tuples span only a single table.

Finding duplicates in unaligned relations is much harder than in aligned tables. Fortunately, quality requirement are less strict. Recall that in the case of duplicate detection in aligned relations, the goal is to detect duplicates with high recall and high precision, i.e., all or most duplicates should be detected and only few false duplicates should enter the result set, respectively. When looking for duplicates in the DUMAS approach, high recall is irrelevant: It is not necessary to detect *all* duplicates, but only as many as required for schema matching. On the other hand, precision is still an issue, because false duplicates may corrupt the schema matching constructed in the matching step.

3.5.2 Multi-Table Duplicates

Up to this point, the objects representing real-world entities were assumed to be structured as a single table. However, the ultimate goal of schema matching is to find correspondences between complex schemata consisting of multiple tables. When trying to match complex schemata, one has to consider issues related to schematic heterogeneity. In particular, one has to be aware that information can be structured in different ways: E.g., an entity type represented as a single table in the source schema can be structured as several tables in the target schema (e.g., due to normalization). Fig. 3.6 depicts such a case: The source schema contains table $CourseByTerm$ describing a course with a course id (CID) and a title, which is taught in a given term by a faculty member. In the target schema, this table is normalized into three tables: One table for the courses, one table for the faculty members (Fac), and a table By that has foreign keys to the other two tables (CID and FID) and an attribute for the term in which a course was taught. The semantics of the source table is similar to the semantics of the table that can be created by joining the three target tables. In a similar fashion, a complex entity type can be described in several tables in the source schema and several tables in the target schema.

In the following, we assume that the tables representing an entity type can be joined together in a meaningful ways. This assumption holds in most cases, because tables related to a single concept are usually connected in a schema, e.g., $Course$, By, and Fac in Fig. 3.6. It is well known that the relational algebra is closed, i.e., relational operators take relations as input and produce relations. We exploit this property and reduce the problem of *detecting multi-table duplicates in unaligned schemata* to the problem of detecting duplicates in unaligned relations: Given that an entity type is represented as relations

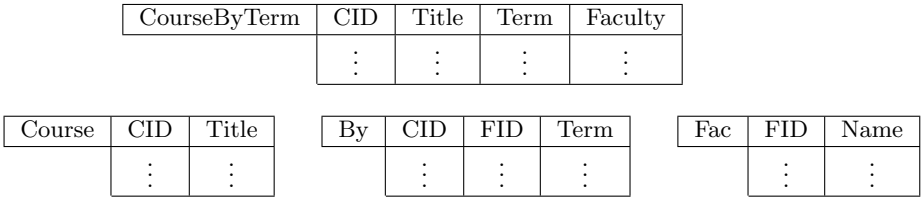


Figure 3.6: Multi-Table Duplicates

$R_1, \ldots, R_m$ in the source schema and relations $S_1, \ldots, S_n$ in the target schema, find duplicates in the unaligned tables $R_{join} = R_1 \bowtie \ldots \bowtie R_m$ and $S_{join} = S_1 \bowtie \ldots \bowtie S_n$.

3.5.3 Related Work

A number of duplicate-based schema matchers were examined in Sec. 2.3.4, and it was shown that they either ignore the problem of duplicate detection or work only in very restricted scenarios: iMap requires the user to provide duplicates, Chua et al. assume that global identifiers exists, and the Internet Learning Agent reduces the duplicate detection problem to simple keyword search, which is insufficient in most scenarios. In fact, we are not aware of any work that fully considers the problem of duplicate detection in unaligned relations.

Finding duplicates that span multiple tables is also a novel problem. As described in Sec. 2.3.4, classic duplicate detection procedures are based on single tables. If correspondences are known and the semantics of the tables are understood by the user, these duplicate detection algorithms can be applied on existing tables or tables that are created by joining certain tables: E.g., if the user knew that the join of tables $Course$, By, and Fac in Fig. 3.6 has the same semantics as table $CourseByTerm$ and the correspondences were known, then one of the discussed duplicate detection procedure could be applied on $Course \bowtie By \bowtie Fac$ and $CourseByTerm$. Unfortunately, in our scenario neither the correspondences nor the semantics of the tables are known, and the problem of multi-table duplicates in unaligned relations has not been considered before.

In the following chapters we present solutions to the problems described here and in the previous chapter. In Chap. 4 a solution to the duplicate detection problem in unaligned relations is presented. In the following Chap. 5 we show how to extract attribute correspondences from the detected duplicates. The duplicate detection procedure in combination with the matching step constitute the DUMAS *table matcher*. Chap. 6 describes the DUMAS *schema matcher*, which detects multi-table duplicates to find correspondences between two complex schemata. The DUMAS *complex matcher*, which is able to extract complex correspondences from duplicates, is discussed in Chap. 7.

Part II

The DUMAS Table Matcher

Chapter 4

The Duplicate Detection Step: Finding Duplicates in Unaligned Relations

This chapter presents an algorithm that solves the problem of duplicate detection in unaligned relations, which is more difficult than the common duplicate detection problem, where attribute correspondences are known. We begin by examining problems that stem from the lack of match information. Our solution to this problem is to consider each tuple as a single string, and to consider the k most similar tuple pairs as duplicates. After defining a tuple similarity measure, an algorithm that efficiently finds the most similar tuple pairs is described. The duplicate detection procedure is experimentally evaluated using both real-world and synthetic data. Note that the duplicate detection procedure presented in this chapter combined with the matching step described in the following chapter constitute the *DUMAS table matcher*, which was originally described in [BN05].

4.1 Duplicate Detection Without Known Correspondences

Recall from Sec. 3.4 that duplicate detection methods essentially have to answer to questions:

1. Given two tuples r and s, is (r, s) a duplicate?
2. What tuples need to be compared in order to find all duplicates?

The answer to the first question determines the accuracy of duplicate detection, while the answer to the second question affects the performance.

Most work on duplicate detection requires existing correspondences to be known. This is necessary to ensure that only related (i.e., corresponding) attributes are compared. In addition, the user needs to have detailed knowledge

of the application domain and the semantics of the attributes. Such knowledge is required to either manually define duplicate detection rules or to extract a few duplicates and non-duplicates, which are used to train the duplicate detection algorithm (see Sec. 3.4). Note that when the attribute semantics of both source and target schema are understood, attribute correspondences are inherently known, because attributes with the same semantics correspond. Knowledge about the semantics of the attributes is also required when determining which tuples need to be compared: E.g., the sorted neighborhood method requires the user to specify criteria for sorting, which strongly depend on the application domain.

Consider tuples r_4 and s_4 from the example in Fig. 3.1 on page 38, which represent the same real-world entity. If the correct correspondences ($\{LastName\}$, $\{LN\}$) and ($\{Phone\}, \{Tel\}$) are given as input, a user who is familiar with the application domain and attribute semantics can easily see that the two tuples are probably duplicates: Given that the last names ("Adams") and phone number ("(541)8127100)") are equal, it is very likely that the two tuples represent the same real-world entity.

r_4	Sam	Adams	m	(541) 8127100	(541) 8121164

s_4	Adams	sam	(541) 8127100	W2000

Figure 4.1: Duplicate tuples.

Finding duplicates in non-aligned relations is more difficult. Fig. 4.1 depicts those two tuples without attribute names and correspondences. This is an example for the kind of input our duplicate detection procedure has to handle. When deciding if those two tuples are duplicates without knowing the actual correspondences, it is very hard to decide which attributes to compare. Obviously, comparing values of attributes at the same position in the tuples is unreasonable: Structural heterogeneity, which is to be resolved by establishing a mapping between the two schemata, gives rise to the problem that semantically related attributes are at different positions, and some attributes only appear in one of the two schemata. Hence, when comparing two arbitrary tuples, it is very hard, if not impossible, two determine if they represent the same entity.

However, in contrast to related work on duplicate detection in aligned relations, our goal is not to develop a general-purpose duplicate detection procedure that finds *all* duplicates. Instead, only as many duplicates as required for schema matching have to be found. In other words, duplicate detection does not need to achieve high recall. This facilitates the modelling of the duplicate detection problem as the search for the k most similar tuple pairs, where k is the number of tuple pairs required for schema matching. Sec. 4.2 describes the assumptions that are made when following this approach. To determine the similarity of two

tuples, a tuple similarity measure is defined in Sec. 4.3. The choice of similarity measure also affects the answer to the second question. Sec. 4.4 describes an algorithm that is able to find the k duplicates with a number of tuple comparisons that is significantly smaller than the overall number of tuple pairs.

4.2 Duplicate Detection as Top-k Search

Before describing the DUMAS duplicate detection algorithm, two underlying prerequisites have to be made explicit: The two relations that are given as input to the duplicate detection procedure must

1. represent the same entity type,
2. contain duplicates.

The first prerequisite simply states that some correspondences exist between the two tables. If the tables are not related, one would not need to look for attribute correspondences. However, it is necessary to make this assumption because in some cases some real-world entities appear in a database in different roles: E.g., if tables R and S in Fig. 3.1 represented customers of a department store and account holders in a computer network, respectively, detected duplicates might indicate correspondences between R and S although they are not related. However, note that this is a rare case, which strongly depends on the intention of the user: If the goal was the integration of customer and account data, the detected correspondences would be correct.

In order for any duplicate-based schema matcher to work, actual duplicates must exist, which is stated in the second prerequisite. As we will see in the following, the described duplicate detection procedure has no direct means to determine if two tuples are duplicates. Instead, it picks some tuple pairs which are more likely to be duplicates than other tuple pairs. In other words, the algorithm *always* finds a few tuple pairs as duplicates, even when no duplicates exist.

Given that duplicates exist and given the requirement to detect only a few duplicates, the algorithm does not need to determine if two tuples represent the same real-world entity. Instead, it only has to find some tuple pairs that are most likely to be duplicates. This decision is based on the following *duplicate detection assumption*: A tuple pair (r_i, s_j) is more likely a duplicate than a tuple pair (r_k, s_l), if r_i is more similar to s_j than r_k is to s_l. This assumption does not have to hold in the entire space of tuple pairs in order to produce a good result: Because only a few duplicates are required, in only a small fraction of the space of all tuple pairs the distinction between duplicates and non-duplicates needs to be clear. In the DUMAS duplicate detection algorithm, we assume that the most similar tuple pairs are true duplicates, and return the k most similar tuple pairs if k duplicates are required for schema matching. This assumption is very intuitive: If there are any duplicates, then the most similar tuples most likely are true duplicates. What is meant by 'similar' in the context of duplicate detection in unaligned relations is discussed in the following section.

Note that the duplicate detection assumption can also be found in other duplicate detection procedures. Recall from Sec. 3.4 that record linkage uses two thresholds to classify tuple pairs into three classes duplicates, possible duplicates, and non-duplicates. Although there is a class of possible duplicates where the distinction between duplicate and non-duplicate is not clear, and thus, clerical review is required, tuple pairs which pass the higher threshold are always considered duplicates.

4.3 The Tuple Similarity Measure

Before defining a tuple similarity measure, inherent problems of duplicate detection in unaligned relations need to identified. After a review of related work, the tuple similarity measure is defined.

4.3.1 Inherent Problems

As stated above, the k most similar tuple pairs are to be returned by the DUMAS duplicate detector. To do so, a reasonable tuple similarity measure *tupsim* has to be defined, which produces high similarity scores for tuple pairs that are duplicates and lower scores for non-duplicates. In addition, it has to overcome several problems:

1. *Unknown schema alignment*: It is unclear which field in one tuple to compare with which field in the other.

2. *Partial schema overlap*: Not all fields in one tuple necessarily have a matching partner in the other. With only few corresponding attributes, the similarity of two tuples is typically low.

3. *Unknown attribute semantics*: We cannot make use of domain knowledge to formulate an effective comparison measure. Common duplicate detection methods use manually or statistically created rules that are based on the similarity of certain corresponding attributes: E.g., the sorted neighborhood method requires an domain expert to formulate duplicate detection rules that are applied to determine if two tuples represent the same real-world entity. When the semantics of the attributes are not known, meaningful rules cannot be created. Instead, a comparison measure that is independent of the fields' semantics and the application domain must be applied.

4. *Misleading value similarities*: In many cases, attribute values of tuples that do not represent the same real-world entity are similar. We distinguish two cases:

   (a) *Corresponding attributes*: Two non-duplicate tuples have the same or highly similar value in two attributes which do correspond. If those two tuples were considered duplicates, and thus, used in the

matching step, some correspondences will not be detected because the tuples are false duplicates. E.g., if the birthplace of two persons matches, it cannot be deduced that they are duplicates, because very many people might have been born in a given place. Using such non-duplicate tuples that have a similar birthplace would result in missed matches between other attributes.

(b) *Non-corresponding attributes*: Two values can be similar even when their attributes do not match, e.g., because the attributes have the same domain. Consider tuples r_7 and s_5 in Figure 3.1: Both tuples have an attribute value 'kate', but are not duplicates. Without knowledge of the correct attribute matches, such a value match can mislead duplicate detection. This problem is closely related to Problem 1.

Problem 3 is tackled by defining a domain-independent tuple similarity measure. In addition, domain independence facilitates the application of the DUMAS table matcher in a wide area of scenarios. String similarity measures appear to be a good choice despite their drawbacks: Much data in databases can be represented in some form of text, there is a wide variety of domain-independent string similarity measures, and string similarity measures have been successfully used in other deduplication work. Apart from duplicate detection, string similarity measures have also been studied in other research areas, e.g., information retrieval [BYRN99], text classification [Seb02], and computational biology [Gus97]. In the following, a number of string similarity measures with their advantages and disadvantages are described based on the survey by Cohen et al. [CRF03]. Afterwards, the tuple similarity measure *tupsim* is defined.

4.3.2 String Similarity Measures

In general, a similarity measure *sim* assigns a large score to objects that are similar and a low score to object that are different. In many cases, the similarity score is normalized such that it is in the interval $[0, 1]$. In contrast, a distance measure *dist* assigns low scores to similar objects and high scores to dissimilar objects. If a distance measure *dist* produces scores in the range $[0, 1]$, it can be translated into a similarity measure *sim* using the formula

$$sim(a, b) = 1 - dist(a, b) \tag{4.1}$$

where a and b are the objects to be compared. Because the distance scores are in the range $[0, 1]$, the similarity score are also in that interval. A similarity measure can be translated into a distance measure in a similar fashion.

A distance measures *dist* is a metric if it fulfills the following properties:

1. *Non-negativity*: $dist(a, b) \geq 0$,
2. *Identity of indiscernibles*: $dist(a, b) = 0$ if and only if $a = b$,
3. *Symmetry*: $dist(a, b) = dist(b, a)$,

4. *Triangle inequality*: $dist(a, c) \leq dist(a, b) + dist(b, c)$.

Searching for nearest neighbors in a metric space can be efficiently performed using metric indices [CNBYM01, HS03] — an important consideration when choosing a tuple similarity measure.

According to Cohen et al., string similarity or distance measures can be classified into the following three categories: (i) edit-distance like functions, (ii) token-based functions, and (iii) hybrid functions [CRF03].

Edit-distance Like Functions

Edit-distance like functions are derived from the classic edit distance, which is the cost of transforming one string into another using three operations: insert a character, remove a character, and substitute a character[1]. Several variants of edit distance assign different costs to edit operations. The *Levenshtein distance* uses unit cost for each operations. Thus, the edit distance becomes the minimum number of edit operations two transform a string. It can be shown that the Levenshtein distance is a metric because it fulfills metric properties defined above.

The edit distance between two strings $a = a_1 \dots \ a_m$ and $b = b_1 \dots b_n$ is computed using the following recursive formula:

$$C(i,j) = min \begin{cases} C(i-1,j)+1 \\ C(i,j-1)+1 \\ C(i-1,j-1)+c(a_i,b_j) \end{cases} \tag{4.2}$$

where $C(i, j)$ is the cost of transforming string $a_1 \dots a_i$ into string $b_1 \dots b_j$ using the three edit operations, and $c(a_i, b_j)$ is the cost of substituting character a_i with b_j. The cost of substituting a character is 0 if the character is substituted with itself or 1 if it is substituted with another character. Eq. 4.2 states that the edit distance of two strings $a = a_1 \dots \ a_m$ and $b = b_1 \dots b_n$ is the minimum of (i) the cost of transforming prefix $a_1 \dots \ a_{m-1}$ to b and removing character a_m, (ii) the cost of transforming a to prefix $b_1 \dots b_{n-1}$ and adding character b_n, and (iii) the cost of transforming $a_1 \dots \ a_{m-1}$ to $b_1 \dots b_{n-1}$ and substituting a_m with b_n.

		r	u	m	o	r	s
	0	1	2	3	4	5	6
d	1	1	2	3	4	5	6
u	2	2	1	2	3	4	5
m	3	3	2	1	2	3	4
a	4	4	3	2	2	3	4
s	5	5	4	3	3	3	**3**

Figure 4.2: Edit distance computation for “dumas” and “rumors”.

[1]Substituting a character with itself has zero cost in all edit-distance like functions.

Fig. 4.2 shows the matrix that is constructed in the process of computing the edit distance between "dumas" and "rumors". Note that the first row and the first column represent the empty source string and target string, respectively. As can be seen in the bottom right cell of the matrix, the distance between the two strings is 3. In addition, the matrix also shows the edit distances between any prefixes of the two strings. The cost of computing the edit distance is $O(mn)$. However, the space requirement is only $O(n)$: Although Eq. 4.2 presents a recursive definition, a dynamic programming algorithm can compute the edit distance column by column (from left to right) or row by row (top-down), and thus, only has to store the latest column or row, respectively.

It can be easily seen that the distance score is not in the range $[0, 1]$ but can be normalized. Given the Levenshtein edit distance $ed(a, b)$ between two strings a and b, the normalized edit distance ned can be computed as follows:

$$ned(a, b) = \frac{ed(a, b)}{\max(|a|, |b|)} \tag{4.3}$$

where $|a|$ and $|b|$ are the lengths of strings a and b, respectively. The distance is divided by the length of the longer string because, as it can be easily shown, the Levenshtein distance cannot be larger than the size of the longer string. Unfortunately, the triangle inequality does not hold for the normalized edit distance. This can be demonstrated using the strings $a =$ "ab", $b =$ "aba", and $c =$ "ba":

$$\begin{array}{lll} ed(a, b) = 1 & ed(b, c) = 1 & ed(a, c) = 2 \\ ned(a, b) = \frac{1}{3} & ned(b, c) = \frac{1}{3} & ned(a, c) = 1 \end{array} \tag{4.4}$$

Thus, $ned(a, c) \not\leq ned(a, b) + ned(b, c)$. The loss of triangle inequality does not affect the quality of similarity scores, but complicates efficient search for similar string: Metric indices and search space pruning methods that exploit triangle inequality cannot be directly applied [HS03].

While the Levenshtein measure is very simple and easy to compute, it does not always reflect the 'true' distance between strings. To illustrate, assume a list of street names. Parts of street names which appear frequently are sometimes abbreviated: 'Street', 'Avenue', and 'Court' are sometimes written as 'St', 'Ave', and 'Ct', respectively. A good distance measure should indicate that 'Street' is close to 'St'. Several variants of the Levenshtein distance that reflect the natural distance between string more closely have been studied. E.g., the Smith-Waterman distance assigns different costs to the three edit operations [SW81]. In addition, the cost for starting a gap (by inserting or removing characters) is larger than for continuing a gap. Monge and Elkan have successfully used this scheme in name matching tasks [ME96].

Token-based Functions

In contrast to edit distance, *token-based functions* require preprocessing of a string: It has to be split into a set of tokens (or terms). Afterwards, the string can be considered an unordered multiset (or bag) of tokens. Several token-based

functions exist. A simple example is the *Jaccard similarity*, which is computed as

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.5}$$

where A and B are multiset representations of strings.

A very well-known token-based measure is the cosine measure with TFIDF weighting, or *TFIDF measure* for short, which has been heavily used in information retrieval [BYRN99]. In principle, the cosine measure requires the strings to be represented as term weight vectors with unit length. The similarity of two strings is computed as the dot product of their vector representations, which is equal to the cosine of the angle between the vectors. When TFIDF weighting is applied, each term is given a weight depending on its *term frequency* (TF), i.e., the number of time it appears in a string, and its *inverse document frequency* (IDF), which is the inverse of the number of strings in which the term appears. In other words, a token is given a large weight if it appears often in a given string or if it appears in very few strings. The unnormalized weight $w'(a, t)$ of a term t in a string a is computed as

$$w'(a, t) = \log(tf_{a,t} + 1) \cdot \log(\frac{N}{df_t} + 1) \tag{4.6}$$

where $tf_{a,t}$ is the number of times term t appears in a (i.e., its term frequency), N is the overall number of strings, and df_t is the number of strings that t appears in (i.e., its document frequency). Note that the weight of a term that does not appear in a string is zero because its term frequency is zero. These weights are normalized such that the resulting weight vector has unit length. The TFIDF similarity $tfidf(a, b)$ of two strings a and b is defined as

$$tfidf(a, b) = \sum_{t \in a \cap b} w(a, t) \cdot w(b, t) \tag{4.7}$$

where $w(a, t)$ is the normalized weight of term t in string a. Only terms that appear in both strings have to be considered when comparing two strings: The weight of a term that does not occur in a string has a weight of zero, and thus, the product of the weights for that term is zero. It can be shown that, after being translated into a distance measure, TFIDF similarity does not fulfill the triangle inequality property, and thus, is not a metric.

Hybrid Functions

Similarity measures that use other similarity functions are called *hybrid functions*. The similarity measure used by the hybrid function is called secondary similarity function. Cohen et al. present two hybrid measures: the recursive matching schema and SoftTFIDF [CRF03].

The *recursive matching scheme rms* computes the similarity of two strings a and b as

$$rms(a, b) = \frac{1}{K} \sum_{i=1}^{K} \max_{j=1}^{L} sim'(a_i, b_j) \tag{4.8}$$

where a_i and b_j are the i'th and j'th token in a and b, respectively, K is the number of tokens in a, and L is the number of tokens in b. Intuitively, each token of a is assigned the token in b which is most similar according to the secondary similarity function sim'. It can be easily shown that the recursive matching scheme is not symmetric.

SoftTFIDF is a "soft" version of TFIDF that allows for errors in terms. Because only equal terms contribute to the final similarity score, small errors unduly decrease similarity (see Eq. 4.7). To compensate, SoftTFIDF also considers tokens that are similar according to the secondary measure sim'. Let $CLOSE(\theta, a, b)$ be the set of terms $a_i \in a$ such that there is at least one term $b_j \in b$ that is very similar (i.e., its similarity is above a given threshold θ):

$$CLOSE(\theta, a, b) = \{a_i \in a | \exists b_j \in b, sim'(a_i, b_j) > \theta\}. \tag{4.9}$$

The SoftTFIDF similarity $softtfidf$ of two strings a and b is defined as:

$$softtfidf(a, b) = \sum_{t \in CLOSE(\theta,a,b)} w(a, t) \cdot w(b, t') \cdot sim'(t, t') \tag{4.10}$$

where t' is the token in b that is most similar to t according to the secondary similarity function sim', and $w(a, t)$ is the normalized TFIDF weight of term t in a as defined above.

4.3.3 The Tuple Similarity Measure *tupsim*

To determine the similarity of two tuples, we consider each of them as a single string. Such a string is created by concatenating a tuple's attribute values. The tuple similarity *tupsim* should create reasonable similarity scores as discussed in Sec. 4.2. In particular, it has to consider the inherent problems described in Sec. 4.3.1. At the same time it must also be efficient. The efficiency of duplicate detection is not only determined by the cost of calculating the actual similarity of two tuples, but also by the number of tuples which have to be compared. In the following, we will discuss the similarity measures described above and provide arguments, why the TFIDF measure is the best choice for *tupsim*.

Edit-distance like functions are order-dependent: Changing the order of a tuple's attributes greatly affects the edit distance to another tuple if a tuple is considered a single string. Thus, edit distance is not a good choice with respect to Problem 1. A solution to this problem would be to add another operation for moving whole blocks of text. However, such variants are known to be computationally expensive, and thus, are not further regarded as a possible tuple similarity measure [LT97].

Of course, edit distance could be used as a secondary measure in one of the hybrid functions. Instead of considering a tuple as a single string, one could compare the attribute values and apply the recursive matching scheme to create a combined score. Eq. 4.8 could be directly applied: a_i and b_j would represent attribute values of tuples a and b, respectively. The recursive matching scheme was not used as tuple similarity measure for reasons of efficiency.

In contrast to the above mentioned similarity measures, the TFIDF measure has features that make it a reasonable choice for a tuple similarity measure:

1. *Order independence*: The TFIDF measure is a token-based measure, and thus, order-independent because each string is represented as a bag of tokens. By considering each tuple as a single string and applying an order-independent string similarity measure we tackle Problem 1.

2. *TFIDF weighting*: The inverse document frequency, which is part of the TFIDF weighting scheme, gives a large weight to terms which appear infrequently. Thus, if a certain value appears in many tokens, it gets a relatively low weight, and thus, has a smaller effect on the similarity of two tuples. Hence, the case described in Problem 4a is tackled by applying TFIDF weighting.

3. *Efficient top-k search*: Efficient algorithms for detecting the k most similar strings exist and can be applied for duplicate detection.

Problems 4b and 2 cannot be directly solved. However, we experimentally show that the TFIDF measure still performs very well even in difficult scenarios.

Because of its useful properties, the TFIDF measure is used to determine the similarity of two tuples in the duplicate detection step. As discussed above, all tuples are translated into strings by concatenating their attribute values. When it is clear from the context, the string is given the same name as the tuple which it represents. The tuple similarity *tupsim* of two tuples r and s is defined as:

$$tupsim(r,s) = tfidf(r,s) = \sum_{t \in r \cap s} w(r,t) \cdot w(s,t) \tag{4.11}$$

where w is the normalized weight as used in Eq. 4.7.

We point out that the tuple similarity measure can benefit from any normalization procedure: As shown in Eq. 4.11, *tupsim* only considers equal terms, and a slight variation might have a large impact on the tuple similarity. The normalization of strings potentially improves duplicate detection accuracy: E.g., the value "86" in an attribute $Year$ could be changed to "1986", if all year entries in the other database are four-digit numbers. However, that kind of preprocessing requires a good understanding of the attributes' semantics by the user. Although there are application-independent normalization techniques (e.g., stemming [BYRN99]), most normalization measures require reasonable knowledge of the schemata, which we do not expect. We also stress that only a few duplicates are required. Thus, if only some of the duplicate tuples have value discrepancies as described above, our duplicate detection procedure still produces a good result.

SoftTFIDF is also applicable for duplicate detection in unaligned relations because it has the same features as TFIDF. However, it is more expensive to compute, and searching for similar tuples is more complicated. Because very good results can already be achieved with the simpler TFIDF measure (see Sec 4.6), SoftTFIDF is not further considered.

4.4 Searching For Duplicates

Duplicate detection is inherently a problem with quadratic complexity: Each tuple in the first table has to be compared with each tuple in the second table. Such *exhaustive search* is clearly infeasible for larger data sets. Reducing the number of tuple comparisons is very important to make duplicate detection scalable. Existing solutions to this problem cannot be applied for the same reason why the duplicate detection algorithms discussed in Sec. 3.4 cannot be used: The semantics of the attributes and correspondences are unknown. E.g., an expert needs to define sort keys when the sorted neighborhood method is applied, which requires extensive knowledge of the application domain and the semantics of the attributes. Consequently, an algorithm that is only based on the tuple similarity measure has to be devised. The goal of such an algorithm is to detect the top-k duplicates, i.e., the k most similar tuple pairs with a minimum number of few tuple comparisons.

A first improvement over exhaustive search would be to only consider tuples which have at least one term in common, because only tuples that share at least a single term have non-zero similarity (Eq. 4.11). A *semi-naïve algorithm* would pick each tuple from the source table and look for tuples in the target database that contain at least one of its tokens. This lookup can be efficiently performed using an inverted index on the target table, which maps terms to tuple identifiers [BYRN99]. The top-k duplicates can be easily extracted from the resulting set of tuple pairs with non-zero similarity.

Compared to exhaustive search, the the semi-naïve algorithm achieves a major reduction in the number of tuple comparisons in most scenarios. However, it ignores the TFIDF weighting of tokens, and thus, misses the chance for a larger performance gain. The algorithm computes the similarity even of tuples that share only low-weight terms although those terms have only a small effect on the similarity score. Hence, the semi-naïve algorithm is considered suboptimal because tuples which have only low-weight terms in common are less likely to be contained in the set of top-k tuple pairs. A more intelligent algorithm would search for the top-k duplicates by finding tuple pairs that have high-weight tokens in common, and stop searching when no tuple pairs with a high similarity score can be expected.

This idea is realized in the implementation of the duplicate detection algorithm, which is an adaptation of the *Whirl algorithm* for similarity joins in relational databases [Coh98]. Whirl performs A* search in the space of possible tuple pairs. A* is a widely known best-first search algorithm that finds a path from a given start state to a goal state with the smallest cost. In each iteration, the algorithm picks the state n from a list of open states with the smallest assigned cost. If the state is a goal state, then it is presented as the result. If it is an intermediate state, the graph is traversed further, and new states are added to the list of open states. The cost $f(n)$ of a state n is calculated as

$$f(n) = g(n) + h(n) \tag{4.12}$$

where $g(n)$ is the actual cost of the path from the source to state n and $h(n)$ is

the estimated cost of the path from n to the closest goal state. A* is optimal if $h(n)$ is an admissible heuristic, i.e., it never overestimates the cost to reach a goal. In most cases $h(n)$ is defined to be zero if n is a goal state.

In the duplicate detection implementation each state is a four tuple $\langle r, s, b, e \rangle$, where r represents a source tuple, s represents a target tuple, b is the current bound, and e is the exclusion list. Both r and s can be either unbound (denoted as $\perp$) or bound to a tuple. Based on the values of r and s, three state types are distinguished: (i) A state is a *start state* when both r and s are unbound, (ii) a state is a *intermediate state* when only r is bound, and (iii) a state is a *goal state* if both r and s are bound. The exclusion list e is a list of tokens which may not be contained in target tuples — the intention of this list is made clear in the description of the algorithm below. The bound b is the maximum similarity of two tuples that can be reached from the given state. Note that the goal is to maximize similarity as opposed to minimize cost. Thus, b must be an overestimate instead of an underestimate. The *bound function* $B(r, s)$ is defined as:

$$B(r,s) = \begin{cases} \infty & \text{if } r = \perp \wedge s = \perp \\ B(r) & \text{if } r \neq \perp \wedge s = \perp \\ tupsim(r,s) & \text{if } r \neq \perp \wedge s \neq \perp. \end{cases} \tag{4.13}$$

The function $B(r)$ determines the bound for intermediate states. It is computed as

$$B(r) = \sum_{t \notin e} w(r,t) \cdot maxweight(t) \tag{4.14}$$

where t is a term that does not appear in the exclusion list e, and $maxweight(t)$ is the maximum weight of term t in the target relation. The maximum weight of a term is stored as additional information in the inverted index, and thus, can be efficiently retrieved.

Before describing the actual search procedure, it has to be noted that not the entire search graph needs to be kept in main memory, but only a list of open states, which is called $OPEN$. As mentioned in the description of A*, the state with the smallest cost (largest bound) needs to be extracted in each iteration. Hence, the $OPEN$ list is implemented as a priority queue, which allows insertion of a single state and removal of the state with the largest bound in $O(\log n)$ [CLR01].

The duplicate detection algorithm is depicted in Alg. 1. Variables are initialized at the beginning: The result set *result* is set to the empty set (line 1), while the list of open states $OPEN$ contains the start state s_0 (line 2). The following loop is executed until (i) the list of open states is empty or (ii) k goal states have been found. At the beginning of the loop, the current state s becomes the state with the largest bound (line 4), which is also removed from the $OPEN$ list (line 5). If the extracted state is a goal state, then it is added to the result set *result* (line 7). Otherwise, child states are created for state s and added to the list of open states (line 9). The creation of child states is described below. The result set is returned after the loop has terminated (line 12).

Algorithm 1: A* search for top-k duplicates (adapted from [Coh98])

Output: Set of states representing the k most similar tuple pairs

```
1  result := {};
2  OPEN := {s_0};
3  while OPEN ≠ ∅ ∧ |result| < k do
4      s := argmax_{s'∈OPEN} B(s);
5      OPEN := OPEN − {s'};
6      if goalState(s) then
7          result := result ∪ {s};
8      else
9          OPEN := OPEN ∪ children(s);
10     end
11 end
12 return result
```

The duplicate detection procedure uses two operations two create child states: *explode* and *constrain*. The explode operation is performed only on the start state, while constrain is performed on intermediate states.

Explode: At the beginning of the duplicate detection procedure, the $OPEN$ list contains only the start state, where $r = \perp$, $s = \perp$, $e = \{\}$ (i.e., e is empty), and $B(r, s) = \infty$ as defined above. In the first phase of the algorithm, this state is extracted from the $OPEN$ list and "exploded": For each source tuple $r_1, \ldots, r_m$, a new state is created where r is bound to the source tuple, s is unbound, e is empty, and b is computed as defined in Eq. 4.14. Those intermediate states are inserted into the $OPEN$ list.

After the explode phase the iterative part of A* starts. In each iteration, the state with the largest bound is extracted from the $OPEN$ list. If it is a goal state, it is added to the result set. The algorithm terminates if it is the k'th element in the result, because only k duplicates need to be found. The algorithm continues if more tuple pairs are required. If the extracted state is an intermediate state, then it must be constrained.

Constrain: An intermediate state that has been extracted from the $OPEN$ list is constrained by "creating" its child states and adding them to the open list. The child states have the same source tuple r, but either a bound target tuple s or an extended exclusion list e. They are created as follows: A term t that appears in the source tuple r, but not in the exclusion list of the state, is picked, and target tuples containing t are extracted using the inverted index on the target relation. From the extracted tuples only the l tuples which do not contain any term of the exclusion list are used to create $l + 1$ new states: l goal states in which the target tuple is bound, and an intermediate state in which the target tuple remains unbound, but the term t is added to the exclusion list.

Because a new term has been added to the exclusion list, the bound of the new intermediate state is lower than the bound of its parent. In order to reduce

the number of tuple comparisons, a term t should be chosen such that the bound of intermediate states quickly decreases. Thus, we pick a term t that maximizes $w(r,t) \cdot maxweight(t)$, because its insertion into the exclusion list has the largest effect on the bound of the derived intermediate state (see Eq. 4.14).

Beside computation of the bound of intermediate states, the intention of the exclusion list is also to avoid creating the same tuple combination twice: If a term t_i appears in an intermediate state for source tuple r, it implies that for all target tuples s containing t_i, there already is or has been a goal state in the $OPEN$ list with r as source tuple and s as target tuple. If such an intermediate state is constrained using term t_j, then no goal states for target tuples containing t_i are created.

4.5 The Effect of Sampling on the Number of Duplicates

The search for duplicates described in Sec. 4.4 is performed in main memory. Depending on the size of the data sets and the available RAM, one might run out of memory if all existing data is searched for duplicates. To make duplicate detection with the search algorithm feasible, the size of the tables need to be reduced: When the tables under consideration contain less tuples, the search space, and thus, the $OPEN$ list is smaller, and memory overflow can be avoided.

To reduce the table sizes, *random Bernoulli sampling* is applied in the implementation of the duplicate detection step. In this scheme, each tuple of a table R has an independent chance of p_R to enter the sample. We call p_R the *sampling rate* of relation R. If a table R has n tuples, and a sampling rate of p_R is used, the expected size s of the sample is $E[s] = p_R \cdot n$.

The goal is to create a sample of size s such that duplicate detection is possible. At the same time, as many tuples should be contained in the table to ensure that enough duplicates can be found. Thus, if a table has n tuples, and the size of the sample should be roughly s tuples, then the sampling rate is set to $p_R = \frac{s}{n}$. The actual size of the sample is unlikely to be exactly s, because the chance of a tuple to enter the sample is independent of other tuples. Note that Bernoulli sampling has been implemented in major DBMS, e.g., DB2 [GGLZ04], and thus, can be efficiently computed.

Unfortunately, random sampling has a negative effect on the number of duplicates. Fig. 4.3 shows an example with two tables R and S containing four duplicates (r_1, s_1), (r_2, s_4), (r_3, s_3), and (r_4, s_2) depicted by black bars. The arrows indicate which tuples represent the same real-world entity. Both tables are sampled with a rate of 50%, i.e., $p_R = p_S = 0.5$. The area above the dashed lines represents the part of the relations that is in the sample. Note that in each sample two tuples, which were considered duplicates in the original data set, can be found: r_1, r_2, s_1, and s_2. However, when considering the sampled tuples, only a single duplicate remains: (r_1, s_1). The number of duplicates decreases more than the sampling rate indicates because the matching tuples of r_2 and

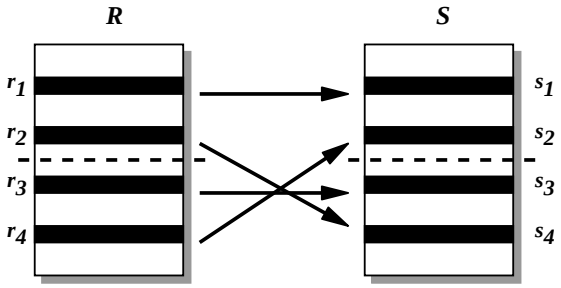


Figure 4.3: Effect of sampling on the number of duplicates

s_2, namely s_4 and r_4, respectively, are lost in the sampling process.

Given the number of duplicates d, the sampling rates p_R and p_S for relations R and S, respectively, the number of duplicates in the sample d_{sample} is expected to be:

$$E[d_{sample}] = d \cdot p_R \cdot p_S. \tag{4.15}$$

Each of the d tuples in R, which has a matching tuple in S, has a chance of p_R to be in the sample. For each of those tuples that made it into the sample, its matching tuple has a chance of p_S to be in the sample of S. Thus, the expected value of d_{sample} is the product of the number of duplicates and the sampling rates.

To summarize, sampling facilitates the detection of a few duplicates even when the underlying data sets are large. However, sampling also reduces the number of duplicates relative to the size of the relations. The experiments described in Sec. 4.6.2 indicate that duplicate detection is feasible even when only very few duplicates exist. Those experiments also show that the precision of duplicate detection decreases at a certain recall level. This precision decrease also affects the number of duplicates that can be used for table matching: If precision decreases at recall level r, the number of tuple pairs used in the matching step should be at most $r \cdot E[d_{sample}]$ to ensure that no false duplicates are used.

4.6 Experimental Evaluation

To evaluate the performance of our duplicate detection procedure, both in terms of effectiveness and efficiency, a number of experiments were performed. To demonstrate the applicability in a real-world scenario, data extracted from real estate advertisements were used (Sec. 4.6.1). Because the results were always perfect, additional experiments on generated data were performed to see how the duplicate detection procedure performs in critical configurations (Sec. 4.6.2). In particular, the effect of Problems 4b and 2 described in Sec. 4.3.1 needs to be studied.

The quality of duplicate detection in aligned relations is measured in terms

of precision and recall, which are calculated as

$$Precision = \frac{|D \cap R|}{|R|} \qquad Recall = \frac{|D \cap R|}{|D|} \tag{4.16}$$

where D is the set of true duplicates, and R is the set of retrieved duplicates. Intuitively, precision is large if there are only few false duplicates (false positives) in the result set, while a large recall is achieved when only few duplicates are missed (false negatives).

Note that there is usually a tradeoff between the two measures: Being very strict and allowing no or few errors benefits precision, but many duplicates might be missed. On the other hand, allowing more errors increases the chance that dissimilar duplicates are found, but also increases the possibility of false duplicates entering the result set. Classic duplicate detection algorithms that work on aligned tables aim at achieving both high recall and high precision. As described above, maximizing both measures is next to impossible, and hence, balancing a duplicate detection process is critical. In the case of duplicate detection in unaligned relations, we are only interested in precision, because the goal is only to detect a few duplicates that are most likely true duplicates.

4.6.1 Real-world Data: Real Estate Advertisements

For the experiments in a real-world scenario data sets containing real estate advertisements of two Berlin newspapers ("Morgenpost" and "Tagesspiegel") were applied. The data was extracted from their web sites in two consecutive weeks, so altogether four data sets were used. The number of tuples ranged between 1509 and 3772. For each possible combination of data sets the top-10 duplicates were detected.

The precision of duplicate detection was always 100% (thus, no graphs are provided). The reasons for this perfect result are hidden in the properties of the used data sets:

1. *Large extensional overlap*: Some people or companies tend to place their advertisements in both newspapers and in consecutive weeks to increase their success chance. Thus, there are several duplicates both when comparing data from different newspapers and different weeks.

2. *Large intensional overlap*: Most attributes attributes are present in both schemata. In particular, the more relevant attributes (e.g., name of the advertising person or company, address of the advertised apartment, contact, etc.) can always be found in advertisements.

3. *Similar representations*: The information that is most relevant for duplicate detection, e.g., name of the advertiser, address, or phone number, is represented in the same fashion across advertisements. Even some of the abbreviations and acronyms used to describe a flat are similar across newspapers. The latter has only a small effect: If the same abbreviation is used in many advertisements, its TFIDF weight will be low.

4.6.2 Experiments on Generated Data

The experiments on real-world data did not demonstrate the effectiveness of the tuple similarity measure in critical configurations. To do so, experiments have to be performed to answer the following three questions, which correspond to the three properties of the real estate advertisements described in Sec. 4.6.1:

1. *Decreased extensional overlap*: How effective is the tuple similarity measure when there are only few duplicates?
2. *Decreased intensional overlap*: What is the precision of duplicate detection when only few attributes correspond?
3. *Erroneous information*: How much is duplicate detection affected by errors in the data (e.g., misspellings)?

All data sets for the following experiments were created using the dirty data generator G2[2] used in [BSS03], which improves the database generator used in [HS98] to create more realistic data. The tool allowed us to inject fuzzy duplicates, where fuzziness is controlled for each attribute through error probability parameters for different types of errors, such as "replacement error" and "deletion error". Thus, the effect of errors is always reflected in the presented results (Question 3).

For each experiment we generated two databases *DB1* and *DB2*, each with 5,000 tuples. Figure 4.4 shows the baseline setup for the following experiments (the schemata for $DB1$ and $DB2$) and the correct matching. Unbeknownst to the duplicate detection algorithm, the two databases have six attributes in common, and each has two or more additional attributes. Attribute values were randomly chosen from long, predefined lists of values. Note that the attribute pairs *Birth-place* and *City* as well as *Birth-district* and *District* draw values from the same domains, making duplicate detection challenging. Finally, the columns were randomly shuffled. All reported results are averages of five independent runs with newly created databases for each experiment.

Reduced Extensional Overlap

To examine the effect of the number of duplicates on the precision of the duplicate detection result (Question 1), data sets for the baseline setup (Fig. 4.4) containing 10, 50, and 100 duplicates were generated. The experimental results depicted in Fig. 4.5 show the precision at certain recall levels, averaged over the five different data sets used for each setup. In this experiment, the number of duplicates detected by the algorithm was not fixed — instead, a list of tuple pairs ranked by decreasing tuple similarity was produced. Hence, a recall level of 20% is the point in the list were 20% of all *true* duplicates have been detected.

The result shows that the precision at early recall levels is always large: Depending on the extensional overlap, precision drops below 100% between

[2]Kindly provided to us by Luca De Santis of the DaQuinCIS team at Universitá di Roma "La Sapienza".

Attr. of *DB1*		Attr. of *DB2*
SSN	⟶	–
Profession	⟶	–
Surname	⟶	*Surname*
Name	⟶	*Name*
Birth-date	⟶	*Birth-date*
Birth-place	⟶	*Birth-place*
Birth-district	⟶	*Birth-district*
Sex	⟶	*Sex*
–	⟶	*City*
–	⟶	*District*

Figure 4.4: The correct matching from *DB1* to *DB2*

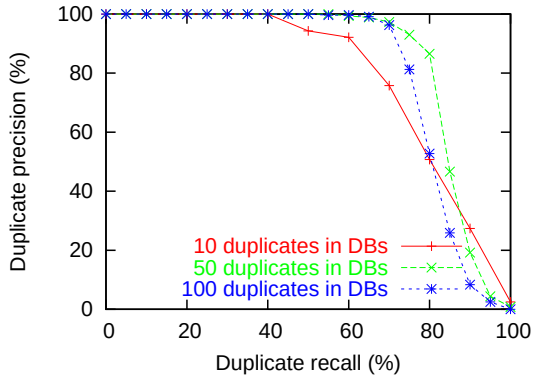


Figure 4.5: Influence of number of duplicates

40% and 60% recall. Because our goal is only to detect a few (i.e., the most similar) tuple pairs, this is a very encouraging result. Only the top-ranked tuple pairs are relevant for the DUMAS table matcher, and the experiment indicates that the tuple similarity measure produces the largest similarity scores only for true duplicates despite a small extensional overlap.

Reduced Intensional Overlap

The next experiment is designed to assess the influence of intensional overlap, i.e., the number of common attributes in both databases. The schema of *DB1* remains the same (see Fig. 4.4). From the schema of *DB2* the overlapping attributes *Sex*, *Birth-district*, and *Birth-place* are successively removed, replacing them by new attributes *Street number*, *Address*, and *Postal code*. Note that by each of those changes, a single correspondence is removed. We use a fixed number of 50 duplicates. The result of these experiments is depicted in Fig. 4.6.

As expected, intensional overlap does have an effect on the precision. When less attributes correspond, the effect of similar values in those attributes on the

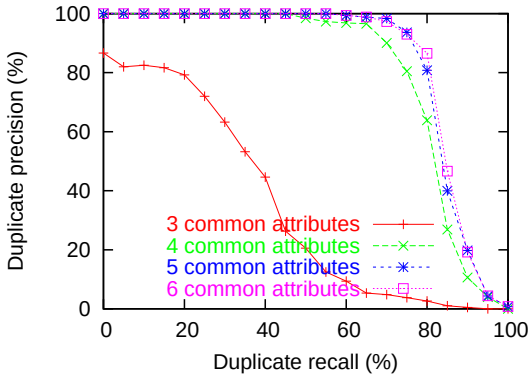


Figure 4.6: Influence of degree of intensional overlap

tuple similarity decreases. Fortunately, only a minor decrease in precision can be observed in most cases. The decrease can only become severe if unrelated attributes have the same value domain (e.g. *Birth-Place* and *City* in Fig. 4.4). In such a case, Problem 4b described in Sec. 4.3.1 arises.

Attr. of *DB1*		Attr. of *DB2*
SSN	⟶	–
Profession	⟶	–
Sex	⟶	–
Birth-district	⟶	–
Surname	⟶	*Surname*
Name	⟶	*Name*
Birth-date	⟶	*Birth-date*
Birth-place	⟶	*Birth-place*
–	⟶	*City*
–	⟶	***District***
–	⟶	*Street no*
–	⟶	*Address*

(a) Four corresponding attributes

Attr. of *DB1*		Attr. of *DB2*
SSN	⟶	–
Profession	⟶	–
Sex	⟶	–
Birth-district	⟶	–
Birth-place	⟶	–
Surname	⟶	*Surname*
Name	⟶	*Name*
Birth-date	⟶	*Birth-date*
–	⟶	***City***
–	⟶	***District***
–	⟶	*Street no*
–	⟶	*Address*
–	⟶	*Postal code*

(b) Three corresponding attributes

Figure 4.7: Reduced intensional overlap: four and three matches.

Note that this problem is already present in the baseline setup: *Birth-place* and *Birth-district* in $DB1$ have the same domain as *City* and *District* in $DB2$, respectively (Fig. 4.4). However, the two attributes in $DB1$ have a matching partner in $DB2$. Thus, the values of *City* and *District* do not mislead duplicate detection. In the step from five to four corresponding attributes, *Birth-district* is removed from $DB2$, resulting in the configuration depicted in Fig 4.7(a). Despite the fact that there is no matching partner for attribute *Birth-district* anymore, no significant decrease could be detected. The reason for this lies in

the TFIDF weighting scheme: There are only few possible districts, resulting in a small inverse document frequency, and thus, the district has only a small effect on the tuple similarity.

The problem becomes severe in the configuration with three correspondences (Fig. 4.7(b)). In this experiment, attribute *Birth-place* is also removed. However, attribute *City* is still in $DB2$. In that configuration, precision of duplicate detection is heavily decreased because (i) these attributes draw values from the same domain, (ii) the impact of a given city on the tuple similarity is significant, and (iii) the three remaining correspondences cannot sufficiently compensate.

4.7 Discussion

In this chapter we discuss the problem of duplicate detection in unaligned relations, which is critical for the feasibility of duplicate-based schema matching, but has not been considered before. We address several challenges that do not occur in duplicate detection when correspondences are known. To find the most similar tuple pairs, we define a tuple similarity measure *tupsim* and describe an algorithm to efficiently find duplicates.

It has been shown that most of the inherent problems defined in Sec. 4.3.1 could be resolved:

- The problem of unknown schema alignment was tackled by ignoring the record structure and considering each string as a single tuple (Problem 1).
- By using a domain-independent string similarity measure, the duplicate detection process is not affected by the lack of information about the semantics of the attributes (Problem 3).
- The TFIDF weighting scheme, which is used in the tuple similarity measure, helps in avoiding the problem of misleading attribute similarities in corresponding attributes (Problem 4a).

The problem of misleading attribute similarities that appear in non-corresponding attributes (Problem 4b) is more difficult to solve: If attribute values match by chance, we still expect the duplicate detection to produce good results. If this case appears frequently, e.g., when non-corresponding attributes have the same value domain, then duplicate detection precision can drop if only few attribute correspondences exist. We consider that problem and the problem of a small intentional overlap (Problem 2) in our evaluation of the algorithm,

The experiments show that it is possible to detect a few duplicates in unaligned relations with very good precision. The properties of the real estate advertisements described in Sec. 4.6.1 are favorable to our duplicate detection procedure. However, this is not an unusual scenario, and similar properties can be found in other application domains, too. In contrast, the generated data sets allowed us to gauge how effective the *tupsim* measure is in critical scenarios. It could be seen that the top-ranked duplicates can be trusted even when only few duplicates exist and, to some extent, when only few attributes correspond.

The experiments also show that false duplicates can be produced if Problem 4b occurs in conjunction with a small intensional overlap (Problem 2). In the following chapter, we evaluate how such false duplicates affect the resulting schema matching, and will examine a way to increase precision of duplicate detection using known correspondences.

Chapter 5

The Matching Step: Extracting Correspondences From Duplicates

The matching step of the DUMAS table matcher uses the duplicates extracted from the tables to establish a matching. Each of the duplicates indicates a certain matching, which might disagree with other duplicates. To create a single matching, the duplicates' indications have to be merged. Afterwards, the resulting correspondences need to be checked, and if some of them are uncertain, the table matching process needs to reiterate back to produce more duplicates (see Fig. 5.1).

5.1 Establishing Correspondences By Aggregating Duplicate Votes

Each of the duplicates detected in the previous step indicate to what extent attributes of the tables are related. These indications can be extracted by pairwise comparison of attribute values. E.g., duplicate tuples r_9 and s_2 in Fig. 3.1 both have some attribute values in common: The string "Dean" can be found in $LastName$ and LN, while "(369) 3663624" is the value of $Phone$ and Tel. Hence, this duplicate indicates the matching $\{(\{LastName\}, \{LN\}), (\{Phone\}, \{Tel\})\}$. This idea is stated in the *matching heuristic*: If two attribute values of a pair of tuples representing the same real-world entity are equal or highly similar, then those attributes are likely to correspond. In contrast, attributes whose values differ are probably semantically different. In other words, each duplicate gives a "vote" that shows how closely attributes of the tables are related.

The indications inherent in the duplicates are not always correct or unambiguous. The duplicate tuple pair (r_3, s_3) in Fig. 3.1, has the same value for attributes Tel, $Phone$ and Fax. Thus, based on on this duplicate, it cannot

be unambiguously deduced which of the two attributes *Phone* and *Fax* corresponds to *Tel*. Duplicate (r_4, s_4), which is also depicted in Fig. 3.3, has the same value "Sam"[1] for attributes *FirstName* and *Acc*, indicating a false match. The effect of such false indications can be reduced by aggregating the votes of several duplicates instead of just using a single one.

Thus, the matching step can be separated into two substeps:

1. *Pairwise Attribute Value Comparison*: The attribute values of each duplicate need to be compared. The result of attribute value comparison is a similarity matrix for each duplicate (Sec. 5.2).

2. *Aggregation and Reasoning*: The similarity matrices need to be aggregated, and a schema matching needs to be extracted (Sec. 5.3).

After establishing a matching, the algorithm has to decide if the correspondences can be trusted (Sec. 5.4). If some of the correspondences are uncertain, the process should resume with detecting more duplicates, as depicted in Fig. 5.1. In contrast to the problem of duplicate detection in unaligned schemata, the correspondences which are considered certain by the algorithm can be used. To do so, an extended tuple similarity measure has to be defined that takes certain correspondences into consideration (Sec. 5.5).

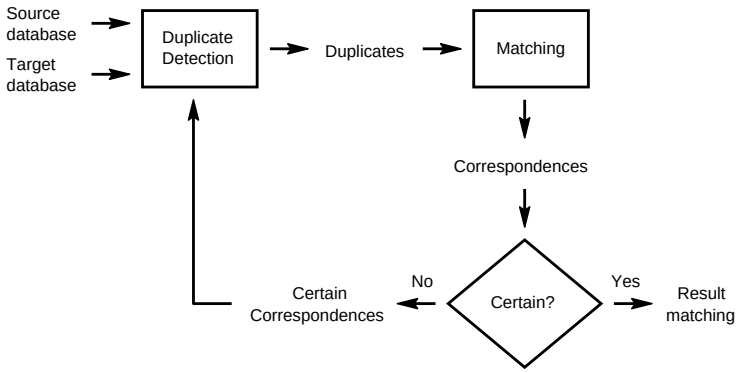


Figure 5.1: The duplicate-based schema matching process

5.2 Comparing Attribute Values of Duplicates

The first phase of the matching step is the pairwise comparison of attribute values. Given a duplicate (r, s) one has to determine the similarity of each attribute value in r with each attribute value in s. To compute a similarity score, the field similarity measure $fieldsim$, which is defined in Sec. 5.2.1, will

[1] Recall that the similarity measure is case-insensitive because the tokenizer normalizes strings.

be used. Based on the similarity scores, a similarity matrix for each duplicate is constructed as described in Sec. 5.2.2.

5.2.1 The field similarity measure *fieldsim*

Similar to the detection of duplicates, a domain-independent string similarity measure will be used to determine the similarity of attribute values. However, there are two differences that have to be considered when designing the field similarity measure:

1. *Lengths of values*: In contrast to tuples, which are comprised of several attribute values, the strings the field similarity measure has to deal with are usually short.

2. *Accuracy*: The field similarity measure should closely reflect the "true" similarity of attribute values. While in duplicate detection it is tolerable to miss duplicates due to too low similarity scores, a low score for similar values might directly result in a missed correspondence. Analogously, a similarity score that is too high can easily produce a false match.

The discussion of possible field similarity measures is based on the overview of string similarity measures in Sec. 4.3.2. In general, the measure should consider possible errors (e.g., misspellings) when determining the similarity of two attribute values. Using the TFIDF measure for comparing attribute values is unlikely to produce good result, because it only considers equal terms (Eq. 4.7). While it has been a good choice as a tuple similarity measure, due to the limited size of attribute values even small errors might cause a dramatic decrease in the similarity score. As described above, this can lead to missed correspondences. For similar reasons, other token-based measures are rejected.

Edit-distance like functions are very useful for short strings, and thus, a better choice than token-based measures. However, they are also order-dependent. This might not be disadvantageous for many attributes, but in some cases different representations with varying orders of terms are possible: E.g., if there is a single attribute for the name of a person, Sam Adams can be represented as "Sam Adams" or "Adams, Sam". The field similarity measure should be able to recognize both attribute values to be similar.

For the reasons described above, the SoftTFIDF measure (Eq. 4.10) is used as the field similarity measure $fieldsim$. As discussed in Sec. 4.3.2, SoftTFIDF is order independent and also considers highly similar terms in addition to equal terms. To determine the similarity of terms, the normalized Levenshtein edit distance *ned* (Eq. 4.3) is transformed into a similarity function. The *term similarity termsim* of two terms t and t' is computed as follows:

$$termsim(t, t') = 1 - ned(t, t') = 1 - \frac{ed(t, t')}{\max(|t|, |t'|)}. \tag{5.1}$$

The field similarity $fieldsim$ of two attribute values a and b is defined as:

$$fieldsim(a,b) = \sum_{t \in CLOSE(\theta,a,b)} w(a,t) \cdot w(b,t') \cdot termsim(t,t') \tag{5.2}$$

where t' is the term in b that is most similar to t according to the term similarity measure $termsim$, and $CLOSE(\theta, a, b)$ is a subset of all terms in a with

$$CLOSE(\theta,a,b) = \{t \in a | \exists t' \in b, termsim(t,t') > \theta\}. \tag{5.3}$$

One reason why SoftTFIDF was preferred over the recursive matching scheme (Eq. 4.8), which is also order-independent and considers similar terms, is the TFIDF weighting scheme: E.g., when comparing attribute values "Microsoft Inc." and "Micosoft" one notices that, beside a spelling error, one of two terms is missing in the second value. Thus, one could deduce that both values are very different. However, the term "Inc." is a standard abbreviation used in many company names — removing it leaves the company name still comprehensible. Because the term is part of very many company names, its TFIDF weight is low, and thus, its removal does not have a large effect on the SoftTFIDF similarity.

The SoftTFIDF similarity is more expensive to compute than TFIDF. However, as opposed to the duplicate detection step, this is not a big issue because only the attribute values of a few tuple pairs have to be compared.

5.2.2 Creating the Similarity Matrix

	John	Dean	m	(369) 3663624	(367) 3663625
Dean	0	1.0	0	0	0
jd	0	0	0	0	0
(369) 3663624	0	0	0	1	0.87
XP	0	0	0	0	0

Table 5.1: Similarity matrix for a duplicate pair

Given a single duplicate tuple pair, the field similarity measure is used to compute the similarity of their attribute values, i.e., each attribute value of one tuple is compared with each attribute value of the other tuple. The similarity scores for a single tuple pair are represented as a *similarity matrix*. Fig 5.1 shows the similarity matrix for duplicate (r_9, s_2) of Fig. 3.1, where the threshold θ of Eq. 5.3 was set to 0.5.

5.3 Aggregating and Reasoning

Because a single duplicate might coincidentally indicate a false correspondence or miss a correspondence, several tuple pairs are used. In the following, we describe how the similarity matrices are aggregated and how correspondences are extracted.

5.3.1 Creating The Average Similarity Matrix By Aggregation

For each detected duplicate a similarity matrix is created. Those matrices are merged into the *average similarity matrix* M by computing their average, i.e.,

$$M = \frac{1}{k} \sum_{i=1}^{k} M_i \tag{5.4}$$

where k is the number of duplicates and M_i is the i'th similarity matrix. The average similarity matrix describes the similarity of the tables' attributes based on the values of the duplicates. Tab. 5.2 depicts the average similarity matrix for the running example, which has been created using the three duplicate tuple pairs. By $M(a, b)$ we denote the average similarity score of attributes a and b in the matrix M.

	FirstName	*LastName*	*Sex*	*Phone*	*Fax*
LN	0	1.0	0	0	0
Acc	0.33	0	0	0	0
Tel	0	0	0	1	0.73
OS	0	0	0	0	0

Table 5.2: Average similarity matrix

5.3.2 Reasoning: How To Extract Attribute Correspondences

Based on the average similarity matrix a simple matching has to be established. The goal is to produce a set of correspondences such that each attribute corresponds to at most one other attribute. As a first step, a *graph matching* is extracted from the matrix. In general, a matching on a graph is a subgraph with the same nodes and a subset of the edges such that each node is incident with at most one edge. The graph on which the matching is performed is constructed from the average similarity matrix as follows: Each attribute is represented as a node, and for each element $M(a, b)$ in the average similarity matrix there is

an edge between the node representing attribute a and the node representing attribute b with the weight $M(a, b)$. It can be easily shown that the resulting graph is bipartite.

Several criteria for choosing a graph matching are conceivable. We evaluated both stable marriage [GI89] and maximum weight matching [Gal86] using synthetic data. The *stable marriage problem* is formulated as follows: Given a set of n men and n women, marry them off in pairs after each man has ranked the women in order of preference from 1 to n, $\{w_1, ..., w_n\}$ and each women has done likewise, $\{m_1, ..., m_n\}$. If the resulting set of marriages contains no pairs of the form $\{m_i, w_j\}$, $\{m_k, w_l\}$ such that m_i prefers w_l to w_j and w_l prefers m_i to m_k, the marriage is said to be stable. When applied to our problem, attributes of the source table and the target table represent men and women, respectively, and the average similarity scores are used to rank partners. In contrast, a *maximum weight matching* is a matching that maximizes the sum of the weights of the edges.

The experiments could not prove any strategy to be superior, but we believe that the similarity scores closely reflect the similarity of attributes, and do not only represent a means of ranking possible partners. Hence, in the DUMAS table matcher, a maximum weight matching is computed. The Hungarian Method is applied for that purpose [PS82].

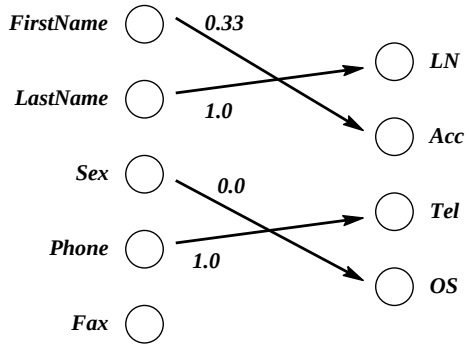


Figure 5.2: A graph matching.

We denote by $\mathcal{GM}$ the set of attribute pairs that is indicated by the edges of the produced graph matching. For the matching in Fig. 5.2, which depicts the maximum weight matching of the bipartite graph that is constructed from the matrix in Tab. 5.2, this set of attribute pairs is

$$\mathcal{GM} = \{(FirstName, Acc), (LastName, LN), (Sex, OS), (Phone, Tel)\}.$$

One can see that the correct correspondences are included, but also a few false matches. To avoid the latter, edges with a weight below a given threshold θ_{prune} are removed in a final *pruning* step to produce the pruned graph matching $\mathcal{GM}'$. Given a pruning threshold $\theta_{prune} = 0.5$, the pruned graph matching is

$$\mathcal{GM}' = \{(LastName, LN), (Phone, Tel)\}.$$

The attribute pairs in $\mathcal{GM}'$ are translated into pairs of attribute groups, which constitute the resulting simple matching $\mathcal{M}$:

$$\mathcal{M} = \{(\{LastName\}, \{LN\}), (\{Phone\}, \{Tel\})\}.$$

5.4 Certainty of Attribute Correspondences

Given the average similarity matrix of Tab. 5.2, it can be seen that the correspondence $(\{LastName\}, \{LN\})$ is very certain: There is no attribute that is nearly as similar to $LastName$ as LN, and vice versa. Attribute Tel is a different case: In many cases, the value of Fax is very similar or even equal to the value of Fax, resulting in a large similarity score for both attributes. We call the match $(\{Phone\}, \{Tel\})$ *uncertain* because there is an alternative matching, in which Fax matches with Tel, which is almost as good as $(\{Phone\}, \{Tel\})$. Because the correspondence is uncertain, we have to search for additional duplicates to strengthen the correspondence of Tel either with $Phone$ or with Fax.

Correspondences are checked for certainty after a schema matching has been established. If all correspondences are certain, the schema matching is presented to the user. Otherwise, the certain correspondences are used to improve the matching by detecting more correspondences with better precision (see Sec. 5.5). The certainty check is performed as follows: First, a score for $\mathcal{GM}$, from which the schema matching has been derived, is computed. This score is the sum of the average field similarities, which are taken from the average field similarity matrix M:

$$score(\mathcal{M}) = score(\mathcal{GM}) = \sum_{(a,b)\in\mathcal{GM}} M(a,b). \tag{5.5}$$

Afterwards, each correspondence is considered separately. For each correspondence $(\{a\}, \{b\})$ in $\mathcal{M}$, a matrix $M_{(a,b)}$ is constructed, which is identical to the average similarity matrix M, except that the similarity score for (a, b) is set to zero:

$$M_{(a,b)}(x,y) = \begin{cases} 0 & x = a \wedge y = b \\ M(x,y) & \text{otherwise.} \end{cases}$$

Afterwards, the maximum weight matching for $M_{(a,b)}$ is computed. Let $\mathcal{GM}_{(a,b)}$ be the set of edges in that matching, and $score(\mathcal{GM}_{(a,b)})$ be the score of the graph matching as defined above. The correspondence $(\{a\}, \{b\})$ is *certain* if the score of the new matching is not close to the score of the original matching, i.e., if the difference $score(\mathcal{GM}) - score(\mathcal{GM}_{(a,b)})$ is below a threshold $\theta_{certain}$.

If not all correspondences are certain, the table matching process iterates back to the duplicate detection step. In the following, we will describe how certain correspondences can be used to improve the precision of duplicate detection. If in the subsequent schema matching step no additional certain correspondences are detected, the matching $\mathcal{M}$ (including uncertain correspondences) is presented to the user.

5.5 The Extended Tuple Similarity Measure

The tuple similarity measure *tupsim* considers each tuple as a single string, and thus, ignores known correspondences. This can lead to suboptimal results. When some correspondences are known, these should be exploited to achieve better precision. Such correspondences can be produced by a previous run of the DUMAS table matcher as described above or by another schema matcher.

In the following we consider the problem of *duplicate detection in partially aligned relations*: Given two tables R and S, and a subset $\mathcal{M}_{part}$ of the correspondences between R and S, find tuples in R and S that represent the same real world entities. On the one hand, this problem is different than the problem of duplicate detection in unaligned tables, because the attributes which are known to be in the matching can be compared with their corresponding partners. Thus, at least for those attributes Problem 4b described in Sec. 4.3.1 can be solved. On the other hand, the problem is more difficult than duplicate detection in aligned tables, because not all correspondences are know. Thus, it has to be assumed that the unmatched part might contain valuable information that must be used to find duplicates.

To perform duplicate detection in partially aligned relations, we define an extended tuple similarity measure *etupsim*. This measure should reflect the characteristics of the problem discussed above: The attributes that participate in the known matching must be compared with their corresponding partners, and the unmatched part has to be taken into consideration as well.

In the DUMAS table matcher, the extended tuple similarity *etupsim* of two tuples r and s given a partial matching $\mathcal{M}_{part}$ of simple correspondences is computed as

$$etupsim(r,s) = \frac{1}{|\mathcal{M}_{part}|+1}\left(tupsim(r_u, s_u) + \sum_{(\{a\},\{b\})\in\mathcal{M}_{part}} fieldsim(r[a], s[b])\right) \quad (5.6)$$

where r_u and s_u is the unmatched part of r and s, repectively (i.e., the concatenated values of the attributes which do not correspond according to $\mathcal{M}_{part}$). Intuitively, the extended similarity of two tuples is the average of the field similarity scores of their corresponding attributes and the similarity of the unmatched part. Note that each correspondence has the same impact on the similarity score: Because the semantics of the correspondences are unknown, it cannot be determined which attributes are more important for duplicate detection than others. If this were known, one can easily adapt the formula by weighting the correspondences. Also note that the unmatched part has the same impact as each single correspondence. Thus, the more correspondences are known, the less influence the unmatched part has on the extended tuple similarity. If no correspondences are know, then *etupsim* is equal to *tupsim*. Thus, *etupsim* is a generalization of *tupsim*.

Another extension to the similarity measure that potentially increases the accuracy of duplicate detection is to ignore attributes that are known *not to correspond* to other attributes. This kind of knowledge can be easily included

in the definition of *etupsim*: Values of attributes that certainly do not match are removed from the unmatched parts r_u and s_u in Eq. 5.6, i.e., only values of attributes that potentially match are concatenated. The effect of this measure can be demonstrated when considering the scenario of three matching attributes in Sec. 4.6.2: The precision of duplicate detection is much lower than in other experiments because the source contains attributes *Birth-place* and *Birth-district*, while the target has attributes *City* and *District* in its schema. None of those attributes has a corresponding partner, but *Birth-place* and *Birth-district* have the same value domain as *City* and *District*, respectively. If the user could determine that some of those attributes, e.g., *City* and *District*, are not related to other attributes, and consequently remove their values from consideration, Problem 4b would not occur as frequently, and thus, the precision of duplicate detection at early recall levels would increase.

However, we believe that this kind of knowledge is hard to gain. Common schema matching algorithms, including the DUMAS table matcher, only produce correspondences between attributes that are thought to be related, but do not determine if two attributes are unrelated. Ruling out the possibility that unmatched attributes might have corresponding partners contradicts the assumption that the provided matching is only partial.

5.6 Searching For Duplicates with *etupsim*

To find the most similar tuple pairs based on *etupsim* in a reasonable amount of time, an algorithm that finds the top-k duplicates with a minimum number of tuple comparisons must be applied. Although a few correspondences are known, existing algorithms are not an option: They always require user-defined input, e.g., a sort key for the sorted neighborhood method or blocking criteria in record linkage. Instead, we devise a new algorithm that is solely based on the *etupsim* measure. This algorithm has been successfully implemented and tested on synthetic data [Kon06].

5.6.1 The Duplicate Detection Algorithm

A* search has proven to be very efficient in the detection of similar tuples based on *tupsim* both in terms of the number of tuple comparisons and actual runtime. Thus, we decided to use the same principal algorithm for duplicate detection with *etupsim*. The basic search procedure shown in Alg. 1 on page 63 can be directly applied. The only point of change is the constraining phase, which requires the adaptation of the bound function and the restructuring of search space states.

The algorithm starts with a start state, in which both the source tuple and target tuple are not known. In the exploding phase, one intermediate state is created for each source tuple and inserted into the OPEN priority queue. In each iteration of the constraining phase, the state with the largest bound is picked from the queue. If it is a goal state (i.e., both source tuple and target

tuple are known), then it is presented as a result. Otherwise, child states are created using the *children* function.

Recall from Sec. 4.4 that the bound function must define an upper limit on the similarity for each tuple pair that can be derived from a given state. We define the *extended bound function* EB as:

$$EB(r,s) = \begin{cases} \infty & \text{if } r = \perp \wedge s = \perp \\ EB(r) & \text{if } r \neq \perp \wedge s = \perp \\ etupsim(r,s) & \text{if } r \neq \perp \wedge s \neq \perp. \end{cases} \tag{5.7}$$

where r and s are the source and target tuples described by the state. The start state has a predefined bound of infinity, while the bound of a goal state is equal to the *etupsim* similarity of the two tuples. The bound for intermediate states $EB(r)$ is more complex to compute: The source tuple is known, but there are various possible target tuples. Similar to Whirl search, we constrain possible target tuples by excluding certain terms. In contrast to the duplicate detection algorithm for *tupsim*, the new algorithm requires more than one exclusion list: The extended tuple similarity measure is computed based on the field similarity of matching attributes and the tuple similarity of the unmatched part. Because those components are considered independent of each other, we need to maintain several exclusion lists for each state: one for each known correspondence and one for the unmatched part. The semantics of the exclusion lists remains the same: If the exclusion list $e[b]$ of a correspondence $(\{a\}, \{b\})$ contains a term t, then the goal states derived from the given state cannot have a target tuple that contains term t in the value of attribute b. The same holds for the unmatched part. These lists of excluded terms are updated in each invocation of the *children* function.

The bound function for intermediate states $EB(r)$ uses the exclusion lists to compute an upper limit on the similarity of tuple pairs (r, s). It is defined as the average of the upper bound of the corresponding attributes and the unmatched part:

$$EB(r) = \frac{1}{|\mathcal{M}_{part}| + 1} \left(B(r_u) + \sum_{(\{a\},\{b\}) \in \mathcal{M}_{part}} FB(r,a,b) \right) \tag{5.8}$$

where r is the known source tuple and $B(r_u)$ is the bound on the unmatched part as defined in Eq. 4.14. Note that when applied to $EB(r)$, the exclusion list e in Eq. 4.14 only refers to the unmatched part and not the content of the whole tuple.

The upper bound for each corresponding attribute pair requires the consideration of similar terms based on *termsim*. Recall from Sec. 5.2.1 that the field similarity measure incorporates a set $CLOSE$ which contains those source terms that have a "matching" term in the target value, i.e., a term that is equal or highly similar based on *termsim* (Eq. 5.3). If the target value is not known, then we must assume all values of the matching attribute. Given a source tuple r, a source attribute a and its corresponding attribute b, we define $CLOSE(\theta, r[a], b)$ as the set of terms in $r[a]$, for which a term t' in *any* value of attribute b with $termsim(t, t') > \theta$ exists. The terms in $CLOSE(\theta, r[a], b)$ are

considered by the bound function FB, which is defined as

$$FB(r,a,b) = \sum_{t \in CLOSE(\theta, r[a], b)} w(r[a],t) \cdot maxsimweight(b,t). \quad (5.9)$$

The function *maxsimweight* determines the maximum possible value for $w(b,t') \cdot termsim(t,t')$ in Eq. 7.1. It only considers terms t' that are not in the exclusion list $e[b]$ for target attribute b. It is defined as

$$maxsimweight(b,t) = \max_{t' \in SIM(\theta,b,t) - e[b]} maxweight(b,t') \cdot termsim(t,t') \quad (5.10)$$

where $maxweight(b,t')$ is the maximum weight of term t' in attribute b and $SIM(\theta,b,t)$ is the set of terms existing in any value of b with $termsim(t,t') > \theta$ (i.e., the list of "matching" terms for t). If all similar terms are included in the exclusion list, then the function returns zero.

In each iteration of the adapted *children* function, which creates child states, we pick the term from either a matched attribute or the unmatched part that either maximizes Eq. 5.9 or Eq. 4.14, respectively. As in the algorithm described in Sec. 4.4, that term is used to search for target tuples. Depending on where the term originated, the algorithm detects tuples that contain the term in the corresponding attribute or the unmatched part. The extension lists are used to avoid tuple pairs to appear twice in the result.

5.6.2 Finding Similar Terms

The computation of the bound of intermediate states requires the identification of similar terms. When those terms are known, the tuples that contain similar terms can easily be identified using the inverted index on the target table. To the best of our knowledge, there exists no index structure based on *termsim*. However, two means of finding similar terms based on edit distance have been successfully used in the database field: tries and q-grams. In the following, we present those techniques and show how they can be applied to search for *termsim*-similar terms.

Indexing methods for edit distance

The extended tuple similarity measure requires the location of similar terms based on *termsim*. Parsing the whole target table clearly does not scale, and thus, we need an index structure to identify tuples containing similar terms. The existing inverted index is able to detect tuples containing a given term. Hence, to find tuples containing terms similar to a given term t, we use a structure on top of the inverted index that determines similar terms t', which are used by the inverted index. Unfortunately, there exists no such data structure for *termsim*. However, tries or q-grams can be used to identify terms with an edit distance below a given threshold [NBYST01].

Tries[2] have been developed in the area of information retrieval [BYRN99]. A trie is a simple tree structure to store strings: Each tree edge is labelled by a character, and the path from the root of the trie to a leaf node represents the string that is the concatenation of the edge labels. A trie is a very compact representation of a set of strings because same prefixes need to be represented only once: The terms "rumors" and "run" are represented in the trie of Fig. 5.3. Both terms have the prefix "ru", and they share the edges representing this string in the trie. To produce a more compact representation of that tree, one could merge several edges into one if there are no intermediate branches: E.g., edges "r" and "u" could be merged into a single edge representing prefix "ru". Although the algorithm described here is also applicable for such a compact trie representation, for the sake of clarity we only use tries whose edges represent a single character.

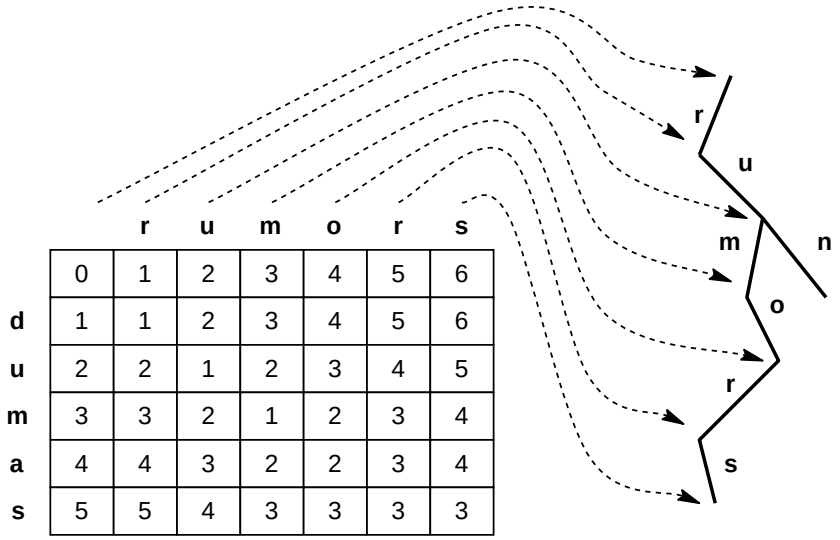


Figure 5.3: Finding similar terms with tries.

Exact search for strings can be performed by a simple traversal of the trie. Approximate search using edit distance is slightly more complicated, but can be done in a backtracking search process: In each search step, the edit distance between the search string and the string represented by the current node is calculated, which only requires the computation of one column in the edit distance matrix [NBYST01]. Fig. 5.3 depicts the edit distance matrix for the search term "dumas" and the term "rumors" in the trie. The first column, which represents the empty string, is associated with the root node. The second column is computed when edge "r" is traversed, the third column after edge "u", etc.

The traversal of a given branch stops if it is guaranteed that the strings represented by the leaf nodes of that branch cannot have an edit distance smaller

[2]The word "trie" is derived from "re**trie**val".

than θ_{ed}. Recall from Eq. 4.2 that scores can only remain the same or increase by one. This property is exploited by the break criterion: The traversal of a given branch stops if the matrix column of the current trie node does not contain *any* score below θ_{ed}. In the example of Fig. 5.3, the traversal would stop at the node representing prefix "rumor" for $\theta_{ed} = 3$, because the smallest value in the corresponding matrix column is 3.

Another indexing technique for edit distance involves q-grams [Ukk92]. A *q-gram* (or *n-gram*) of a term is a substring of length q. Given a term t, its q-grams are obtained by "sliding" a window over its characters. To ensure that the q-grams at the beginning and the end of the string contain q characters, the string is conceptually extended with $q-1$ special characters at both ends. In the following, we will use character '#' for prefixing and character '$' for suffixing terms. E.g., the term "dumas" has 2-grams "#d", "du", "um", "ma", "as", and "s$".

Q-grams have successfully been used in similarity measures. Ukonen presents a string distance measure that is based on the difference of the strings' q-gram sets [Ukk92]. However, q-grams can also be used to provide a lower bound on the edit distance of two strings. The intuition is that strings with a small edit distance share many q-grams [ST96]. E.g., if two strings have an edit distance of 1 due to an insertion, deletion, or substitution operation, then their q-gram sets differ by at most q q-grams. In general, if two terms t_1 and t_2 are within an edit distance of k, then their q-gram sets have at least $\max(|t_1|, |t_2|) - 1 - (k-1) \cdot q$ q-grams in common.

Gravano et al. have exploited this property to implement edit-distance based similarity joins using standard database technology [GIJ+01]. Before a similarity join can be performed, the involved tables need to be preprocessed: For each tuple, the q-grams for the value of the attribute involved in the similarity join are created. The q-grams and their position in the attribute value are stored in a special table, which is used by an SQL query to find tuple pairs that *potentially* have similar terms in the joined attributes using the bound described above. Note that because q-grams only provide a bound on the edit distance, the correct edit distance has to be computed for the tuple pairs produced by the SQL query to remove false positives. However, the edit distance function has to be invoked much less frequently in comparison to its application on all tuple pairs, and thus, exploiting the q-gram bound results in a performance gain.

As we have shown, both tries and q-grams can be used for indexing. However, a q-gram index creates a huge overhead, while tries are a very compact data structure. Hence, we decided to apply trie-based indexing to search for terms with a small edit distance. In the following we show how to search for $termsim$-similar terms given a edit-distance based index.

Translating between edit distance and term distance

The data structures described above can be used to answer the following question: Given a term t, which terms t' exist with $ed(t, t') < \theta_{ed}$. However, our goal is to find terms t' which have a term similarity above a given threshold,

i.e., $termsim(t, t') > \theta$. If we use a data structure based on edit distance, we need to determine the maximum edit distance a term t' can have with respect to a given term t if its term similarity should be above the threshold.

To determine the maximum possible edit distance, we first transform the above inequality:

$$\begin{aligned} termsim(t, t') &> \theta \\ 1 - ned(t, t') &> \theta \\ 1 - \frac{ed(t, t')}{max(|t|, |t'|)} &> \theta \\ \frac{ed(t, t')}{max(|t|, |t'|)} &< 1 - \theta \\ ed(t, t') &< (1 - \theta) \cdot max(|t|, |t'|) \end{aligned} \tag{5.11}$$

As shown in Eq. 5.11, the bound on the edit distance also depends on the length of term t', which is unknown. If the length of term t' increases, the bound on the edit distance also increases, and thus, it seems impossible to determine a maximum edit distance. However, there exists an additional constraint that we can exploit: The edit distance between two terms t and t' is at least as large as the difference in the lengths of the two terms:

$$ed(t, t') \geq ||t'| - |t|| \tag{5.12}$$

W.l.o.g. we assume that term t' is larger than the given term t. Eq. 5.11 and 5.12 then yield

$$\begin{aligned} |t'| - |t| &< (1 - \theta) \cdot |t'| \\ |t'| - |t| &< |t'| - \theta|t'| \\ -|t| &< -\theta|t'| \\ |t'| &< \frac{|t|}{\theta} \end{aligned} \tag{5.13}$$

We now know that the length of the yet unknown string t' is bounded. This knowledge can be used in Eq. 5.11 to determine a bound on the edit distance:

$$\begin{aligned} ed(t, t') &< (1 - \theta) \cdot \frac{|t|}{\theta} \\ ed(t, t') &< \frac{|t|}{\theta} - |t| \end{aligned} \tag{5.14}$$

Eq. 5.14 shows the maximum possible edit distance a term t' can have w.r.t. term t if their term similarity must be above threshold θ. To illustrate, assume the term "dumas", which has a length of 5. Assuming a threshold $\theta = 0.7$, the above equation determines a maximum edit distance of 2.14. Terms t with an edit distance of 3 or larger are guaranteed to be dissimilar to t: If there was a term t with an edit distance of 3 and a *termsim*-similarity above θ, then according to Eq. 5.11 that term must be at least 11 characters long. However, according to Eq. 5.12 such a term has an edit distance of at least 6. Thus, we have shown by contradiction that such a term does not exist.

5.7 Experimental Evaluation

The matching step and duplicate detection procedure need to be experimentally evaluated. We used the same real-estate data that was used in Sec. 4.6.1 to determine the quality of the schema matching step in a real-world scenario (Sec. 5.7.1). Because the results were perfect, several artificial data sets were created using the same data generator as in Sec. 4.6.2 to assess the quality of the detected matching in critical situations. We also performed experiments to evaluate the increase in duplicate detection accuracy of the extended tuple similarity measure with respect to *tupsim*. For each setup, five different pairs of relations were created, and the average of five experiments is reported.

To evaluate the accuracy if schema matching, the precision and recall measures are used:

$$Precision = \frac{|C \cap R|}{|R|} \qquad Recall = \frac{|C \cap R|}{|C|} \tag{5.15}$$

where C is the set of true correspondences and R is the set of retrieved correspondences.

5.7.1 Experiments on Real-Estate Advertisements

Using the ten most similar tuple pairs, the matching between every combination of the four data sets, which were used to assess the accuracy of duplicate detection (Sec. 4.6.1), was established. These matchings were always perfect, i.e., no false correspondences were included and no correspondences were missed. This is not surprising, because

1. the detected duplicates were true duplicates, and
2. the terms used to describe apartments are very similar.

The duplicate detection experiments on the real estate advertisements yielded 100% precision for the top-10 duplicates. Hence, no misleading false duplicates enter the matching step. Using common acronyms and abbreviations to describe an apartment is also beneficial, because the term similarity measure *termsim* is able to handle small deviations or misspellings, but cannot identify synonyms.

In the following, experiments on synthetical data are described. The goal of those experiments is to determine how well the matching step can compensate a few false duplicates. Note that common errors are inserted by the data generator.

5.7.2 Accuracy of Schema Matching

When performing the matching step with the data sets that were used to evaluate the effect of limited extensional overlap (Fig. 4.5), the resulting matching was always perfect. This is not surprising, because most of the top-ranked tuple pairs are true duplicates. Similar results were achieved with the data sets used

to examine the effect of limited intensional overlap (Fig. 4.6). Consequently, additional experiments to study the influence of false duplicates on the schema matching result have been performed.

The schemata of the data sets generated for the following experiment have four corresponding attribute. In fact, they are the same schemata as the four-correspondence case in Sec. 4.6.2, which were used to study the effect of reduced intensional overlap. Data containing three, five, and ten duplicates were used, and K, the number of duplicates used in the matching step, varied accordingly.

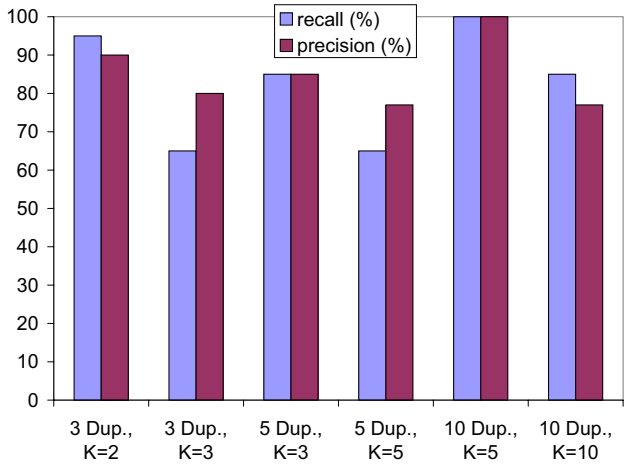


Figure 5.4: Precision and recall of schema matching

Fig. 5.4 depicts the results, which is the average of five data sets, which were independently generated for each number of duplicates. It shows that when the number of duplicates used for matching converges on the number of duplicates in the data set, both precision and recall decreases. This can be expected, because the duplicate detection experiments showed that after 40 - 60% of all duplicates have been detected, false duplicates appear among the top-ranked tuple pairs. Such false duplicates have a negative effect on the schema matching and lead to false and missed correspondences.

As discussed above, false duplicates have a negative effect on the detected correspondences, but a schema matching with reasonable quality can still be expected if only few non-duplicates are used. The next experiment is designed to examine the behavior of the matching procedure with respect to such false duplicates. From a database with 50 duplicates we handpicked the top ten duplicates and performed schema matching based on this perfect choice. Next, we incrementally added the top 20 false positives, i.e., very similar tuple pairs that are known not to be duplicates. The results are presented in Figure 5.5, which shows that recall and precision of schema matching degrade only after as many false positives as true positives are used. The reason for this robustness is that all duplicates support a similar matching, while each false positive might

support a different matching. Both curves level after a certain amount of noise (false positives) is introduced. Precision and recall do not drop till 0% because non-duplicates can still provide information that helps in schema matching: If two tuples representing different persons have the same value for *Birth-place*, then they might miss other correspondences, but still indicate a match for *Birth-place*. Recall that the most similar non-duplicates were used because they would appear in the list of tuple pairs before all other non-duplicates. Since those tuple pairs are very similar, they must have some values in common, and thus, provide enough information for a few correspondences.

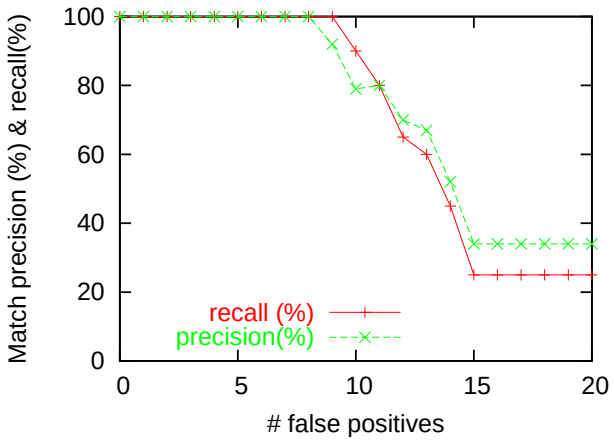


Figure 5.5: Robustness of schema matching with 10 true duplicates in the presence of false positives

5.7.3 Duplicate Detection with Partial Alignment

The goal of exploiting known correspondences in duplicate detection is to increase precision. To show the effectiveness of the extended tuple similarity measure *etupsim*, we used data sets with four corresponding attributes and 50 duplicates.

Fig 5.6 shows that the precision-recall-curve shifts rightward with the number of known correspondences. Intuitively, that means that true duplicates move upward in the list of tuple pairs ranked by decreasing extended tuple similarity. Consequently, more tuple pairs can be used for schema matching without including too many non-duplicates. While the first false duplicate appeared after approx. 45% of all duplicates had been detected with *tupsim* (i.e., no partial match), this number could be raised to 85% by using all four correspondences in *etupsim*. Thus, it can be concluded that the extended tuple similarity measure successfully exploits known correspondences to increase the precision of duplicate detection.

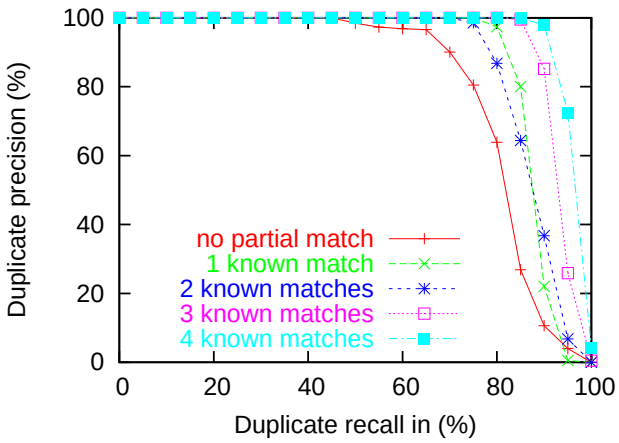


Figure 5.6: Influence of degree of partial alignment

5.7.4 The Certainty Check

Various experiments showed that a second iteration of the algorithm is very rare. In most cases, the certainty check determined that all correspondences can be trusted, making another run of the duplicate detection step unnecessary. In only a single case, a second iteration resulted in an additional correspondence after duplicates have been detected with the extended tuple similarity measure. It can be concluded that duplicate detection with *tupsim* is sufficiently accurate, which is also shown in the experimental evaluation.

Deciding whether to search for additional duplicates is a typical case for a tradeoff between accuracy and performance: Despite the performance optimizations described in Sec. 4.4, duplicate detection is still an expensive operation and should only be performed if necessary. On the other hand, a second iteration has the potential to improve the schema matching result. The experiments indicate that the possible improvement does not justify the cost of detecting more duplicates. However, there might be application domains where the distinction between match and non-match is not as clear as in the data used in the experimental evaluation. In such a case, the user should be included in the matching process. Uncertain matches can be highlighted, and the user must decide whether she is happy with the matching result, or if further refinement is necessary.

5.8 Discussion

This chapter describes the matching step of the DUMAS table matcher. Tuple pairs are compared attribute-by-attribute, and the resulting similarity matrices are aggregated. From such an aggregated matrix, a simple matching is extracted: a set of correspondences such that each attribute corresponds to at

most one other attribute. This notion of simple matching is found in all related schema matching work. However, it is more restrictive than the definition of "simple matching" in Sec. 2.2.2, which states that each correspondence should be a pair of singleton attribute groups. Thus, the DUMAS table matcher only finds matchings in which each attribute is in at most one correspondence, i.e., a matching such as $\{(\{a_1\},\{b\}),(\{a_2\},\{b\})\}$[3] would not be found. However, we believe that in practical scenarios, this is not a major restriction. A straightforward adaptation of the matching step is to skip the graph matching and immediately use all correspondences that pass the pruning threshold. However, as demonstrated in the example, this would also lead to false correspondences, which affect the user's confidence in the resulting matching.

Furthermore, a method for determining which correspondences can be trusted is described. A possible consequence of this certainty check is to detect additional duplicates, which are used to improve the schema matching. For that purpose, an extended tuple similarity measure is applied. The extended measure takes a partial alignment into consideration, which can be produced by a previous iteration of the DUMAS table matcher. Note that the extended tuple similarity measure can also be applied in the first run of the table matcher if certain matches are supplied by an expert or by another schema matcher. In the experimental evaluation we showed that the new measure improves duplicate detection. However, is was also established that the result of the first run of the table matcher was certain in most cases and could not be improved by a second iteration. Further experiments with real world data are necessary to determine in which cases the extra cost of a second duplicate detection procedure is justified.

[3]Note that $\{(\{a_1\},\{b\}),(\{a_2\},\{b\})\}$ is a different matching than $\{(\{a_1,a_2\},\{b\})\}$. The latter is a complex matching, which frequently occurs in real world scenarios.

Part III

Complex Matchings and Complex Schemata

Chapter 6

Matching Complex Schemata

The DUMAS table matcher described in the two previous chapters is able to detect correspondences between two tables if the assumptions described in Sec. 4.2 are fulfilled: The tables must represent the same entity type and must contain duplicates. Obviously, these two assumptions do not hold when comparing two arbitrary tables in complex schemata, which are comprised of multiple relations. The *DUMAS schema matcher* described in this chapter is able to detect correspondences in such complex schemata.

A number of problems have to be solved in the process. Firstly, it has to be determined if the result of the DUMAS table matcher can be trusted, because the two assumptions do not hold when comparing arbitrary tables. The heuristics developed to answer this question affect the quality of the schema matching, because false correspondences have to be avoided. The correspondences that we trust constitute the initial matching. To find more correspondences, the initial matching is used to join tables, which leads to the second problem: One has to decide which table combinations to consider, and which tables must be compared. This is mostly an efficiency issue, because there is an exponential number of table combinations in each schema, and considering all of them clearly does not scale.

6.1 Iterative Schema Matching Using Duplicates

Information can be structured in different ways. That is why we have to deal with semantic heterogeneity when integrating independently developed data sources. When matching single tables, the kinds of differences that have to be considered are limited: Attributes can be given different names or domains, some attributes can be missing, etc. The problem is more challenging in whole schemata, because there is not always a 1:1 relationship between tables: E.g., the entity type represented as a single table in the source schema can be normalized

into several tables in the target schema, or vice versa.

Following a duplicate-based approach for matching complex schemata implies that multi-table duplicates, as described in Sec. 3.5.2, have to be detected. That problem is tackled using the solution for single tables: The problem of finding multi-table duplicates can be reduced to the problem of finding single-table duplicates because the result of joining several relations is also a table, which can be compared to any other relation using the DUMAS table matcher.

Unfortunately, there are very many possible combinations of tables in each schema, and generating and matching all of them clearly does not scale. Furthermore, the table matcher does not always produce a correct result when comparing two arbitrary tables because of the two assumptions made in Sec. 4.2: Our duplicate detection method can be applied if the two tables

1. represent the same entity type and

2. contain duplicates.

Because it is not known if those two assumptions hold when matching two arbitrary tables, it has to be determined how to interpret the result of comparing two tables, or in other words, if the result can be trusted.

Apart from the effectiveness of duplicate detection and schema matching, efficiency is a major concern. It is well known that the number of possible table combinations in a schema is exponential in the size of the schema. This holds even when only "reasonable" combinations of tables are considered, i.e., tables that can be joined by some join path. As join path we consider a sequence of tables that are related. The relationship between tables can be represented in different ways, e.g., foreign key dependencies or join conditions in stored procedures. Because it is not feasible to compare every reasonable table combination, one goal of the algorithm described in this chapter is to (i) compute and (ii) compare only necessary combinations. A table combination is considered necessary if it has the potential to contribute to the schema matching.

A straightforward solution is to immediately apply the DUMAS table matcher on *universal relations*: For each schema, one could create its universal relation, which represents all the information in the database in a single table [Ull89]. After the underlying data have been transformed into the universal relation, the DUMAS table matcher can be applied to find attribute correspondences. Although this solution is technically possible, it was rejected for several reasons. Firstly, the universal relations will be very large, both in the number of attributes and in the number of tuples. The large number of attributes would require us to use a small sample in order to make duplicate detection feasible. This can result in poor precision, because sampling reduces the number of duplicates in the tables (Sec. 4.5). Secondly, experimental results with the DUMAS table matcher presented in the previous chapter indicate that the matching between two tables can be of poor quality if the tables have only few attributes in common. Hence, if the schemata are only slightly related (i.e., they only have a small intensional overlap), the accuracy of the matching between their respective universal relations is likely to be low.

To illustrate the second point, assume that the databases of two online shops — a book shop and a music store — need to be merged. Because both shops sell different types of items, the inventory data of the two databases can be considered (both intentionally and extensionally) disjoint. In particular, the products are described in different ways using different attributes. However, both databases contain customer information, and duplicate entries are likely if the two shops are well known. If the databases are transformed into universal relations, the small intensional overlap — only customer information is represented in both databases — might result in poor duplicate detection precision, and thus, a low-quality schema matching, when using the DUMAS table matcher. However, if the table matcher were applied on certain substructures, namely the customer data, we would expect better matching quality.

Hence, the algorithm described here tries to match those schema substructures that produce a good matching. The *DUMAS schema matching algorithm* involves two main phases:

1. Initial matching,
2. Match extension.

The first step is the creation of an initial matching. The goal of this step is to establish a set of correspondences that we are very confident of. This matching could be established using any available schema matcher. Sec. 6.2 describes the challenges associated with using a duplicate-based schema matching approach to create the initial matching, and an algorithm that applies the DUMAS table matcher to create initial correspondences is formulated.

The second phase of the algorithm is run iteratively. In the first iteration, the initial matching is used to create table combinations. Those table combinations can lead to additional correspondences, which are used in the following iteration to create even larger table combinations. Sec. 6.3 covers the match extension step. We show how to reduce the number of table combinations and comparisons based on the detected correspondences. Possible changes to the algorithm that improve performance but potentially decrease accuracy are discussed in Sec. 6.4.

6.2 Creating an Initial Matching

In the first phase of the DUMAS schema matcher an *initial matching* must be established, which will spark the first match extension in the following phase. Although other definitions are possible, we restrict the initial matching to be a simple matching such that there are no correspondences with the same source table but different target table, and vice versa. In other words, the initial matching consists of several table matchings such that each table participates in at most one table matching.

Fig. 6.1 depicts the scenario used in this chapter in a simplified fashion. The source schema contains five tables 1 through 5, while the target schema has only four tables A though D (depicted as circles). The arrows within a

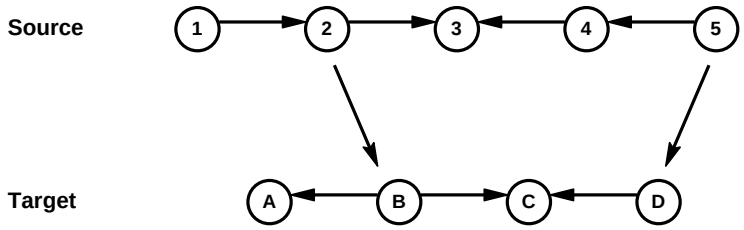


Figure 6.1: An initial matching between source and target schema

schema represent foreign key relationships, which are not of interest in the initial matching phase. Note that although all tables in each schema are related in a connected graph, this is not required by the matching algorithm. Each arrow between the schemata indicates a set of correspondences. As stated above, the correspondences in the initial matching relate a table to at most one other table: In the example scenario, table 2 matches with B and 5 is related to D.

6.2.1 Interpreting the Table Matching

Creating an initial schema matching using the DUMAS table matcher is a challenging problem, because the underlying assumptions do not hold when comparing two arbitrary tables. Hence, it has to be determined if the result of comparing two tables can be assumed to be correct. Various intermediate results created in the matching process can be used. In this section, we will discuss the applicability of the tuple similarity, the field similarity, and the detected matching to determine if the resulting matching between two tables can be trusted.

The tuple similarity measure *tupsim*

The underlying assumption of duplicate-based schema matching is that duplicates provide sufficient information to produce a good matching. A possible approach to determine if a matching can be trusted is to ensure that the detected tuples are, in fact, duplicates. Unfortunately, the tuple similarity measure *tupsim* is not a good indicator. On the one hand, tuple similarity scores for true duplicates can be relatively low, e.g., because the tables under consideration have few corresponding attributes. On the other hand, false duplicates can have high scores. Although it has been shown that *tupsim* is very reliable when the tables represent the same entity type, false duplicates with high similarity scores can be returned by the algorithm if this assumption does not hold.

An example for this problem is depicted in Fig. 6.2. The source **R** contains information about products, whose name and producer are represented in table *Product*. The different editions of the products, including the edition number *Ed* and the year, can be found in table *Edition*. The arrow represents a foreign key relationship between *Edition* and *Product*. In contrast, the target **S** contains statistical data about a soccer tournament: Information about each player

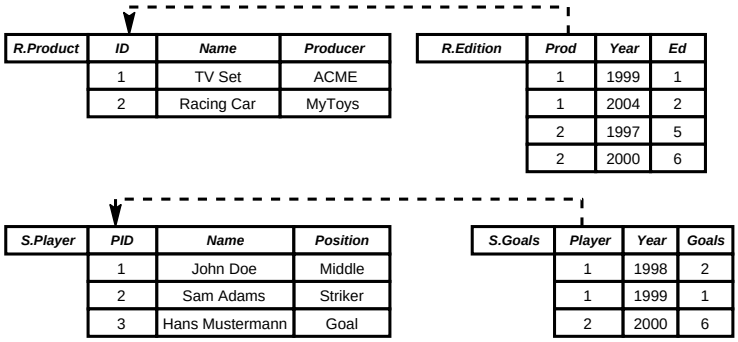


Figure 6.2: Misleading tuple similarity: Tables with different semantics

is provided in table *Player*, while the number of goals scored by a given player in a given year in shown in table *Goals*. Although tables *Edition* and *Goals* have completely different semantics, one notices that a few tuples look very much alike: E.g., the tuples "1 1999 1" and "2 2000 6" (represented as a single string as required by the similarity measure) can be found in both tables. Using those tuple pairs in the matching step would result in false correspondences.

Such high scores occur if two tuples have many terms in common — a situation that is described as Problem 4b in Sec. 4.3.1. Several factors contribute to high scores:

1. *Same attribute domain*: If two attributes have the same domain (e.g., *Year* in the example above), there are likely to be some unrelated tuples which have the same value for those attributes. The probability that two attributes have the same value is larger if the attribute domain has few values (e.g., *Ed* and *Goals*).

2. *Few values*: If tuples have only few values, spuriously matching values have a large effect on the tuple similarity. Few values occur in two cases:

   (a) *Few attributes*: Assume that two tuples have one matching attribute value. The effect of this value match on the tuple similarity is larger if the respective relations have only few attributes, because in that case there are less (non-matching) values that decrease the tuple similarity score.

   (b) *Many null values*: This case is similar to the case of few attributes. The less values exist, the less information we have to distinguish unrelated objects. There are two ways to interpret null values in the duplicate detection step: One can ignore them, i.e., remove null values from the string representation, or consider `null` a special token. Experiments have shown that the difference between both approaches is negligible: The weight of a `null` token is close to zero if many tuples have at least one `null` value, and thus, the null value has a minimal effect on the tuple similarity.

In general, the larger the strings resulting from tokenizing the tuples are, the less likely large tuple similarity scores occur. However, because even duplicate tuples can have low similarity scores, the tuple similarity measure is not a good indicator.

The field similarity measure $fieldsim$

Apart from the tuple similarity measure, another intermediate result is the similarity matrix for each duplicate, which contains field similarity scores for each attribute pair. Assuming that the tuples are true duplicates, such a matrix is a good indicator for existing correspondences.

However, a matrix is constructed only for tuple pairs that are very similar with respect to *tupsim*. Hence, the problems described above also affect the similarity matrices. Thus, a *single* similarity matrix can lead to false conclusions when deciding if a matching can be trusted.

The detected attribute correspondences

In contrast to the similarity measures, which work on a single tuple pair, the detected attribute correspondences are the result of *several* tuple pairs. Thus, a correspondence that is part of the resulting schema matching indicates that many tuple pairs have the same of very similar value in the corresponding attributes.

If spurious matches in attribute values occur randomly, it is unlikely that the average similarity score for any attribute combination passes the matching threshold, and thus, the resulting matching will be empty. A false correspondence will be extracted only if many non-duplicates have similar values for a given attribute combination. E.g., the most similar tuple pairs for tables *Product* and *Player* in Fig. 6.2 are those with the same value for attributes ID and PID, respectively. In this case, both attributes have the same value domain, and the values of other attributes are unrelated. Consequently, all detected tuple pairs have the same value for those two attributes, resulting in a false correspondence.

As shown in the above example, it is possible that a small number of false correspondences is extracted from the tuple pairs. However, *many* correspondences are less likely to be established between unrelated tables, because they would imply that the majority of the tuple pairs have highly similar values for *all* those attribute pairs. Consequently, table matchings are judged based on the number of its constituent correspondences, which is stated in the following *matching trust heuristic*: A table matching with many attribute correspondences is more trustworthy than a table matching with few attribute correspondences. Note that by "many" we refer to the absolute number of correspondences. Thus, we inherently prefer larger tables.

The matching trust heuristic defined above does not tell whether a matching can be trusted, it only says if a matching can be trusted more than another matching. However, it has been shown to be sufficient for removing

false matches, and thus, increasing the precision of our duplicate-based schema matcher. It is directly applied in the initial matching procedure (Sec. 6.2.2), but can also be found in the iterative part of the algorithm (Sec. 6.3).

6.2.2 Initial Matching with the DUMAS Table Matcher

The initial matching created by the DUMAS schema matcher is based on the matchings for every table combination. Let $\mathcal{M}_{i,j}$ be the matching between tables R_i and S_j, and $|\mathcal{M}_{i,j}|$ be the size of the matching, i.e., number of correspondences in the matching. The matching sizes for every table combination can be represented in a *matching size matrix*. Fig. 6.2.2 shows the matrix for two schemata with three tables.

	S_1	S_2	S_3
R_1	**7**	0	1
R_2	0	**5**	2
R_3	1	**6**	0

Figure 6.3: A matching size matrix

As stated above, only matchings with a large number of correspondences can be trusted. We define *maxcor* as the maximum number of correspondences detected by comparing every source table with every target table:

$$maxcor = max_{R_i \in \mathbf{R}, S_i \in \mathbf{S}} |\mathcal{M}_{i,j}|. \tag{6.1}$$

In the example matrix in Fig. 6.2.2 the maximum number of correspondences is 7. In the following, we only consider the set of large table matchings $\mathcal{LM}$ that contains matchings $\mathcal{M}_{i,j}$ whose size is close to *maxcor*, i.e.,

$$\mathcal{LM} = \{\mathcal{M}_{i,j} : |\mathcal{M}_{i,j}| \geq \theta_{numcor} \cdot maxcor\}$$

where θ_{numcor} is a threshold in the range [0,1] set by the user. The initial matching is created from the table matchings in $\mathcal{LM}$. With a large threshold, only few table matchings are used in the initial matching. This could decrease the recall of the overall algorithm. Decreasing the threshold allows more correspondences to enter the initial matching, but could also decrease precision. In the example a threshold $\theta_{numcor} = 0.6$ was used. All table matchings that have more than the resulting minimum number of correspondences of 4.2 are highlighted in bold.

Afterwards, ambiguous table matchings are removed from $\mathcal{LM}$: In very rare cases a table can have large matchings with more than one other table. E.g., the matchings of table S_2 with both R_2 and R_3 pass the threshold. This is in contrast to the goal of creating a set of table matchings such that each table has correspondences with at most one other table. One way to resolve this conflict is to produce a maximum weight graph matching similar to the matching step in Sec. 5.3.2 to relate tables. If that approach were followed, the initial matching would assign R_1 to S_1 and R_3 to S_2.

However, another goal of the initial matching step is to avoid false correspondences. If a table X has many correspondences with more than one other table, then it is unclear which table it should be assigned to. Choosing one table matching based on a graph matching includes the risk of using false correspondences in the following phase. To assure that only true correspondences are used, we decided to ignore all matches of table X. The set of remaining table matches $\mathcal{IM}$ is set set of large table matchings that are certain:

$$\mathcal{IM} = \{\mathcal{M}_{i,j} \in \mathcal{LM} : \neg\exists \mathcal{M}_{k,l} \in \mathcal{LM} : (i = k \wedge j \neq l) \vee (i \neq k \wedge j = l)\} \quad (6.2)$$

In the above example, $\mathcal{IM}$ would only contain the table matching between R_1 and S_1. If the set $\mathcal{IM}$ were empty because of this procedure, then a maximum weight matching will be computed on the matching size matrix, and the matches above the threshold that are part of that graph matching constitute $\mathcal{IM}$. The final initial matching $\mathcal{M}_{initial}$ between the source schema and the target schema is the union of the table matchings in $\mathcal{IM}$:

$$\mathcal{M}_{initial} = \bigcup_{\mathcal{M} \in \mathcal{IM}} \mathcal{M}. \quad (6.3)$$

Note that although the initial matching is based on the matchings between all tables, it is not necessary to consider every table combination. It is obvious that the size of the matching can only be as large as the smaller table, i.e., $|\mathcal{M}_{i,j}| \leq min(|R_i|, |S_j|)$. Hence, only tables whose size is larger or equal to $\theta_{numcor} \cdot maxcor$ (where $maxcor$ is the current maximum number of correspondences) need to be considered. In the DUMAS implementation, the algorithm starts with the largest tables and suspends computation if no more table pairs need to be considered.

6.3 Crawling Through Schemata

The basic idea of the second step of the algorithm is to use the initial matching to "crawl" through each schema: The tables that participate in the initial matching are joined with their neighboring tables to produce larger tables. These new tables are matched with other tables to produce additional correspondences, which are used to further extend tables, and thus, start the next iteration.

In the following, the tables that are present in the original schemata are called *base tables*, while the tables that are the result of joining several tables are called *join tables*. In cases where the algorithm does not distinguish between the two types of tables, the term *table* is used.

6.3.1 A Run Through The Example

To demonstrate the intuition behind the DUMAS schema matching algorithm we will go through the example depicted in Fig. 6.1. The figure shows initial correspondences between tables 2 and B and between 5 and D. In the following,

we say that two tables *match* if there are correspondences between them, and denote a matching by an arrow: $2 \rightarrow B$ and $5 \rightarrow D$.

Based on this matching, tables 2, 5, B, and D are *extended*: Join tables are created by merging those tables with their respective neighbors. A *neighbor* of a table is another table in the same schema that is related to the given table. Without loss of generality, it is assumed that it is known how two tables are related. In the implementation of the DUMAS schema matcher, foreign key dependencies are extracted from the metadata repository of the database. More elaborate methods, such as analyzing queries or the database content [DJMS02], are also conceivable.

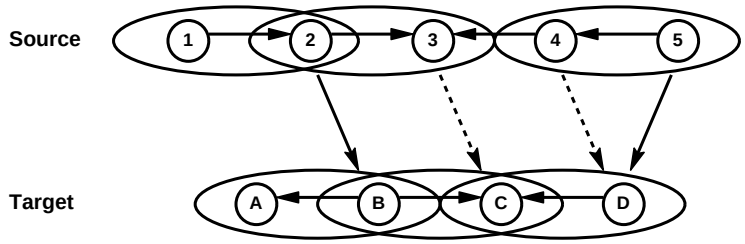


Figure 6.4: Join tables and new correspondences

Fig. 6.4 shows the created join tables as ellipses. In the following, we denote the result of joining table X with table Y as XY. One can see that the initial matching leads to the construction of six join tables: 12, 23, 45, AB, BC, and CD. Those join tables should be compared with tables in the other database, respectively, to improve the matching. To facilitate efficient schema matching, table comparisons must be restricted to tables that have a chance of producing additional correspondences. The decision which tables to compare is based on the known matching: Because table 2 matches with table B, we compare tables 12 and 23, which are derived from 2, with B and tables that were derived from B (i.e., AB and BC). Tables 12 and 23 are not compared to any other table in the target schema, because at this point there is no known relationship to any table other than B. The same reasoning applies to all join tables in the source and target schemata. The left hand side of Fig. 6.5 shows the comparisons performed in the first iteration.

Using the DUMAS table matcher on tables 23 and BC results in additional correspondences between attributes of 3 and C, which are depicted as a dashed arrow in Fig. 6.4. Their target attributes are in table C, but could only be detected by creating join table BC. Hence, we say that table 23 matches with BC. Comparing join table 45 with tables D and CD also resulted in new correspondences. The target attributes of those new correspondences are part of table D. Because there are no correspondences with C, we conclude that table 45 matches with D and not CD.

The new correspondences are used in the next iteration to produce larger join tables, which are are also subject to table matching. The table compar-

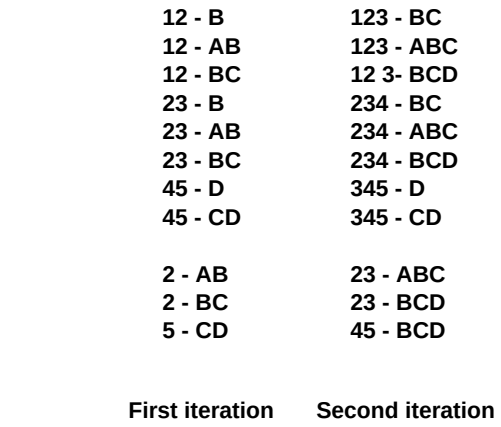


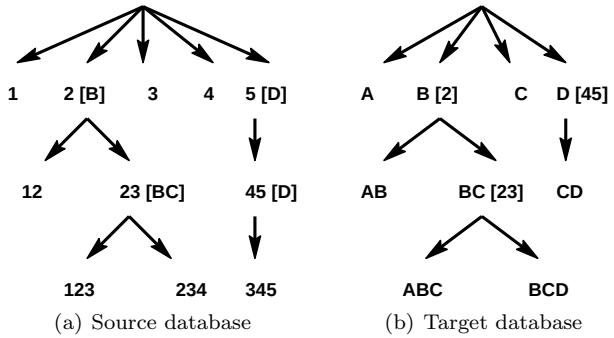
Figure 6.5: Table comparisons performed in the running example.

isons performed in the first and second iteration of the example are depicted in Fig. 6.5. This process stops when no more correspondences can be expected.

The description of the example exhibits two main aspects of the extension step: (i) creation of new join tables and (ii) assigning those join tables to other tables. The latter means that each table (either base or join table) has at most one matching partner table. This assignment aspect is similar to the initial matching phase, where each base table is matched to at most one base table.

6.3.2 The Derivation Tree

The matching algorithm is based on *derivation trees*, which describe how a join table was created, i.e., which table was extended to create a given join table. Fig. 6.6 shows the derivation trees of the source and target schemata of the running example. The name of a node is the same as the name of the table represented by the node.



(a) Source database (b) Target database

Figure 6.6: Derivation trees of the source and target databases

The derivation tree of the source database is shown in Fig. 6.6(a)). The child of a node X represents a table that is created as an extension of table X. E.g., node 12 is the child of node 2 because its table has been created by extending table 2. The children of the root node represent the base tables, while nodes further below in the derivation hierarchy represent join tables. Note that a single join table can be created in more than one way, i.e., by extending more than one table. Assume that the initial matching contained table matchings $2 \rightarrow B$ and $3 \rightarrow C$ (Fig. 6.4). In that case, both extending table 2 and table 3 lead to the construction of table 23. Although this join table is created only once, it is represented by two nodes in the derivation tree — one that is the child of 2 and one that is the child of 3 — which are considered separately by the algorithm. Thus, it is ensured that the derivation tree is a tree. As a consequence of this rule, a single join table can have multiple states: Each node of 23 represents a role of the table (e.g. "derived from 2" or "derived from 3"), and both nodes can have a different activation status[1], which affects the table comparisons performed by the schema matcher.

Some nodes in Fig. 6.6 are annotated by the name of their *corresponding* or *matching node*, i.e., the node in the opposite tree that it is assigned to. Recall from Sec. 6.3.1 that each table is assigned to another table. Fig. 6.6(a) shows that 2 is assigned to B (as in the initial matching) and 23 corresponds to BC. Note that node D is the target tree matches with node 45, although in the initial matching D was related to 5 (Fig. 6.6(b)). However, in the first iteration of the match extension step, join table 45 is assigned to D, and the target tree is adapted accordingly. Beside the matching node, the set of correspondences (i.e., the matching) between the tables is also stored. This is required by the algorithm described in Sec. 6.3.3 to evaluate the matching computed for child nodes.

To summarize, each node contains:

1. A reference to the table, which is represented by the node (mandatory). The function $table(n)$ is used to reference the table of node n.

2. A reference to a partner node, i.e., the node in the opposite derivation tree, which the node is assigned to (optional). The function $partner(n)$ is used to reference the partner node of node n.

3. A node matching, i.e., a set of correspondences between the node's table and the table of the partner node (optional). The function $matching(n)$ is used to reference the matching assigned to node n.

The partner node and the matching are optional because not all tables have a matching partner.

[1] Active and inactive nodes are discussed below.

6.3.3 The Schema Matching Algorithm

Algorithm 2: DUMAS Schema Matching Algorithm

Input: Source schema s; Target schema t
Output: Schema Matching between s and t

```
1  match ← initialMatch(s, t);
2  sourceTree ← new DerivationTree(s, match);
3  targetTree ← new DerivationTree(t, match);
4  repeat
5      nodeMatches ← ∅;
6      lastMatch ← match;
7      extend(sourceTree); extend(targetTree);
8      if numPending(sourceTree) + numPending(targetTree) == 0 then
       Break;
9      foreach pend in getPending(sourceTree) ∪ getPending(targetTree)
       do
10         parent ← parent(pend); pPartner ← partner(parent);
11         ppMatch ← match(table(pend), table(pPartner));
12         if sanityCheck(ppMatch) then
13             nodeMatches ← nodeMatches ∪ ppMatch;
14         end
15         nodeMatches ←
           nodeMatches ∪ childMatches(pend, pPartner, ppMatch);
16     end
17     nodeAssignment ← maximumWeightMatching(nodeMatches);
18     foreach match in nodeAssignment do
19         sNode ← getSource(match); tNode ← getTarget(match);
20         matching(sNode) = match; matching(tNode) = match;
21         if isLeaf(sNode) then activate(sNode);
22         if isLeaf(tNode) then activate(tNode);
23     end
24     match = globalMatching(sourceTree);
25 until |lastMatch| ≥ |match|;
26 return lastMatch;
```

The algorithm to compute a matching between complex schemata is summarized in Alg. 2. At the beginning the initial matching is constructed as described in Sec. 6.2.2 (line 1). Afterwards, the derivation trees for the source and target schemata are constructed (lines 2 and 3).

Lines 4 through 25 describe the iterative part. It starts by initializing two variables: *nodeMatches* is a set of matchings between nodes, which are constructed in the current iteration. At the beginning of the iteration it is empty (line 5). The variable *lastMatch* contains the schema matching that has been constructed in the previous iteration. In the first iteration, the initial matching

is used (line 6).

Afterwards, active nodes of the source tree and the target tree are extended (line 7). A node in a derivation tree is an *active node* if it has produced new correspondences in the previous iteration. Active nodes can easily be extracted from a derivation tree: Active nodes are leaf nodes which have a partner node and a node matching. A node is extended in two steps: First, new join tables are created by joining the node's table with neighboring base table. Neighboring tables are base tables that are not part of the node's table but are connected to one of its constituent base tables by a foreign key dependency. For each of the neighboring tables, a new join table is created that is the result of joining the node's table with the neighboring table[2]. Second, a child node that is the child node of the active node is created for each new join table. The newly created nodes are called *pending nodes*. The iteration stops if the number of pending nodes is zero (line 8). This could occur in small schemata, when the active nodes represent the whole schema, and thus, no base tables can be joined.

Matching Extended Tables

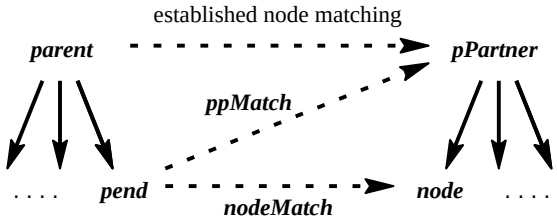


Figure 6.7: Comparing relevant tables.

After the creation of pending nodes, each of their tables is compared with some tables in the other database (line 9 - 16). As stated in Sec. 6.3.1, only reasonable comparisons should be performed. Given a pending table, we only want to compare it with tables that were derived from the partner of its parent. Thus, to determine which tables need to be compared, the parent of a pending node *parent* (which is the active node that has been extended to create the current pending node) and its partner node *pPartner* are considered (Fig. 6.7).

First, a matching *ppMatch* between the table of the pending node and the table of the parents partner is established using the DUMAS table matcher (line 11). A sanity check is performed to determine if the correspondences can be trusted (line 12). The details of the sanity check are described in Sec. 6.3.4. If the matching passes the sanity check, then the node matching is added to *nodeMatches* (line 13).

Second, node matchings between the current pending node and the child nodes of *parent* are constructed (line 15). The function that computes the set

[2]If such a join table exists already, e.g., because it has been produced by extending another node, then it is reused.

Function `childMatches`

Input: Pending node *pend*, parent's partner $pPartner$, table matching $ppMatch$ between *pend* and $pPartner$

Output: Set of table matchings between *pend* and tables derived from $pPartner$

```
1 foreach node in getDescendants(pPartner) do
2     nodeMatch ← match(table(pend), table(node));
3     if sanityCheck(nodeMatch) ∧ |nodeMatch| > |ppMatch| then
4         nodeMatches ← nodeMatches ∪ nodeMatch;
5     end
6 end
7 return nodeMatches;
```

of node matches is shown as Function childMatches. The process is similar to the matching with the parent's partner, except that the node matching *nodeMatch* is added to *nodeMatches* only if it passes the sanity check and contains more correspondences than the matching with the parent's partner node *ppMatch*. E.g., the node matching between tables 45 and CD does not fulfill this additional constraint because the node matching between 45 and D ($ppMatch$) contains the same correspondences. Consequently, the node matching $45 \rightarrow CD$ is not further considered.

Creating Node Assignments

After all pending nodes have been processed, *nodeMatches* contains matchings between nodes where at least one of them is pending. Now partner nodes have to be assigned to pending nodes based on their node matchings (line 17). The goal is to produce a 1:1 assignment, i.e., each pending node is assigned to at most one node in the opposite derivation tree, which is not required to be pending. This problem related to finding a 1:1 matching of attributes based on field similarity scores, and hence, a similar solution is applied: A *node assignment matrix* with one row for each source node and a column for each target node is constructed. The value of a matrix cell is based on the matching between the respective derivation tree nodes. Similar to the initial matching step, the scoring is based on the assumption that a matching can be trusted if it contains many correspondences. Consequently, the value of a matrix cell is the sum of the number of correspondences in the matching between the respective nodes, which is contained in *nodeMatches*, and the average field similarity scores of corresponding attributes, which is a value in the interval (0,1]. The latter is added to the size of the matching to break ties. If there is no matching between the nodes in *nodeMatches*, then the cell value is set to zero.

Fig. 6.8 shows the node assignment matrix for the running example (zero values are left for reasons of clarity). Table 12 does not create additional correspondences, and thus, does not pass the sanity check. Table 23 is a successful

	A	B	C	D	AB	BC	CD
1							
2							
3							
4							
5							
12							
23					7.8		
45				4.6			

Figure 6.8: The node assignment matrix for the running example.

extension because, when matching it with table BC, correspondences between attributes of base table 3 and base table C are detected. One can see that the matching between table 23 and BC contains 7 correspondences with an average similarity score of 0.8. Table 45 is also a successful extension. However, its matching with table CD is not larger than its matching with table D, and thus, the former is not further considered. Pending tables on the target side are processed accordingly.

Interestingly, the example matrix contains entries only for the correct assignment. This might not always be the case: Several alternative assignments could be possible. To resolve this issue, a maximum weight matching is performed on the node assignment matrix. The result is a set of source node – target node pairs, where at least one of the two nodes is a pending node.

The set of node pairs in the graph matching represents the node assignments, which must be reflected in the derivation trees. In addition, some nodes become active while others become inactive (lines 18 - 23). Given a node pair (sn, tn) of the graph matching, where sn is a source node and tn is a target node, the partner node of sn is set to tn, and vice versa. This is also necessary for non-pending nodes: E.g., node D is initially assigned to 4, but is reassigned to 45 in the following iteration. Both nodes' matching is set to the matching between sn and tn.

After partner nodes and matchings have been determined, some nodes need to be activated. If a node in pair (sn, tn) is a leaf node, then it becomes an active, non-pending node, which will be extended in the following iteration. The activation of nodes is restricted to leaf nodes because non-leaf nodes have already been extended. All pending nodes that do not have a partner become regular nodes, i.e., neither active nor pending.

Resolving Match Conflicts

Note that although the node assignment is a proper 1:1 matching of nodes, it might not be a 1:1 matching of attributes. Some nodes overlap in their base tables. E.g., nodes 12 and 23 overlap because both contain table 2. Assume that

both nodes have a partner. In that case, their matchings could assign different target attributes to a given attribute of 2. To resolve this issue, an attribute matching is produced (line 24).

The attribute matching step is similar to the matching step described in Chap. 5. First, a matrix is constructed with a row for each source attribute and a column for each target attribute. The score of a matrix cell should reflect how much we believe that those attributes correspond. Again, the heuristic that matchings with many correspondences are more reliable is used in the scoring method. All node matchings which involve at least one active node are considered. A node matching $\mathcal{NM}$ between nodes sn, whose table is constructed from base tables $R_1, \ldots, R_m$, and tn, whose table is constructed from base tables $S_1, \ldots, S_n$, is broken down into $m \cdot n$ *base matchings* between their constituent base tables. E.g., the matching between 45 and D is split into two base matchings between 4 and D and between 5 and D. A base matching $\mathcal{M}_{ij}$ between tables R_i and S_j contains only the correspondences of $\mathcal{NM}$ whose source attribute is in R_i and whose target attribute is in S_j. For each such correspondence $(\{a\}, \{b\})$, the size of $\mathcal{M}_{ij}$ (i.e., the number of correspondences) is added to the value of the cell in the matrix that represents the attribute pair (a, b).

After all node matchings have been processed as described above, a maximum weight matching is performed. The resulting attribute pairs are transformed into a matching *match*. If that matching contains more correspondences than the matching of the previous iteration $lastMatch$, then the next iteration is started. Otherwise, the matching of the previous iteration $lastMatch$ is the final result of the DUMAS schema matcher.

6.3.4 The Sanity Check

As pointed out above, the table matching algorithm can produce misleading results when comparing unrelated tables. To reduce the likelihood of false correspondences, the initial matching step produces only matchings with many correspondences. In each iteration, a large number of table matchings are produced, and it has to be to determined which of those matchings can be trusted. However, in contrast to the initial matching phase, there is additional information which this decision can be based on, namely the matching produced in the previous iteration. In the following, the heuristics used to determine if a matching is correct are described.

The sanity check procedure has to decide if a matching *match* between a pending node *pend* and another node, which is either the partner of the pending node's parent *pPartner* or a child *node* of *pPartner*, can be trusted (Alg. 2). It uses the *parent matching*, which is the matching between *parent* and *pPartner*, to evaluate *match*. The sanity check's decision is based on two heuristics:

1. *Correspondence Retention*: All correspondences contained in the parent matching must also be contained in *match*. In other words, no correspondences must be lost by extending tables.

2. *Match Extension*: The matching *match* must be larger than the parent matching. If *match* does not contain more correspondences than the parent matching, the the extension is not successful, and thus, the matching should be dropped.

The first heuristic is based on the assumption that the parent matching is correct. Hence, if some of its correspondences cannot be found in the child matching, the *match* must be flawed. The match extension heuristic reflects the idea that additional correspondences are to be found by extending some tables. If an extension does not lead to additional correspondences, then the extension was useless. If correspondences are lost, i.e., *match* has fewer correspondences than the parent matching, then it is assumed that *match* is incorrect. Hence, it is required that matchings have more correspondences than their parent matchings in order to pass the sanity check.

As stated above, the correspondence retention heuristic is based on the assumption that the detected correspondences are correct. However, it is not uncommon that a few attributes are mismatched, even though the tables are correctly assigned. In a special case, the first heuristic does not need to hold, and we allow a child matching to 'overrule' its parent matching:

3. *Attribute Reassignment*: If more attributes of the parent's partner have a corresponding attribute in the child matching, then correspondence retention heuristic does not need to hold and the child matching is considered correct.

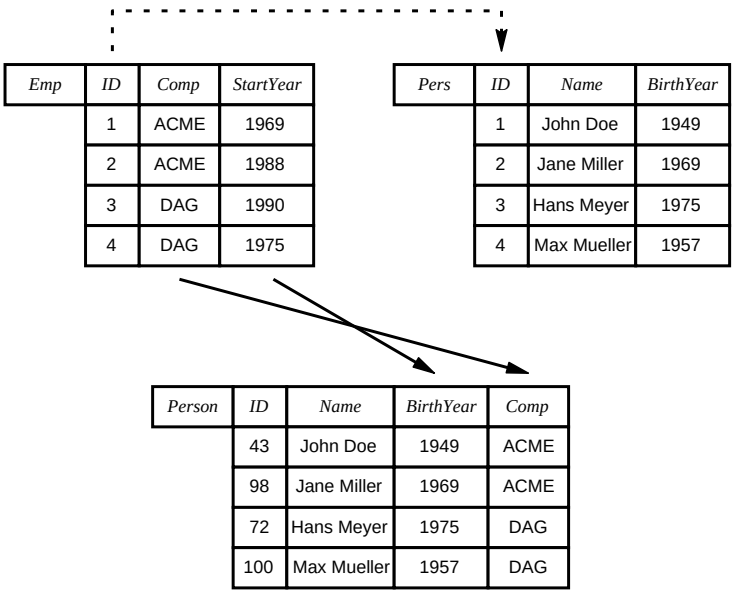


Figure 6.9: Initial matching with one false correspondence

To illustrate the attribute reassignment, consider the scenario in Fig. 6.9: Both source and target contain information about persons. Source table $Pers$ contains the name and birth year of persons, while table Emp has employment data about those persons, namely the name of the company they work for and the year they started in that company. In the target database, this information is contained in a single table $Person$, except that there is no data about the start year. Assume that the initial matching $Emp \rightarrow Person$ contains the correspondences indicated by arrows in Fig. 6.9. Note that there is a false correspondence between $StartYear$ and $BirthYear$, which has been created because the most similar tuple pairs $(1, 98)$ and $(4, 72)$ have equal values in those attributes.

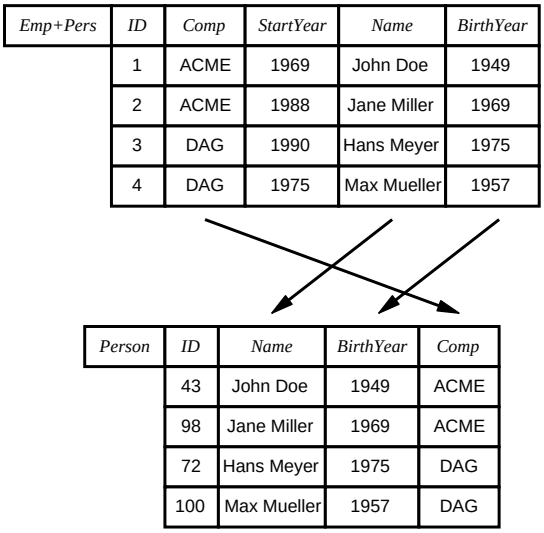


Figure 6.10: Correct matching after extending table Emp.

After extending table Emp by joining it with table $Pers$, the correct matching $Emp + Pers \rightarrow Person$ is produced (Fig. 6.10). The matching contains three correspondences, and thus, satisfies the second constraint. However, the correspondence between $StartYear$ and $BirthYear$, which is part of the initial matching, was dropped. Instead, $BirthtYear$ in $Emp + Pers$ is correctly assigned to $BirthYear$ in $Person$. Hence, the correspondence retention requirement is not met.

However, when comparing the new matching with the parent matching $Emp \rightarrow Person$, one notices that more attributes of the parent's partner have a corresponding attribute in the source[3]. Hence, we consider the new matching a better matching, and allow it to overrule the parent matching.

The initial matching related table Emp with table $Person$. Note that the

[3]Note that in this case the parent's partner is equivalent to the matching table of $Emp + Pers$, because the target database contains only a single table.

tuple pairs $(1, 98)$ and $(4, 72)$, which are used to construct the matching, are no true duplicates. In fact, both tables represent different entity types. One might even argue that the tuples in table *Emp* have no identity and are mere annotations for the persons represented in $Pers$. However, as that is not known to the DUMAS schema matcher, the algorithm has to allow for erroneous matchings. By facilitating an overruling of previous matches, we take into account that the table matcher can produce false correspondences when its assumptions do not hold.

6.3.5 Considering Complex Relationships by Deferred Deactivation

For the sake of exposition we have assumed in Sec. 6.3.3 that the algorithm should stop extending a table if the previous extension has not been successful: When a join table does not produce any additional correspondences, the sanity check will fail, and hence, the matching will not be part of $nodeMatches$. Because of that, the pending node is not activated at the end of the iteration, and thus, will not be further extended.

This is a too restrictive decision, because certain design decisions require the algorithm to make two or more steps to detect additional correspondences. A very common example is an m:n relationship between two entity types, which is usually represented as three tables in the database. On the other hand, one might also create only a single table, if redundancy is not an issue. Assume two databases containing employment information. The source database has a table $Person$ containing data about employees, a table $Comp$ listing companies, and an association table $WorksFor$ connecting those two tables. The association table only contains foreign keys to the primary keys of the other two tables. In the target database, this m:n relationship is represented as a single table $Employment$. Assuming that the initial matching involves table $Person$, joining tables $Person$ and $WorksFor$ does not yield additional correspondences, because the association table does not contain relevant information. If the algorithm stopped at this point, it would miss the correspondences involving table $Comp$.

Instead of discontinuing the extension of a table immediately after it has not been successful, the deactivation is deferred until it has been unsuccessful in the last k steps. In the implementation k is set to 2, which allows for the common representation of m:n relationships as three tables. The algorithm is adapted accordingly: A pending node is activated if it contains a matching or if its predecessors with a distance $k-1$ have a matching. Thus, if k is set to two, a leaf node is activated if the node itself or its parent has an attached node matching. In addition, the parent matching, which is used in the sanity check, is the matching of the closest predecessor that actually has a matching. Furthermore, nodes are reactivated not only when they are part of a node assignment, but also when a predecessor with a distance smaller than k is part of an assignment.

6.4 Table Extension vs. Duplicate Extension

Finding duplicates in two unaligned tables is a complex procedure: Its inherent complexity is $O(n^2)$, because each source tuple must be compared to each target tuple. Using the Whirl algorithm to detect the most similar tuple pairs drastically reduces the actual number of tuple comparisons. Nevertheless, duplicate detection is still an expensive operation, in particular when the tables contain many tuples.

The schema matching algorithm described above uses the DUMAS table matcher to compare two tables. Although only tables that can be expected to contain duplicates are compared, the number of table comparisons can become large. Besides table comparison, the actual creation of join tables is another performance issue. Again, the algorithm aims at reducing the number of join tables. However, the effort of constructing join tables is considerable.

One way to improve efficiency is to reduce the size of the tables. *Table extension*, i.e., joining whole tables, is an expensive operation if tables are large. A possible adaptation of the algorithm to improve performance is *duplicate extension*: Assume that the initial matching between tables 2 and B was established using k duplicates. In the extension step, the tuples of table 2 that are part of the k duplicates are used to construct join tables 12 and 23 by joining those tuples with tables 1 and 3, respectively. Join tables AB and BC are created in the same way. Because the resulting join tables are much smaller than those created by table extension, less data has to be written to the database, and thus, the construction of join tables is much faster. The performance of duplicate detection is also improved, because less tuples have to be compared. In general, extending duplicate tuples as opposed to whole tables can be considered a major efficiency improvement.

Unfortunately, similar to other problems in computer science, there is a tradeoff involved. Extending the 'best' tuple pairs can result in join tables whose most similar tuple pairs are not as similar as the most similar tuple pairs that would have been detected when extending whole tables. Consequently, some correspondences can be missed.

The underlying reason for the loss in accuracy is variable data. To illustrate the problem, consider the example depicted in Fig 6.11. Assume that the initial matching (depicted by solid arrows) is created using duplicates $(1, 43)$ and $(3, 72)$ of the $Pers$ tables. Note that the birth year of a person never changes, and the name does not change frequently. Joining those tuples with the $WorksFor$ tables results in tuples where the $Comp$ attribute, which represents the company the persons work for, has different values for duplicate tuples. This is not unusual, because people change their employers, and the two data sources might have created their data at different points in time. Because the $Comp$ values of duplicate tuples have different values, the correspondence between the two $Comp$ attributes cannot be detected by the DUMAS table matcher.

Another problem of duplicate extension is the possible lack of join partners for duplicate tuple pairs: If a duplicate tuple is part of a referenced table, then there does not need to be a tuple that has a foreign key to that tuple. As

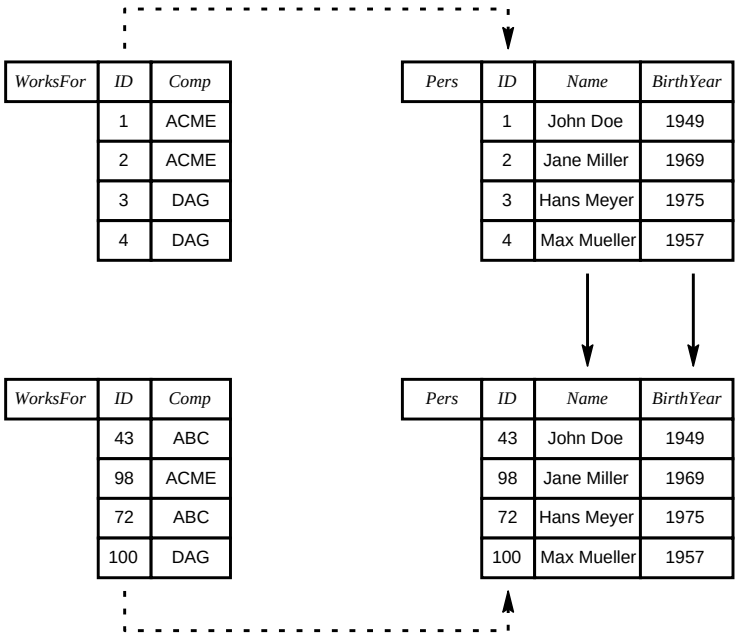


Figure 6.11: Extending duplicates: tuples with outdated information.

a consequence, extending duplicates might require several database operations to search for tuples that have join partners, which could easily become more expensive than a single table join.

Because accuracy of schema matching can decrease when duplicate extension is applied, we chose to perform table extension. However, alternative solutions are conceivable. When the size of the schemata under consideration are too large, the cost of schema matching using the above algorithm might become prohibitive. In that case, it is suggested to create join tables as described above with the $n \cdot k$ most similar tuple pairs, where n can by any number larger than 1. Using a large n increases the size of the join tables, but also reduces the risk of missing attribute correspondences. Unfortunately, this procedure does not guarantee that all $n \cdot k$ duplicate tuples have a join partner.

6.5 Experimental Evaluation

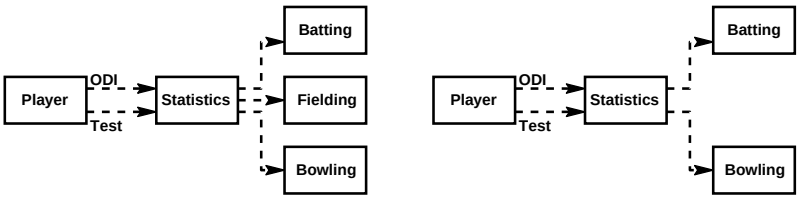
The algorithm described above has been implemented and tested on real-world data. The experimental results are reported in this section.

6.5.1 Implementation and Data

The DUMAS schema matching algorithm has been implemented in Java 1.4. The data used in the experiments reside on a IBM DB2 UDB V8.1 database,

which is accessed by the implementation using JDBC. We store join tables created by the algorithm as declared global temporary tables in the database. Statistics for each table, which are required by the similarity measures of the DUMAS table matcher, are collected only once and stored on the file system.

To assess the effectiveness of the DUMAS table matcher, two data sets containing information about cricket players were used. These data sets have previously been used in [DLD+04] to evaluate the iMap schema matcher (see Sec. 2.3.4). The original data had been extracted from two web sites Cricinfo[4] and Cricketbase[5] and transformed into XML format. Because the DUMAS schema matcher works on relational schemata, the data had to be transformed into the relational model such that the schemata closely reflect the original XML structure.



(a) Cricinfo schema (b) Cricketbase schema

Figure 6.12: Cricket schemata

Fig. 6.12 depicts the schemata of the two cricket databases. Tables are represented as boxes and foreign keys as dashed arrows. Each player has (optional) statistics about one-day internationals (ODI) and / or test games. Statistics are comprised of bowling, batting, and (only in Cricinfo) fielding statistics. The number of attributes in each table and the number of correspondences are shown in Fig. 6.13. Note that some of the correspondences involve complex value transformations, which are not detected by the DUMAS matcher. Thus, in addition to the overall number of correspondences, the number of correspondences involving transformations is provided in parentheses.

Table	Cricinfo	Cricketbase	Correspondences
Player	21	26	8(1)
Batting	9	7	6
Fielding	2	N/A	N/A
Bowling	11	9	8(1)
Sum	43	42	22(2)

Figure 6.13: Number of attributes in schemata and number of correspondences.

[4]http://www.cricinfo.com/
[5]http://www.cricketbase.com/

6.5.2 Accuracy of Schema Matching

The goal of the experiment is to evaluate the result of schema matching using the previously applied precision and recall measures. In addition, the F-Measure is used to get a combined score for both precision and recall:

$$Precision = \frac{|C \cap R|}{|R|} \qquad Recall = \frac{|C \cap R|}{|T|} \qquad F\text{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

where C is the set of true correspondences and R is the set of correspondences returned by the matching algorithm.

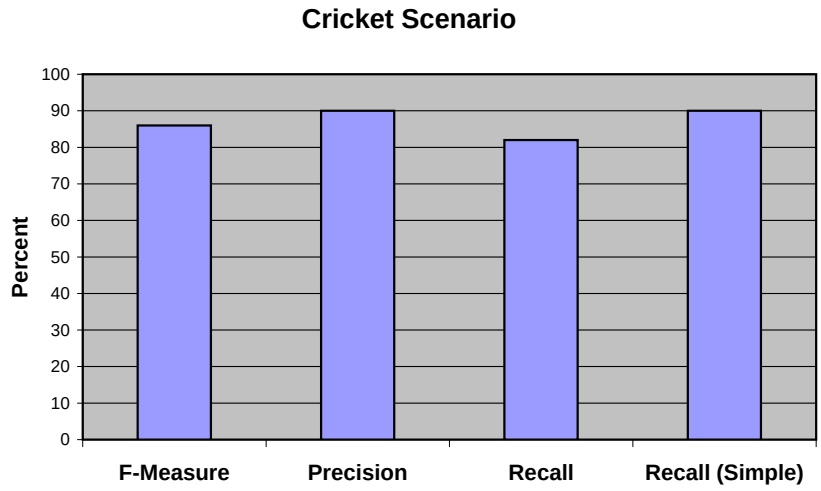


Figure 6.14: Matching quality.

Fig. 6.14 shows the schema matching accuracy of the DUMAS schema matcher on the cricket data. The high precision of 90% suggests that no false duplicates have contributed to the final matching. This assumption has been confirmed by visual examination of intermediate results. Two recall measures are provided: one that is based on all true correspondences and one that considers only simple correspondences, i.e., correspondences that do not require value transformations. The numbers suggest that our matching algorithm performs well with respect to simple correspondences (recall = 90% for simple correspondences), but can be improved when value transformations are considered (recall = 76% for all correspondences). The resulting F-Measure is 82%. Note that the results show an improvement over previously reported results of less than 80% accuracy for the same data set [DLD+04].

Note that the final outcome did not depend on the number of corresponding tables in the initial matching: When the algorithm starts with a matching between *Player* tables, it produces the same result as when it starts with correspondences between *Player*, *Batting*, and *Bowling* tables. The small size of the schemata certainly contributed to this positive outcome.

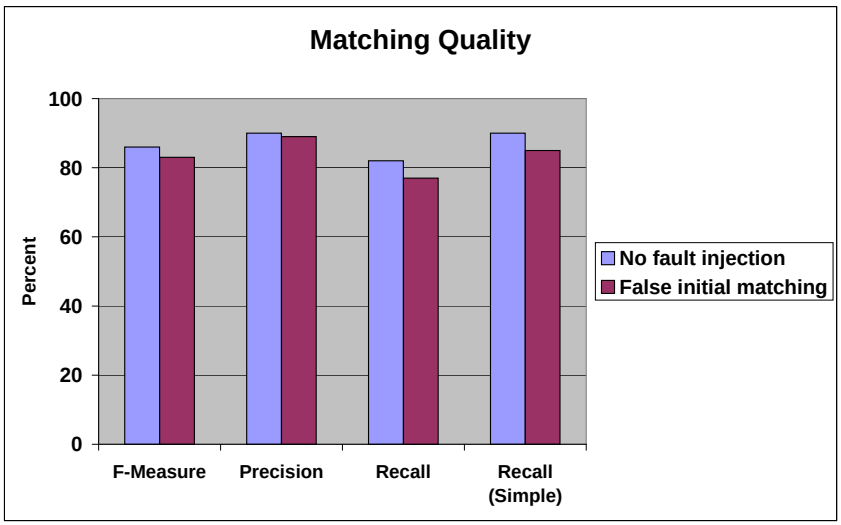


Figure 6.15: Matching quality with false initial matching.

The initial matching produced by the algorithm described in Sec. 6.2 never contained false table pairs. To examine how the algorithm handles false duplicates, we started the extension phase with an incorrect initial matching between *Batting* in one schema and *Bowling* in the other. The result of this experiment is depicted in Fig. 6.15 (see 'False initial matching'). The quality of the matching was surprisingly good. The reason for this is the increase in duplicate detection accuracy when the *Player* tables are joined in. The matching extracted from those duplicates contains more correspondences for the previously matched tables. As a result, the *attribute reassignment* rule described in Sec. 6.3.4 is used to overrule the previous matchings. We point out that the use of false correspondences comes at the price of additional iterations, and thus, increased running time.

6.6 Discussion

The DUMAS schema matcher, which is capable of finding attribute correspondences in complex schemata, is described in this chapter. To the best of our

knowledge, it is the first duplicate-based schema matcher that works on whole relational schemata: Other matchers compare only two tables or, in the case of iMap, XML data, for which duplicates have to be manually provided.

As a first step, an initial matching is created. Such an initial matching relates only a subset of all tables. One goal of this phase is to avoid false correspondences, which could mislead the algorithm. Hence, a very conservative strategy is applied: Only table matchings with a large number of correspondences are used. A global threshold, which depends on the maximum number of correspondences, is applied for that purpose. Although the sanity check procedure has shown to be very effective in overruling false correspondences, we believe that a conservative approach is more promising in order to achieve a quality schema matching.

Based on the initial matching, tables are extended to produce additional correspondences. Those correspondences are used to create more tables, thus, starting the process again. It is obvious that a possibly large number of tables has to be created and compared. The heuristics described in this chapter aim at reducing both the number of join tables and the number of table comparisons. Furthermore, a strategy to drastically reduce the size of join tables is described, which makes the algorithm applicable in very large databases. Although this strategy comes with the cost of possible decrease in schema matching accuracy, the user should be given the choice between efficiency and quality. However, it has to be kept in mind that schema matching is an off-line process, i.e., it is performed during the development of a system and not at runtime, and thus, efficiency is a secondary goal.

Chapter 7

Finding Complex Matchings

The DUMAS table matching algorithm is designed to detect simple correspondences, which align one source attribute with one target attribute. While 1:1 correspondences are very common, 1:n or m:n correspondences also occur in practice: E.g., the name of a person can be represented as a single attribute or several attributes for given name, middle initial, and surname. This chapter describes the DUMAS complex matcher that facilitates the detection of such complex correspondences. An analysis of the existing algorithm reveals that some components of the existing algorithm can be reused, and only the matrix construction of the matching step needs to be adapted. We describe a general search procedure that creates all possible matrices. Because the number of matrices is far too large for real-world scenarios, various optimizations that facilitate the applicability of the DUMAS complex matcher in practice are discussed. Finally, the algorithm is experimentally evaluated.

7.1 Problems Associated With Complex Matchings

The problem of finding complex matchings has only recently been addressed by the research community. Early schema matching solutions considered only 1:1 correspondences, because it is assumed that in most scenarios the majority of existing correspondences are simple. However, complex matches do occur and must be detected by a schema matcher if it is to be applied in practice. Recall from Sec. 2.2.2 that a correspondence is of the form $(\{a_1, \ldots, a_m\}, \{b_1, \ldots, b_n\})$, where each a_i is a source attribute and each b_j is a target attribute. Simple correspondences align single attributes (i.e., $m = 1$ and $n = 1$), while complex correspondences relate several attributes (i.e., $m \geq 1$ and $n \geq 1$). Fig. 7.1 depicts a very common example for a complex relationship: While the source

table R contains separate attributes for first name (FN) and last name (LN), the target table S has only a single attribute for the full name (N). Hence, if the data of the source table is to be moved to the target table, the values of FN and LN need to be concatenated to create values for N. In addition to names of persons, both tables contain address information ($A1$ and $A2$). Thus, the correct matching is $\mathcal{M} = \{(\{FN, LN\}, \{N\}), (\{A1\}, \{A2\})\}$.

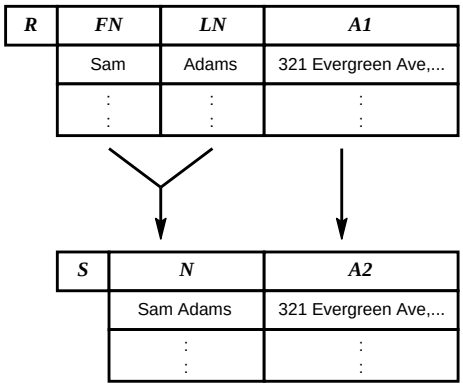


Figure 7.1: A complex matching between R and S.

Although complex matchings exist in many scenarios, the majority of existing algorithms only detect simple correspondences. One reason for the lack of complex matchers is the added complexity when attribute sets larger than one need to be considered. The *space of possible matchings* between two is drastically increased: If only simple correspondences (singleton attribute groups) are considered, then $|R| \cdot |S|$ simple correspondences between tables R and S are possible. If the bound on the size of the attribute groups is removed, then the number of possible correspondences becomes exponential in the size of the tables: Given a relation R with $|R|$ attributes, there are $2^{|R|} - 1$ possible combinations of attributes if order is not relevant[1]. Hence, there are $(2^{|R|}-1)\cdot(2^{|S|}-1)$ possible complex correspondences between tables R and S.

While the search space is large but bounded, the number of possible *functions* to combine attribute values is infinite. Although the schema matching output is only comprised of attribute sets and not the functions to combine attribute values, we note that applying different functions in the matching process can improve the quality of the matching. We do not address this problem in this thesis and only consider concatenation of strings. However, we point out that the adapted matching step, which is described in the following, can also be applied with other functions.

[1]Given a set S with n elements, its power set contains 2^n elements. The power set also includes the empty set, which we wish to exclude in our considerations.

7.2 Adapting the DUMAS Table Matcher

Table matching is comprised of two steps, which are subject to change in order to facilitate the detection of complex correspondences. The *duplicate detection* step uses the tuple similarity measure *tupsim* to find duplicates. Recall that this measure ignores the record structure and is order-independent. Consequently, it can also be used to detect duplicates for complex matching if concatenation of string values is the only considered attribute combination function: In the example in Fig. 7.1, concatenating the first name 'Sam' with last name 'Adams' yields the string 'Sam Adams', which is equal to the value of column N in the target table. The source attributes FN and LN can be combined in any order, because 'Sam Adams' and 'Adams Sam' are considered equal by the tuple similarity measure.

The existing duplicate detection procedure can also be applied when the extended tuple similarity measure *etupsim* is used. The unmatched part is compared using tupsim, and thus, no changes are necessary. Corresponding attributes are compared using the field similarity measure $fieldsim$. Because this measure is also order-independent, *etupsim* can be used when the known partial matching contains complex correspondences.

The *matching* step is a different issue: Because the algorithm described in Chap. 5 is not designed to find complex matches, it will detect the correspondence between the address attributes and probably a false correspondence involving attribute N. Fig. 7.2 depicts the average similarity matrix for the example scenario with the weights of edges that are the result of maximum weight matching highlighted in bold. Depending on the pruning threshold used in the matching step, the incomplete correspondence $(\{LN\}, \{N\})$ could enter the matching.

	N	$A2$
FN	0.6	0.1
LN	**0.7**	0.1
$A1$	0.1	**1**

Figure 7.2: Average similarity matrix for example tables.

Because the existing procedure cannot find complex correspondences, the matching step needs to be adapted. Fortunately, some of the techniques can be reused:

1. *Field similarity measure*: The field similarity measure $fieldsim$ is order-independent, and thus, can also be used to compare attributes when their values are concatenated.

2. *Similarity matrices*: The construction of the similarity matrices for each duplicate has to be adapted, because combinations of attributes have to be considered. As in the case of simple matchings, an average similarity matrix can be constructed based on the similarity matrices for each duplicate.

3. *Maximum weight matching*: We will show how the same maximum weight matching process can be applied to detect complex correspondences.

To summarize, the complex matching step is very similar to the matching step for simple matchings. The main difference is the consideration of combinations of attributes in the construction of similarity matrices. Hence, in addition to single attributes, the matrices for each duplicate must contain rows and columns for *attribute groups*, which are combinations of attributes. Given a tuple r of relation R, the value of an attribute group $A \in 2^{att(R)}$ is a single string that contains r's space-separated values of the attributes in A. Note that a single attribute can be considered a singleton attribute group. In the following, the terms 'attribute' and 'singleton attribute group' are used interchangeably.

Given that a tuple's value of an attribute group is simply a concatenation of attribute values, the construction of the similarity matrices for each duplicate is straightforward. If all matrices have the same structure (i.e., each matrix represents the same attribute groups for both tables), then the average similarity matrix can be computed as described in Sec. 5.3.1.

7.3 Searching For Complex Correspondences

The complex matching algorithm is designed under the premise that most components of the existing matching step can be reused. In particular, the field similarity measure and the methods for creating, aggregating of, and reasoning on similarity matrices have been shown to be very effective, and thus, should be applied in the search for complex correspondences.

As described above, the main difference is the construction of the similarity matrices for each duplicate, and consequently, the average similarity matrix, on which maximum weight matching is performed. The structure of those matrices is different than in the simple case because attribute groups have to be considered. We discuss two ways of structuring similarity matrices and describe the search space that has to be traversed by the algorithm in the following. Finally, we identify several problems that have to be solved by the matching algorithm.

7.3.1 The Matrix Structure

In addition to single attributes, which essentially are singleton attribute groups, similarity matrices need to represent attribute groups consisting of more than one attribute. In the following we assume that the construction is based on the start matrix (Fig. 7.2), which contains only singleton attribute groups and is constructed as described in Chap. 5.

Sec. 2.3.4 describes the complex matching approach by Chua et al. [CCL03]. They *extend* tables by adding rows and columns for attribute groups consisting of two or more attributes. As we have shown, this approach can lead to false correspondences because attribute groups may overlap.

Because of the drawbacks of the extension procedure, we decided to follow a *merge* strategy: Instead of adding additional rows and columns for larger

	$\{N\}$	$\{A2\}$
$\{FN, LN\}$	**1**	0.1
$\{A1\}$	0.1	**1**

(a) Correct grouping: $\{FN, LN\}$.

	$\{N\}$	$\{A2\}$
$\{FN, A1\}$	0.25	**0.6**
$\{LN\}$	**0.7**	0.1

(b) Incorrect grouping: $\{FN, A1\}$.

Figure 7.3: Merging attributes to detect a complex matching.

attribute groups, those new attribute groups replace their constituent attributes in the merge strategy. Fig. 7.3(a) shows the matrix that is the result of merging attributes FN and LN. The rows for those two attributes have been replaced by a row for their union $\{FN, LN\}$. This strategy has the advantage that the existing maximum weight matching procedure can be applied to detect complex correspondences: Because there is no overlap between attribute groups, the problems described in Sec. 2.3.4 do not occur.

7.3.2 Searching for a Matrix Structure

The general idea of the algorithm is to start with a simple matching (as depicted in Fig. 7.2) and subsequently merge some attributes if the merging improves the matching. Fig. 7.3(a) shows the matrix after attributes FN and LN have been merged into an attribute group. Choosing those two attributes for merging has been a good choice because the only complex correspondence in this scenario requires those two attributes to be concatenated. One can easily observe the improvement in the table matching, which is represented as bold numbers in Fig. 7.3(a): There are still two correspondences, but in contrast to the start matrix, both have a score of 1. In contrast, Fig. 7.3(b) shows the matrix with attributes FN and $A1$ being combined. The bold numbers indicate that the matching has degraded.

Before describing the space of possible matrices, we define a merge function $merge(M, A, B)$ that takes a matrix M and merges attribute groups A and B, resulting in a new matrix M'. Obviously, the origin of both A and B must be either the source or the target table. If the attributes of A and B are source attributes, then the structure of M' is the same as the structure of M except that the rows for A and B have been replaced by a row for $\{A, B\}$. The computation of the values for the new row is straightforward: Given a target attribute (or attribute group) C, the field similarity $fieldsim(A \cup B, C)$[2] between attribute group $\{A, B\}$ and C is computed for each duplicate tuple pair. To do so, the values of attributes A and B are concatenated and compared with the value of C using the field similarity value. The value of $M'(\{A, B\}, C)$ is the average of the field similarities of the duplicates. The process is analogous if A and B are target attribute groups.

[2]Because $fieldsim$ compares bags of tokens representing attribute values, combining two attribute values essentially resolves to creating the union of their token bags. Hence, we denote the combination of attribute values as $A \cup B$ in $fieldsim$.

The search through the space of possible matrices is based on the merge function. Fig. 7.4 depicts all matrices that can be derived from the start matrix in Fig 7.2. The grey cells represent the edges that are part of the graph matching and, after pruning low-weight edges, would constitute the complex table matching. Each arrow indicates one application of the merge function.

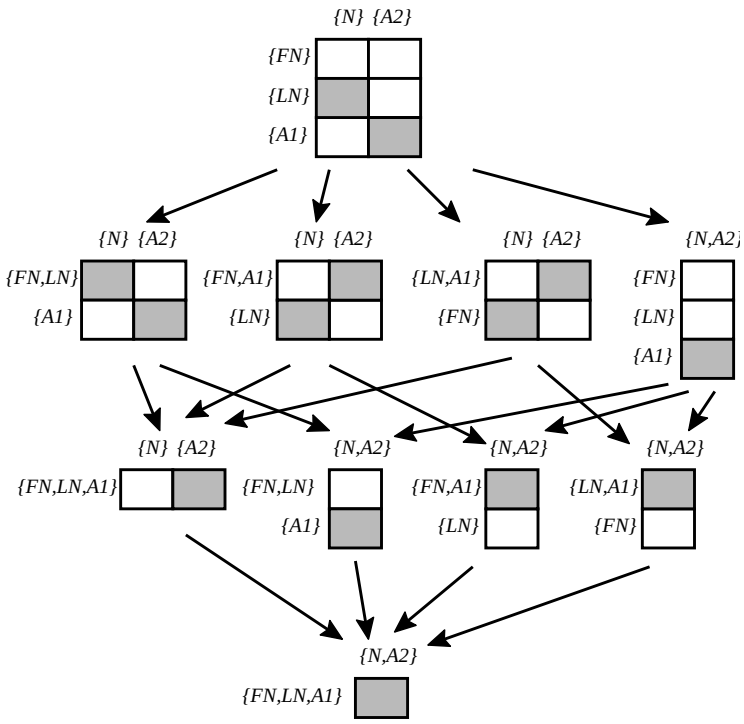


Figure 7.4: The complete search space of the running example.

As stated above, the general idea of the match algorithm is to search though the space of possible matrices and pick the 'best' matrix and its inherent graph matching, which represents the resulting complex correspondences. Although the idea sounds simple, two challenges have to be tackled by the algorithm:

1. *Score function*: The score of a matrix needs to reflect the 'quality' of the matching with respect to other matrices, i.e., better matrices must have the higher scores.

2. *Number of attribute combinations*: In each search step an exponential number of possible merges exist. Assume that a matrix M has m source attribute groups and n target attribute groups. There are $\begin{pmatrix} m \\ 2 \end{pmatrix}$ matrices that can be derived from M by merging source attributes and $\begin{pmatrix} n \\ 2 \end{pmatrix}$

matrices that can be created by merging target attributes. Altogether, there are $B(|R|) \cdot B(|S|)$ possible matrices for tables R and S, where $B(n)$ is the Bell number (i.e., the number of possible partitions) for a set with n elements [Rot64]. This number increases exponentially with n: E.g., $B(10) = 115,975$, while $B(100)$ is a 116-digit number.

Although only a few duplicates and not the whole tables are used to compute the matrices, reducing the number of matrices that have to be considered by the algorithm is crucial to facilitate its applicability in scenarios where tables contain many attributes. The complex matching algorithm, which is described in the next section, uses various criteria to prune the search space and only use matrices that improve the matching.

7.4 Detecting 1:n Matchings

The complex matching algorithm described in Sec. 7.4.1 applies a breadth-first search procedure to traverse the space of possible matrices. As pointed out before, this space is very large, and thus, measures to reduce the number of matrices that are considered have to be taken. Sec. 7.4.2 describes various heuristics to prune the search space. Measures to asses the improvement in the matching in each search step are discussed in Sec. 7.4.3.

7.4.1 Discovering the Best Matrix

The complex table matching algorithm, which is shown in Alg. 4, begins the search for the best matrix with the start matrix, which is the start state for the search process (line 1). This matrix is constructed as described in Chap. 5. The variable $newMatrices$ represents all matrices produced in the current search step. Initially it contains the start matrix (line 2). The 'best' matrix discovered until a given point is kept as $bestMatrix$, which is also the start matrix at the beginning (line 3).

Lines 4 – 14 describe the search process. The search continues only if additional matrices have been generated in the previous iteration (line 4). All of those matrices are considered in the current search step (line 5). We also have considered *beam search* as the underlying search procedure, which would only consider the best k matrices in each search step, and thus, reduce the search space. However, the heuristics described in the following section have shown to be very effective, rendering additional pruning unnecessary. We also need to compare the currently best matrix $bestMatrix$ with the best matrix created in the previous iteration. If there has been an improvement, we need to update $bestMatrix$ (line 6).

Each of the matrices is used to generate child matrices, which are constructed by merging attribute groups (line 9). Recall that only child matrices that improve the matching are created. Those matrices are added to $newMatrices$ and used in the subsequent search step to construct additional matrices (line 11).

Algorithm 4: DUMAS Complex Table Matching Algorithm

Input: Source table R; target table S
Output: Complex matching between R and S

```
1  M ← startMatrix(R, S);
2  newMatrices ← {M};
3  bestMatrix ← M;
4  while newMatrices ≠ ∅ do
5      matrices ← newMatrices;
6      bestMatrix ← best(bestMatrix, newMatrices);
7      newMatrices ← empty;
8      foreach M ∈ matrices do
9          childMatrices ← childMatrices(M);
10         if childMatrices ≠ ∅ then
11             newMatrices ← newMatrices ∪ childMatrices;
12         end
13     end
14 end
15 GM = graphMatch(bestMatrix);
16 M = prune(GM);
17 return M
```

The search is discontinued if no more matrices that improve the matching can be generated. The matrix with the highest match score is considered the 'best' matrix. Based on this matrix, a maximum weight matching is performed (line 15). The result is a set of edges between nodes representing source and target attribute groups such that each node is incident with at most one edge, and each edge connects one source with one target attribute group. Edges with a weight below a given threshold θ_{prune} are pruned (line 16). The remaining edges represent the table matching, which is the output of the algorithm.

It has to be noted that other search procedures are conceivable. Beam search can be applied if the heuristics described in Sec. 7.4.2 prove to be insufficient for very large tables. The adaptation of the existing implementation would be straightforward: Instead of using all matrices in each search step, only the best k matrices are considered (line 5).

7.4.2 Creating Child Matrices

Child matrices are created by merging attribute groups. Because the number of possible matrices is very large, we use three heuristics to reduce the number of matrices that are considered by the algorithm:

1. *Partner attribute groups*: Given an attribute group A, we consider an attribute group P a partner (or partner attribute group) of A if A is highly similar to P (i.e., P potentially matches with A). When considering

attribute group A in the algorithm, new attribute groups involving P are created.

2. *Local criterion*: Merging the partner P of A with another attribute group B must increase the similarity with A. Otherwise, it is not considered an improvement. This criterion called 'local' because only A, P, and B are considered.

3. *Global criterion*: The resulting matrix must improve its parent. As opposed to the local criterion, the global criterion considers all attribute groups.

Those heuristics can be found in the function `childMatrices`, which generates child matrices for a given matrix M. The structure of those matrices is the same as the structure of M except that two attribute groups have been merged (Fig. 7.4).

Function `childMatrices`

Input: Average Similarity Matrix M
Output: Matrices derived from M by merging attribute groups

```
1  foreach attribute group X in M do
2      partners ← simAttGroups(M, X);
3      foreach P ∈ partners do
4          foreach attribute group Y ≠ P do
5              if avgSim(X, P ∪ Y) > avgSim(X, P) then
6                  M' = merge(M, P, Y);
7                  if score(M') > score(M) then
8                      childMatrices = childMatrices ∪ M';
9                  end
10             end
11         end
12     end
13 end
14 return childMatrices
```

A matrix M is given as input to `childMatrices`. The function considers each attribute group, both from the source and from the target side (line 1). For each attribute group X, it picks *partner attribute groups* (line 2). Those partners are attribute groups in the opposite table that are potential matches for X (i.e., their similarity passes a threshold $\theta_{partner}$). If more than m attribute groups pass the threshold, only the m most similar attribute groups are used as partners. If no attribute group passes the threshold, then *partners* is empty.

For each partner group P, the new attribute group $\{P, Y\}$, where Y is an attribute group on P's side (except P itself), is considered a potential improvement. To determine if the matrix involving attribute group $\{P, Y\}$ needs to be

constructed, the *local criterion* is applied: The average similarity of X values with the concatenation of P and Y values must be higher than with P values (line 5). If the local criterion is satisfied, the matrix M' is created by merging attribute groups P and Y (line 6). This child matrix must also satisfy the *global criterion*: The overall match score of the child matrix M' must be higher than the match score of its parent M (line 7). If that is the case, then the matrix is added to the result set (line 8).

To illustrate the method, assume M to be the start matrix shown in Fig. 7.2 and X to be the target attribute group $\{A2\}$. To determine the set $partners$, the similarity of $A2$ with all source attribute groups needs to be checked. As stated above, the m most similar attribute groups with a similarity score above the threshold $\theta_{partner}$ are taken as partners. In the following, we assume that $m = 2$ and $\theta_{partner} = 0.2$. Because only $A1$ is highly similar to $A2$, $partners$ contains the attribute group $A1$. This attribute group is merged with all other source attribute groups, resulting in the following new attribute groups: $\{FN, A1\}$ and $\{LN, A1\}$. In contrast, if X is the target attribute N, both LN and FN are taken as partners.

The average field similarity of those new attribute groups and X is computed using the detected duplicates. One will see that the similarity of $\{FN, LN\}$ with N is larger than the similarity of FN with N and of LN with N. Hence, the local criterion is satisfied. In contrast, attribute group $\{LN, A1\}$ leads to a decrease in the similarity score with respect to $A2$, because the last name LN is not part of the address $A2$.

With only three source attributes, the effect of using only a limited set of partner groups is marginal. When the tables under consideration become larger, the factor m can be used to regulate the computation time of the algorithm. In contrast, the introduction of the threshold $\theta_{partner}$ has increased both accuracy and performance: The performance gain is obvious, because less attribute combinations are considered for each X. Interestingly, accuracy is also positively affected: If the similarity $avgSim(X, P)$ is very low, then the similarity $avgSim(X, P \cup Y)$ can coincidentally be higher even though P and Y are unrelated. Thus, if no threshold were applied, the child matrix could pass the local criterion. None of the examined score functions for the global criterion would satisfactorily filter out the majority of such 'false' merges.

The Local Criterion

Although it is possible that the local criterion filters out correct merges (e.g., due to dirty data) or consider false merges as being correct (e.g., due to coincidental use of the same terms), in most cases it makes a correct decision. In the following, we show why the criterion works with the field similarity measure $fieldsim$.

The local criterion is based on the assumption that merging unrelated attributes or attribute groups decreases the average similarity: LN is very similar to N, because the given name is part of the full name. Merging LN with $A1$ decreases its similarity to N, because a lot of terms from the address $A1$ that

are unrelated to the name N are added. On the other hand, merging related attributes increases the similarity score: The concatenated FN and LN values are much more similar to N values than the FN and LN values themselves. Essentially, this behavior reflects the intuition behind any string similarity measure: If a pair of strings is more similar than another pair of strings, then it must get a higher similarity score. Consequently, the algorithm makes a greedy decision if a merge step decreases a similarity score and drops the respective matrix.

To illustrate the behavior of the field similarity measure, we start with the following example: A name of a person n consists of m terms $t_1 \dots t_m$. The person's first name fn and last name ln are comprised of disjunct subsets of the name terms: $t_1 \dots t_k$ and $t_{k+1} \dots t_l$, respectively. Assume that the remaining terms $t_{l+1} \dots t_m$ include the titles of the person. In addition, there is the address $a1$ that is comprised of tokens that do not appear in any of the previous strings.

$$\begin{aligned} n &= t_1 \dots t_m \\ fn &= t_1 \dots t_k \\ ln &= t_{k+1} \dots t_l \\ a1 &= t_{m+1} \dots t_o \end{aligned}$$

Recall from Sec. 5.2.1 that SoftTFIDF is used as the field similarity measure. The field similarity $fieldsim$ of two attribute values a and b is defined as:

$$fieldsim(a,b) = \sum_{t \in CLOSE(\theta,a,b)} w(a,t) \cdot w(b,t') \cdot termsim(t,t') \tag{7.1}$$

where t' is the term in b that is most similar to t according to the term similarity measure $termsim$, and $CLOSE(\theta, a, b)$ is a subset of all terms in a with

$$CLOSE(\theta,a,b) = \{t \in a | \exists t' \in b, termsim(t,t') > \theta\}. \tag{7.2}$$

The term similarity $termsim$ is based on the normalized edit distance.

We start with the similarity of n and fn. For each term in n, there is a matching term in fn, and thus, the term similarity is always 1. In the following, we also assume that the unnormalized TFIDF weight w' for a given term t is equal in all attributes. This is not a very restrictive assumption: The term frequency is usually small because of the nature of database attributes, and the distribution of terms should be similar across databases. We use $w'(t)$ to denote the unnormalized weight of term t. If these assumptions hold, the field similarity of n and fn is computed as:

$$fieldsim(n,fn) = \frac{\begin{pmatrix} w'(t_1) \\ \vdots \\ w'(t_k) \\ w'(t_{k+1}) \\ \vdots \\ w'(t_m) \end{pmatrix}}{\sqrt{w'^2(t_1) + \dots + w'^2(t_m)}} \cdot \frac{\begin{pmatrix} w'(t_1) \\ \vdots \\ w'(t_k) \\ 0 \\ \vdots \\ 0 \end{pmatrix}}{\sqrt{w'^2(t_1) + \dots + w'^2(t_k)}}$$

$$= \frac{w'^2(t_1) + \ldots + w'^2(t_k)}{\sqrt{w'^2(t_1) + \ldots + w'^2(t_m)}\sqrt{w'^2(t_1) + \ldots + w'^2(t_k)}}$$
$$= \frac{\sqrt{w'^2(t_1) + \ldots + w'^2(t_k)}}{\sqrt{w'^2(t_1) + \ldots + w'^2(t_m)}}$$

Adding tokens to fn reduces the individual, normalized weights to tokens $t_1 \ldots t_m$, because the token vector becomes longer. This reduces the similarity with n. However, if the new tokens also occur in n, their addition has an increasing effect. Assume that ln is added to fn, which results in the following field similarity:

$$fieldsim(n, fn \cup ln) = \frac{\begin{pmatrix} w'(t_1) \\ \vdots \\ w'(t_l) \\ w'(t_{l+1}) \\ \vdots \\ w'(t_m) \end{pmatrix}}{\sqrt{w'^2(t_1) + \ldots + w'^2(t_m)}} \cdot \frac{\begin{pmatrix} w'(t_1) \\ \vdots \\ w'(t_l) \\ 0 \\ \vdots \\ 0 \end{pmatrix}}{\sqrt{w'^2(t_1) + \ldots + w'^2(t_l)}}$$
$$= \frac{\sqrt{w'^2(t_1) + \ldots + w'^2(t_l)}}{\sqrt{w'^2(t_1) + \ldots + w'^2(t_m)}}$$

Obviously, the field similarity has increased, because the numerator is larger while the denominator is the same as before. In contrast, adding terms that do not appear in n (e.g., adding terms of $a1$) decreases the score, because there are no tokens that compensate the decreasing normalized weight of matching terms.

Unfortunately, real-world data is not as perfect as described above. Dirty data can cause a decrease in the field similarity even when related attributes are added. Assume that a person has several last names, and n contains only a single last name, while ln has all of them. Adding ln to fn can decrease the field similarity as described above: The normalized weight decreases, and the single new matching term cannot compensate. A similar effect occurs if the term similarity of matching tokens is not 1. Assume three strings:

$$s = t_1 t_2$$
$$s_1 = t_1$$
$$s_2 = t'_2$$

where $termsim(t_2, t'_2) = 0.4$. Given that the unnormalized TFIDF weights of all terms are equal, the field similarity of s and s_1 is $fieldsim(s, s_1) = \sqrt{0.5} = 0.707$. Adding s_2 to s_1 decreases the similarity with s independent of the threshold used: If $\theta \geq termsim(t_2, t'_2)$, then t'_2 is not considered a match for t_2, and thus, $fieldsim(s, s_1 \cup s_2) = 0.5$. If the term similarity passes the threshold, then the field similarity is $fieldsim(s, s_1 \cup s_2) = 0.5 + \sqrt{0.5} \cdot \sqrt{0.5} \cdot 0.4 = 0.7 < 0.707$.

Thus, the field similarity decreases although attributes are grouped correctly because t_2' is too dissimilar to t_2.

On the other hand, there is also the chance that adding unrelated attributes increases the similarity score. Prob. 4 in Sec. 4.3.1 states that unrelated attributes might have highly similar values, which can mislead duplicate detection. Such a case can also mislead the complex matching step, because an increase in the similarity score is considered an improvement. Both problems are considered in the experimental evaluation in Sec. 7.7.

7.4.3 Assessing Match Improvement

In contrast to the local criterion, which determines if a merge has improved the matrix from the view point of a single attribute group, the global criterion uses the whole matrix. It is not unusual that even though the local criterion is satisfied, the merging of attribute groups has lead to an unintentional matrix. If that happens, the global criterion in the function `childMatrices` must reject the child matrix.

The creation of child matrices is not the only part of the algorithm where matrices need to be assessed. The overall goal of the algorithm is to find the best matrix, which represents the correct matching between two tables. Hence, we need to define a function that assigns scores to the matrices created by the algorithm such that the best matrix has the highest score.

Discussion of possible functions

In both tasks, the score function should assess matrices based on the matching that they produce. There are two choices of **input** to the function:

1. *Graph matching*: The correspondences described by the maximum weight matching of the similarity matrix.

2. *Table matching*: The correspondences of the graph matching that pass the pruning threshold θ_{prune}.

Both choices have their advantages. The main difference lies in the *sensitivity*: The table matching does not always change even when attribute groups are correctly merged. To illustrate, assume a correspondence $(\{X_1, \ldots, X_4\}, \{Y\})$. Because the values of four source attributes need to be merged to create a value for Y, the similarity scores for each simple correspondence $(\{X_i\}, \{Y\})$ will be low. Merging X_1 and X_2 is a correct merge, because it leads to the above existing correspondence. However, because the attribute group $\{X_1, X_2\}$ consists of only two of the required four attributes, the similarity score is likely not to pass the threshold, and thus, the actual improvement cannot be seen in the table matching. In contrast, the improvement is easily observable when the graph matching correspondences are used.

The score function needs to combine the similarity score of the correspondences into a single match score. We considered two possible **functions**:

1. *Average*: The average of the correspondence scores.

2. *Sum*: The sum of the correspondence scores.

Using the average of correspondence scores has the advantage of always being in the range $[0, 1]$, while value range of *sum* depends on the number of correspondences, which in turn depends on the size of the tables: The number of correspondences before pruning is equal to the smaller number of attribute groups of the two tables. Hence, *sum* appears to be a bad choice for determining matrix improvement: If two attribute groups from the smaller side are merged, the sum of correspondence scores before pruning can decrease even when the merge is correct.

While the function *average* has the advantage that matrices are comparable even when the number of correspondences decreases, we have observed a tendency to 'overmerge' attribute groups: E.g., $\{(\{LN, FN, A1\}, \{N, A2\})\}$ would be considered an overmerge for the running example depicted in Fig. 7.1, because the expected two correspondences are merged into a single correspondence. In addition, correspondences can be missed, because the average score can increase when the number of correspondences decreases.

Assume the average of graph matching score to be defined as

$$avgScore(\mathcal{GM}) = \frac{\sum_{(A,B)\in\mathcal{GM}} avgSim(A, B)}{|\mathcal{GM}|} \tag{7.3}$$

where $\mathcal{GM}$ is a set of attribute group pairs (A, B) representing the maximum weight matching in the matrix M. The value of $avgScore$ increases if the sum of field similarity scores $avgSim$ increases or if the size of the matching $\mathcal{GM}$ decreases. The latter is an unintended effect that occurs with correspondences both before and after pruning: E.g., if only correspondences with a score above the pruning threshold are used, then the algorithm aims at removing correspondences that are just above the threshold.

	$\{A\}$	$\{BC\}$
$\{A\}$	**1**	0
$\{B\}$	0	**0.75**
$\{C\}$	0	0.63
$\{D\}$	0	0

(a) Start matrix.

	$\{A\}$	$\{BC\}$
$\{A\}$	**1**	0
$\{BC\}$	0	**0.9**
$\{D\}$	0	0

(b) Correct grouping.

	$\{A\}$	$\{BC\}$
$\{A\}$	**1**	0
$\{BD\}$	0	0.53
$\{C\}$	0	0.63

(c) Wrong grouping.

Figure 7.5: Increasing score average by overmerging.

Assume the start matrix shown in Fig. 7.5(a) and a pruning threshold of 0.7 (the correspondences above the threshold are highlighted in bold), which has an average score of 0.875. The average table matching score of the correct merging shown in Fig. 7.5(b) is 0.95, and thus, it would be considered a better matrix. However, the wrong grouping depicted in Fig. 7.5(c) has an even larger

score of 1, because merging B and D caused the average field similarity of the correspondence with BC to drop below the threshold.

In contrast, correspondences cannot be removed from consideration when the graph matching (i.e., the correspondences before pruning) is used. However, the number of correspondences is reduced when attribute groups of the smaller side are merged. We have observed cases where the nominator of Eq. 7.3 decreases in a merge step, but the *avgScore* value increased because the denominator decreased.

Function	Input	Advantages	Disadvantages
Sum	Graph Matching	Overmerge avoided	Major sensitivity to restructuring.
	Table Matching	Overmerge avoided	Minor sensitivity to restructuring.
Average	Graph Matching	Changes immediately reflected. Fixed value range.	Overmerging.
	Table Matching	Fixed value range.	Overmerging

Table 7.1: Comparison of scoring functions.

The discussion of possible scoring functions is summarized in Tab. 7.1. Several experiments with synthetic data sets confirmed that the two functions have antagonistic properties: While the *sum* might avoid (both correct and incorrect) merging of attributes, the average can lead to an overmerge.

Determining matrix improvement

The main goal of the score function *score* in the function `childMatrices` is to avoid matrices that do not lead to the correct matrix, and thus, reduce the overall number of matrices that have to be considered. Positive and negative changes should be immediately visible. As described above, the table matching correspondences only change when correspondences with a score above the threshold are affected by the merge. Hence, all correspondences of the graph matching are used.

Because the number of graph matching correspondences always decreases when attribute groups of the smaller side are merged, we chose to use the average of the correspondence scores as the *score* function:

$$score(M) = avgScore(\mathcal{GM}) = \frac{\sum_{(A,B)\in\mathcal{GM}} avgSim(A,B)}{|\mathcal{GM}|} \tag{7.4}$$

where $\mathcal{GM}$ is the maximum-weight graph matching of matrix M. The score is always in the range $[0,1]$, which facilitates the comparison of a parent matrix with its child matrices.

Choosing a best matrix

The matrices that pass the local and global criteria are candidates for the best matrix. Similar to the problem of determining matrix improvement, we assign scores to matrices and chose the matrix with the highest score as best matrix.

In contrast to the problem of determining matrix improvement, the average of graph matching scores has shown to be suboptimal for choosing the best matrix, because it can lead to overmerging. Hence, we define a new function *matchscore*. The input to this function is the correspondences that pass the pruning threshold, because they have shown to be a more reliable indicator. Each correspondence $(A, B) = (\{a_1, \ldots, a_i\}, \{b_1, \ldots, b_j\})$ whose field similarity is above the pruning threshold has a *correspondence score*, which is weighted by the number of attributes involved:

$$corscore(A, B) = (|A| + |B|) \cdot avgSim(A, B) \tag{7.5}$$

where $avgSim(A, B)$ is the average field similarity of attribute groups A and B. The match score of a matrix M is the sum of the correspondence score of its table matching $\mathcal{M}$, i.e., the correspondences that pass the pruning threshold:

$$matchscore(M) = matchscore(\mathcal{M}) = \sum_{(A,B)\in\mathcal{M}} corscore(A, B). \tag{7.6}$$

Recall from Tab. 7.1 that the sum does not have the problem of overmerging, but avoids attribute group merges if the number of correspondences decreases. Weighting attribute correspondences by the number of attributes involved tackles this problem, which is particularly important in the presence of m:n correspondences, which are discussed in the following section: Fig. 7.6 shows the start matrix and the matrix representing the correct matching in a m:n scenario (correspondences highlighted in bold). Note that the (unweighted) sum of field similarity scores is smaller in the second matrix, because the number of correspondences has decreased with respect to the first matrix. However, the *matchscore* has increased, because its field similarity 0.97 is larger than both 0.76 and 0.75, and it contains 6 attributes, while each of the two correspondences from which is was derived contain three attributes.

Note that the *matchscore* function does not work well in the previous problem of determining matrix improvement, because its value range differs depending on the matrix structure.

7.5 The Complex Matching Algorithm

The greedy approach described in Sec. 7.4.2 only works as desired in the presence of n:1-correspondences: If an m:n correspondence exists, a single merge will not improve the matrix, and thus, that branch of the search space will not be considered by the algorithm. This section describes the final complex matching algorithm, which is an adaptation of the algorithm described in Sec. 7.4.

The function *matchscore* successfully detects the best matrix even in the presence of m:n correspondences (Eq. 7.6). In contrast, the *score* of a matching does not always increase in such a scenario even when the merge is intuitively correct (Eq. 7.4).

	$\{A\}$	$\{BC\}$	$\{D\}$
$\{AB\}$	**0.76**	0.43	0
$\{C\}$	0.39	**0.75**	0
$\{D\}$	0	0	**0.93**

(a) Start matrix.

	$\{A, BC\}$	$\{D\}$
$\{AB, C\}$	**0.97**	0
$\{D\}$	0	**0.93**

(b) Correct grouping.

Figure 7.6: Start matrix and correct grouping in a m:n scenario.

Consider the matrices depicted in Fig. 7.6. Getting from the start matrix (Fig. 7.6(a)) to the correct matrix (Fig. 7.6(b)) requires to steps, which can be performed in any order: One can first merge source attribute groups $\{AB\}$ and $\{C\}$ and target attribute groups $\{A\}$ and $\{BC\}$ afterwards, or vice versa. In any case, the first merge step is likely to decrease the score of the matching.

Fig. 7.7 depicts the matrix after source attribute groups $\{AB\}$ and $\{C\}$ have been merged. As a result, the *score* of the matrix has dropped from 0.81 to 0.76. This indicates a false merge, although the merge step leads to the correct matrix. While the actual numbers in this example are fictitious, we have frequently observed such behavior in our experiments. One might argue that another *score* function should be used to catch such a scenario. However, we believe that even for a human user, who does not know the semantics of the attributes, the 'improvement' in the step from Fig. 7.6(a) to Fig. 7.7 is very hard to discover. This holds especially because false merges also lead to matrices that look like the matrix depicted in Fig. 7.7.

	$\{A\}$	$\{BC\}$	$\{D\}$
$\{ABC\}$	0.49	**0.59**	0
$\{D\}$	0	0	**0.93**

Figure 7.7: First step: Merging attribute groups $\{AB\}$ and $\{C\}$.

Adapting the algorithm

To be able to detect m:n correspondences, we chose to use function *score* as defined in Eq. 7.4 and to adapt the original algorithm (Sec. 7.4). Instead of immediately discarding matrices that do not improve their parent, we allow the algorithm to make another search step: Assume that the child matrix M_1 does not improve its parent M_0, but M_1's child M_2 has a higher *score* value than M_0. In that case we consider M_2 as an improvement. In the example scenario, the matrix in Fig. 7.7 would not be considered an improvement, but

since the (correct) matrix in Fig. 7.6(b) has a higher score than the start matrix in Fig. 7.6(a), it is further considered by the algorithm.

Function `childMatrices2`

Input: Average Similarity Matrix M
Output: Matrices derived from M by merging attribute groups

```
1  foreach attribute group X in M do
2      partners ← simAttGroups(M, X);
3      foreach P ∈ partners do
4          foreach attribute group Y ≠ P do
5              if avgSim(X, P ∪ Y) > avgSim(X, P) then
6                  M' = merge(M, P, Y);
7                  if isImprovement(M) then
8                      childMatrices = childMatrices ∪ M';
9                  else
10                     if improvesGrandParent(M') then
11                         childMatrices = childMatrices ∪ M';
12                     end
13                 end
14             end
15         end
16     end
17 end
18 return childMatrices
```

Function `childMatrices2` shows the new method for creating child matrices, which is mainly affected by the adaptation. Although not visible in the pseudo code, the first change with respect to the original function occurs in line 1, where variable X ranges over some attribute groups. In the original function, the variables ranges over all attribute groups in the matrix. This still holds if matrix M has been an improvement (i.e., if its score is larger than its parent's score). If that is not the case, then the function has been called with M because it is assumed that an m:n correspondence exists. Hence, we do not want to merge attribute groups of the table which has not been affected by the creation of M: E.g., if M has been created by merging source attribute groups, then only target attribute groups must be merged to create child matrices for M. Consequently, X must only range over source attribute groups of M.

Lines 7-12 show the new global criterion. The treatment of the new matrix M' depends on the status of its parent M: If M is an improvement over its parent (i.e., the grandparent of M'), then M' is added to the list of child matrices. Although not shown in the algorithm, M' is compared to M using the function *score* to determine if it is an improvement and labelled accordingly. If M has not been an improvement, then M' needs to be compared with its grandparent. That procedure is described below. If it is an improvement with respect to its

grandparent, then it is added to the list of child matrices.

Note that beside the function `childMatrices`, the only other part of the complex table matching algorithm that is affected by the adaptation is the definition of the function *best* in Alg. 4: Instead of considering all child matrices, the best matrix is chosen only among the matrices that are considered an improvement.

Comparing a matrix with its grandparent

The method `childMatrices2` ensures that if a matrix M' is to be compared with its grandparent M_{gp}, then both the source and target side of M' contain exactly one attribute group that is the result of merging two attribute groups of M_{gp}. In other words, M' has been created by first merging two source attribute groups and then two target attribute groups, or vice versa. This knowledge is exploited when determining if a matrix improves its grandparent: Instead of considering the whole matrices, we only look at the four fields in M_{gp} that represent the similarity of the source and target attribute groups that are merged to create M' and the single field in M' that is the result of merging those attribute groups. We consider a matrix an improvement over its grandparent if the field similarity of the merged attribute groups in M' is larger than each of the four field similarities in the grandparent's matrix M_{gp}.

	$\{A\}$	$\{BC\}$
$\{AB\}$	**0.76**	0.43
$\{C\}$	0.39	**0.75**

(a) Extract of start matrix.

	$\{A, BC\}$
$\{AB, C\}$	**0.97**

(b) Extract of correct grouping.

Figure 7.8: Extract of start matrix and correct grouping in Fig. 7.6.

Fig. 7.8 depicts the relevant extract of the example in Fig. 7.6. Assume that the matrix in Fig.7.6(b) needs to be compared with its grandparent in Fig.7.6(a). Obviously, it is the result of merging source attribute groups AB and C and merging target attribute groups A and BC. Fig.7.8(a) shows the matrix fields representing the field similarity of those attribute groups, while Fig. 7.8(b) depicts the field that is the result of the merge. The field similarity of the merged attribute groups (0.97) is larger than each of the field similarities of its constituent attribute groups (0.76, 0.43, 0.39, 0.75). Note that in practice we need to ensure that the difference between the matrix values is relevant, because minor improvements can spuriously occur and lead to overmerge. In the implementation, we used a small threshold of 0.1, i.e., the similarity score of the merged matrix must differ at least by that threshold from each of the four scores in the grandparent's matrix.

7.6 Matching and Mapping with Combination Functions

The DUMAS complex matching algorithm concatenates attribute values on the source and target side to create complex correspondences. While that procedure is sufficient to create a matching, it leaves two questions open: (i) What is the mapping derived from the matching, and (ii) does the process work with other functions, as well? This section discusses these questions and sketches out possible solution.

7.6.1 Query Discovery with Complex Correspondences

The matching algorithm combines attribute values using the concatenation function *concat* to create 1:n, n:1, and m:n correspondences. Note that attribute on both the source schema and the target schema can be combined. In contrast, the mapping derived from those correspondences needs to determine a value for each individual target attribute.

Simple correspondences and n:1 correspondences can be easily translated into a mapping, because source values can be used immediately or have to be processed by a combination function, respectively. The existence of 1:n or m:n correspondences poses a problem in the query discovery process, because it is only known that a combination of target attributes is related to one or several source attributes, respectively. However, it is unclear how individual target values can be computed. Assume that source attribute $Name$ is a concatenation of target attributes $GivenName$ and $Surname$. Transforming source tuples is a difficult task, because values of $Name$ have to be split into their given name and surname. This can only be done heuristically, because the concatenation function has no inverse. However, it is conceivable that known duplicates provide enough information to learn such heuristics in a supervised process.

7.6.2 Matching and Mapping with Different Functions

Although the complex matching algorithm has been designed and implemented to work with the concatenation function *concat* as the only combination function, in principle different functions can also be applied. The complex matching procedure is based of the matching step of Chap. 5: The similarity matrix is used as a starting point, and attributes are merged by combining their values in the duplicate tuple pairs. When functions other than *concat* are to be used, the only point of change is the *merge* procedure in Func. `childMatrices2`, which creates child matrices.

Assume that a source database describes real estate with attributes *Length* and $Width$, while the target database has a single attribute *Area*, which is the product of the former two attributes. Combining *Length* and $Width$ with simple string concatenation leads to a low field similarity with *Area*. However, if the matching system included an equation discovery system (e.g., LAGRAMGE [TD97]), it would be possible to find an equation that combines

Length and $Width$, namely the product of the two attributes, such that the resulting value is very close to the $Area$ value of the respective duplicate target tuple[3]. We propose an optimistic use of different possible field similarities: One should always use the better combination function to compute field similarities. In the above example, the matrix value for the correspondence $(\{Length, Width\}, \{Area\})$ would be $fieldsim(Length \cdot Width, Area)$ instead of $fieldsim(concat(Length, Width), Area)$.

7.7 Experimental Evaluation

As in Chap. 5, we have evaluated the complex table matching algorithm using both synthetic and real-world data. After defining quality measures for the complex matching problem, we present the experimental results.

7.7.1 Quality Measures for Complex Matching

While the schema matching literature agrees on using precision, recall, and the F-measure for assessing the quality of simple matchings, no such standard can be found for complex matchings. This is partly due to the fact that there is no large body of work on this topic, yet.

In contrast to simple matchings, a detected complex correspondence can be 'almost' correct: E.g., the example in Fig. 7.1 contains a complex correspondence $(\{FN, LN\}, \{N\})$. If the matching algorithm detected an incomplete correspondence $(\{LN\}, \{N\})$, this would not be considered a correct solution. However, it would give the user enough information to rectify the correspondence after it has been identified as being incomplete: Instead of searching through the whole source schema for corresponding attributes of N, he could look for the missing partner in the vicinity of LN, i.e., the same table or referenced tables. Hence, the detected correspondence could be considered partially correct.

To evaluate the result of the DUMAS complex matching algorithm, we define two versions of both recall and precision. Recall from Sec. 5.7 that these two measures are defined as

$$Precision = \frac{|C \cap R|}{|R|} \qquad Recall = \frac{|C \cap R|}{|C|} \tag{7.7}$$

where C is the correct result and R is the retrieved result. If precision and recall are applied on complex correspondences, we call those functions *C-Precision* and *C-Recall*. Those are very strict measure, because the correspondence $(\{LN\}, \{N\})$ would be considered incorrect.

To catch the above intuition that correspondences can be partially correct, we also define two functions *S-Precision* and *S-Recall*. They are calculated as described above, except that the constituent simple correspondences of each complex correspondence are considered: A complex correspondence

[3]Note that we need to find a single equation that holds for *all* duplicate tuple pairs.

$(\{a_1, \ldots, a_m\}, \{b_1, \ldots, b_n\})$ has $m \cdot n$ constituent correspondences $(\{a_i\}, \{b_j\})$ with $1 \leq i \leq m$ and $1 \leq j \leq n$. E.g., the correspondence $(\{FN, LN\}, \{N\})$ has two constituent simple correspondences $(\{FN\}, \{N\})$ and $(\{LN\}, \{N\})$. If only the latter was found as described above, we would have achieved an *S-Precision* of 1 and an *S-Recall* of 0.5.

It has to be noted that although the latter two functions reflect partial correctness, they penalize overmerging in the precision measure: If the complex matching result of Fig. 7.1 was $(\{FN, LN, A1\}, \{N, A2\})$, then it would be split into six constituent simple correspondences, resulting in a *S-Precision* of $\frac{3}{6} = 0.5$.

7.7.2 Real-world data

To determine how well the complex matching algorithm works on real-world data, we have performed experiments with data from the computational biology and the movie domain.

Computational biology

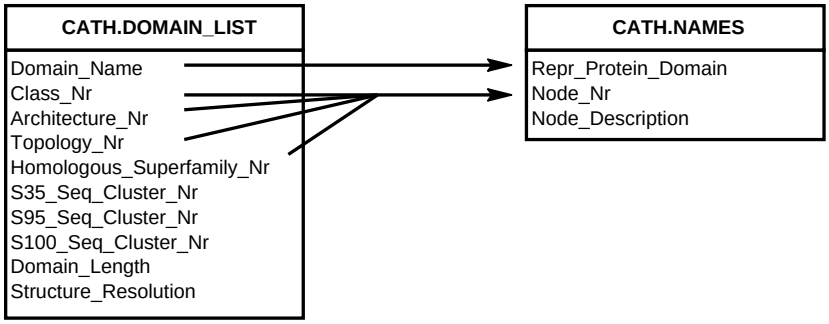


Figure 7.9: Tables containing information about proteins.

To evaluate the DUMAS complex matcher, we have used two tables containing information about proteins. Fig. 7.9 depicts the schemata of *DOMAIN_LIST* and *NAMES* containing 67,954 and 1,793 tuples, respectively. The arrows indicate the correspondences as defined by the domain expert: $(\{Topology\_Nr, Architecture\_Nr, Class\_Nr, Homologous\_Superfamily\_Nr\}, \{Node\_Domain\})$ and $(\{Domain\_Name\}, \{Repr\_Protein\_Domain\})$.

The DUMAS table matcher detected the following correspondences: $(\{Class\_Nr, Architecture\_Nr, Topology\_Nr, \}, \{Node\_Domain\})$ and $(\{Domain\_Name\}, \{Repr\_Protein\_Domain\})$, i.e., it did not include $Homologous\_Superfamily\_Nr$ in the matching. The reason for this can be found in the definition of $Node\_Nr$[4]: "Description of each node in the CATH hierarchy for all class, architecture and topology levels. CATH homologous

[4]ftp://ftp.biochem.ucl.ac.uk/pub/cathdata/v2.6.0/CathNames.v2.6.0

superfamily names are also included if they have been defined." Thus, the $Node\_Nr$ does not always contain a value for the homologous superfamily. In six of the ten detected duplicates, the $Node\_Nr$ value was comprised of of only three source values, resulting in a missed constituent correspondence.

Movie data

The movie data that was used in the experimental evaluation originated from the Internet Movie Database (IMDB)[5] and Filmdienst (FD)[6]. To assess the effectiveness of the algorithm in detecting complex correspondences, we used the single FD table that contains information about people. That table contained two attributes representing the given name and the last name of a person. In contrast, the IMDB schema contains several tables for actor, actress, director, etc. In each of those tables, the name of a person was represented as a single attribute.

The single FD table was compared with each of the IMDB tables representing people involved in the movie business. The result was perfect in all matching tasks, i.e., the DUMAS table matcher was able to determine that the name in the IMDB tables is a concatenation of the name attributes in the FD table.

7.7.3 Synthetic data

In addition to real-world data, we have performed extensive experiments on the generated data sets previously used to test the simple matcher. Fig. 7.10 shows the schema configuration that is the baseline for the experiments: one with six corresponding attributes and one with three corresponding attributes. We have used the same five database pairs for each configuration as in Sec. 4.6.2, each containing 5,000 tuples and 100 duplicates.

The baseline configuration depicted in Fig. 7.10 contains only simple correspondences. To create complex correspondences, attributes that have a corresponding partner were merged (i.e., their values were concatenated) and the matching algorithm was applied to the restructured databases.

Detecting 1:n correspondences

To assess the matching quality in the presence of 1:n correspondences, we have generated every possible 2:1 and 3:1 correspondence by merging source attributes. Because every possible attribute combination was used, the number of actual table matchings was quite large: In the case of six corresponding attributes, there are 15 groups of two source attributes and 20 groups of three attributes. Given that five different data sets exist for each configuration, the DUMAS complex matcher was applied 75 and 100 times to detect 2:1 and 3:1 correspondences for the baseline configuration of Fig. 7.10(a). Note that each

[5] Available from ftp://ftp.fu-berlin.de/pub/misc/movies/database/
[6] http://film-dienst.kim-info.de/

Attr. of *DB1*		Attr. of *DB2*
SSN	⟶	–
Profession	⟶	–
Surname	⟶	*Surname*
Name	⟶	*Name*
Birth-date	⟶	*Birth-date*
Birth-place	⟶	*Birth-place*
Birth-district	⟶	*Birth-district*
Sex	⟶	*Sex*
–	⟶	*City*
–	⟶	*District*

(a) Six corresponding attributes

Attr. of *DB1*		Attr. of *DB2*
SSN	⟶	–
Profession	⟶	–
Sex	⟶	–
Birth-district	⟶	–
Birth-place	⟶	–
Surname	⟶	*Surname*
Name	⟶	*Name*
Birth-date	⟶	*Birth-date*
–	⟶	*City*
–	⟶	*District*
–	⟶	*Street no*
–	⟶	*Address*
–	⟶	*Postal code*

(b) Three corresponding attributes

Figure 7.10: Schema configuration for complex matching experiments.

experiment only contained a single complex correspondence, the remaining correspondences were kept simple.

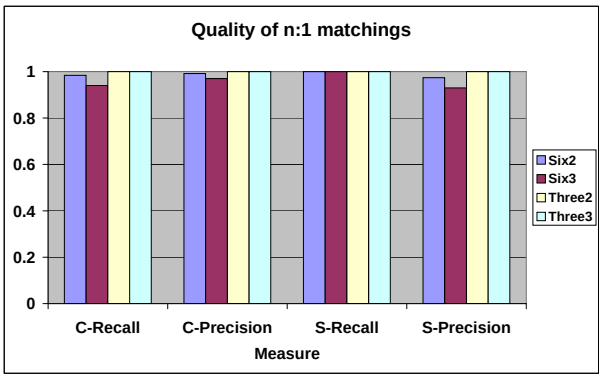


Figure 7.11: Experiments with 1:n correspondences.

Fig. 7.11 shows the experimental results for the baseline schemata with six correspondences and 2:1 correspondences (Six2), 3:1 correspondences (Six3), and for the baseline schemata with three correspondences and 2:1 correspondences (Three2), and a single 3:1 correspondence (Three3). One notices that Three2 and Three3 always produced perfect results.

The experiments with the databases derived from Fig. 7.10(a) produced errors, but the result was still satisfactory. An analysis of the experiments in Six2 showed that in only a single of the 75 matching tasks a false (i.e., non-

corresponding attribute) was added to the matching. The most prominent error was overmerging as described in Sec. 7.7.1, which resulted in a decrease in *S-Precision*. However, as can be seen in the perfect *S-Recall* all simple correspondences and the constituent correspondences were found. Of course, overmerging also caused *C-Precision* and *C-Recall* to decrease. We point out that the algorithm always produced a perfect result for four of the five data sets. The analysis of the Six3 experiments resulted in similar insight. Only a single data set produced false matchings, and in only a single case a non-corresponding attribute was matched.

Detecting m:n correspondences

The experiments for m:n correspondences were created in a similar fashion. In contrast to experiments with n:1 experiments, attributes in both the source and target had to be merged to create m:n scenarios. We restricted ourselves to merging two attributes in each schema, and required the source and target attribute group to have at least one (corresponding) attribute in common: If the source attributes a_i and a_j are merged, then the corresponding partner of either a_i or a_j must be merged with another target attribute that has a corresponding source attribute other than a_i or a_j. E.g., if *Surname* and *Name* in the source database of Fig. 7.10(a) were merged, then either *Surname* or *Name* of $DB2$ must be merged with another target attribute. Assuming that we merged *Name* and *Birth-place*, the correct matching would contain the correspondence ({*Name+Surname, Birth-place*},{*Name+Birth-place,Surname*}). Note that we performed experiments with every possible combination, resulting in 120 different configurations, and thus, 600 runs of the matching algorithm.

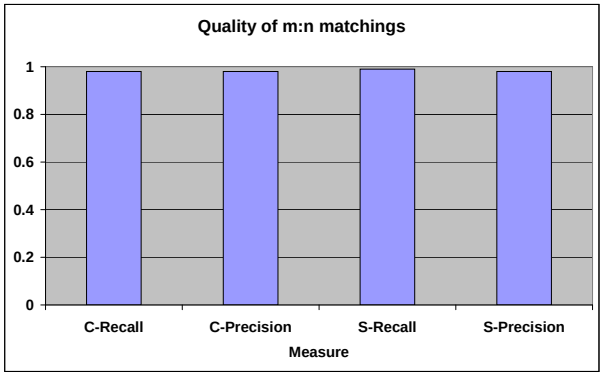


Figure 7.12: Experiments with m:n correspondences.

Fig. 7.12 shows the result of the experiments. With every quality measure being 0.98 or above, the algorithm can be considered successful. The analysis

of the matching result showed that, in addition to overmerging, some errors are the result of missed matches. Hence, the recall of simple correspondences is below 1. In only four of the 600 runs of the algorithm, a false attribute was added to the matching.

7.8 Discussion

While the majority of correspondences in matching tasks are 1:1, there are many scenarios where complex (i.e., n:1, 1:n, or m:n) correspondences exist. Detecting such correspondences is a very challenging task, because the number of possible complex matchings is very large when compared to the number of simple matchings. In fact, the majority of schema matching algorithms only produce simple correspondences.

The DUMAS complex matching described in this chapter is able to detect complex correspondences between two tables. Based on the similarity matrix constructed to detect simple correspondences, the algorithm merges attributes to improve the matching. The matcher uses various heuristics to reduce the number of matchings that have to be considered. We have experimentally shown that the algorithm detects complex matchings with high accuracy.

The general approach implemented in the DUMAS complex matcher is generic with respect to the combination function: Although the algorithm is presented and implemented with concatenation as the only function, we have also shown that other functions can be integrated into the matching system. However, we believe that this possibility has to be further investigated in various experiments. It has to be noted that the existence of correspondences involving complex data transformations also affects the quality of the duplicate detection result: As our duplicate detection algorithm is solely based on the *tupsim* measure, a complex relationship like $Area = Length \cdot Width$ might have a negative effect on duplicate detection accuracy.

Part IV

Discussion

Chapter 8

Conclusion

In this chapter we summarize our results and contributions and discuss future work on schema matching and data integration.

8.1 The DUMAS approach

The semi-automatic detection of attribute correspondences is a challenging task, because in most scenarios available information is not sufficient to infer the semantic relationships between schemata with 100% accuracy. Different 'hints' from data and metadata have to be exploited. Consequently, a variety of schema matching algorithms have been proposed. We described related work on schema matching and the drawbacks of the proposed solutions in Sec. 2.3. In the following, we argued that duplicates can help when schema-based and vertical instance-based matchers fail.

We developed several algorithms, which show that duplicates can be used for schema matching. The goal of *DUMAS table matcher* is the detection of simple correspondences between two tables. To do so, the matcher has to find duplicates in unaligned relations, which is a challenging problem that has not been discussed before. The experimental evaluation shows that our proposed tuple similarity measure is able to detect top-k duplicates with very high precision even in critical configurations. We also presented a procedure to extract attribute correspondences from the detected duplicates, which has shown to be accurate with both real-world and synthetic data.

The *DUMAS schema matcher* extends the duplicate-based approach to complex schemata. Here we face an additional problem: When comparing two arbitrary tables, it is not known if the tables are related and contain duplicates. We developed heuristics to interpret the result of the table matcher, and present an iterative matching algorithm that exploits multi-table duplicates. Finally, we developed the *DUMAS complex matcher* to extract complex correspondences from duplicates. The matcher produces various matrices by merging attributes and chooses a 'best' matrix that represents the complex matching.

With this thesis we have shown that duplicates can be used to detect attribute correspondences. The experimental evaluation indicates that the algorithms produce a highly accurate matching. However, we believe that there is still room for improvement. In the following, we discuss future research in duplicate-based matching and schema matching in general.

8.2 Combining DUMAS with other matchers

It is commonly agreed that no single indicator leads to the best matching: As discussed in Sec. 2.3, both schema- and instance-based matchers produce suboptimal results in certain cases (e.g., non-descriptive attribute labels or semantically different, but structurally similar attributes). The same holds for duplicate-based matchers: E.g., the duplicate detection procedure can produce false duplicates when few attributes correspond, and the value domains of some attributes overlap (e.g., place of residence and birthplace). Using such false duplicates in the matching step can degrade matching accuracy.

A possible way to improve matching quality is to combine the DUMAS matcher with other matchers. Sec. 2.3.5 discusses two design approaches: hybrid matchers and composite matchers. Hybrid matchers combine various matching approaches into a single algorithm. In their current state, the DUMAS algorithms already provide interfaces to interact with other matchers: E.g., the extended tuple similarity measure *etupsim* exploits known correspondences, which can be detected using any available matcher. The DUMAS schema matcher requires an initial matching, which could also be produced by another schema matcher.

Composite matchers execute several matchers independently and combine the matching results. In contrast to hybrid matchers, the constituent matchers do not benefit from the results of the other matchers. However, composite matchers are very flexible because new matchers can be easily added. An initial effort to integrate the DUMAS table matcher into COMA++ was discontinued due to the differences in the matching strategy: The COMA++ framework transforms a schema into a generic graph [ADMR05]. Descriptive features (e.g., attribute name, data type, statistics, child nodes) are extracted from each graph node in a preprocessing step. This process does not consider other databases, which might have to be matched with the current database in the future. The matchers that are used by COMA++ must only use the extracted features, i.e., the matchers work on the preprocessed data and not on the original database. Consequently, only schema-based and vertical instance-based matchers are applicable. Duplicate-based matchers need to access the databases for matching: It is impossible to detect duplicates in the preprocessing step, because different tuples are duplicates depending on which databases are to be matched.

8.3 Schema Matching and Data Integration

Schema matching is an important task in many integration scenarios. However, the result of the matching step, namely the attribute correspondences, are only the input to subsequent steps. Consequently, schema matching algorithms are unlikely to be distributed as separate tools. Instead, they are part of more complex data integration products.

Data integration has received significant attention mostly by the research community, but also by commercial manufacturers. While a few years ago only research prototypes (e.g., Clio [HHH+05]) were available, several commercial products can be found nowadays. Altova MapForce™2005 provides a visual interface to declare mappings between relational database schemata, XML schemata, electronic data interchange (EDI) interfaces, and flat files [Alt05]. The user only needs to draw correspondences between semantically related elements and specify data transformation functions. MapForce automatically generates the code in XSLT 1.0, XSLT 2.0, XQuery, Java, C++, or C#. The IBM Rational Data Architect, which has been heavily influenced by the Clio research project at IBM, goes one step further in the automatization of the mapping procedure [Gor05]. The tool exploits schema information as well as instance samples to perform schema matching, and thus, provides additional support for the specification of attribute correspondences.

The IBM Rational Data Architect is a step in the right direction, because it supports the user in the definition of attribute correspondences. However, a data integration should also help the user in the evaluation process, when the expert removes false correspondences and adds missing correspondences. Yan et al. have shown that sample data can aid the developer in the definition of mappings [YMHF01]. We believe that duplicates are particularly useful for the specification of correspondences: Instead of interpreting random tuples, the user can see how the same real-world entity is represented in both the source and the target. Duplicates are also very helpful for the evaluation of the detected correspondences: If the provided tuple pairs cannot be confirmed to be true duplicates, then the resulting matching is unlikely to be very accurate and must be changed by the expert. In addition, the user can easily specify complex data transformations based on the information provided by duplicates.

The schema mapping tools described above can be used to describe semantic relationships between schemata, i.e., models of data. The goal of *model management* is the provision of a generic framework to handle any kind of models, including those for which no instances exist (e.g., entity-relationship diagrams or UML class diagrams) [Ber03, MRB03]. Model management tools transform models into a generic format. Various operators are defined to manipulate those models, e.g., *Merge* to integrate models or *Diff* to determine the difference between models. One of the more complex operators is *Match*, which is used to detect correspondences between semantically related model elements. Although existing implementations of a generic match operator exploit only metadata, it is agreed that available data can improve the matching result [MGMR02].

Schema mapping and model management tools are restricted to the manip-

ulation of schemata. HumMer goes one step further by integrating the data as well: After the schemata are integrated based on the matching established by the DUMAS table matcher, the tool also detects duplicates and resolves inconsistencies on the instance level [BBB+05]. The result is a duplicate-free representation of the information contained in the source tables. We believe that tools like HumMer are particularly useful for ad-hoc or one-time integration tasks, e.g., catalogue integration, where schema mapping tools provide only limited support (see Sec. 1.5.1).

8.4 The Future of Schema Matching

Schema matching has been an active research area in the past years, and a variety of algorithms have been developed. While the experimental results are promising, new schema matching approaches that are currently not conceivable are likely to be proposed in the future. As has been shown in the past, schema matching research can greatly benefit from work in other areas, e.g., machine learning, text classification, or information theory. Many techniques that have emerged in those and other research areas are potentially applicable to detect attribute correspondences, and thus, contribute to the field of schema matching.

The vast majority of schema matching work has been done in academic research. Consequently, many experiments have used small data sets that have been extracted from Internet sources. In the future, these algorithms will have to prove themselves in commercial projects, including large-scale integration tasks. While the goal of schema matching research has been to improve accuracy, performance will be a major concern in industrial-strength schema matching. We emphasize that schema matching is an off-line process, and thus, performance is not as much an issue as in runtime systems. However, schema matching tools are used by human experts. These users might want to sacrifice a little matching accuracy for a large performance gain, e.g., because they can easily correct the matching based on their expert knowledge.

Exploiting expert knowledge is another potential research direction. Schema matching is always described as being a semi-automatic process. That notion is rarely reflected in the matching algorithms: Most schema matchers produce a schema matching without interacting with the user and assume the expert to correct the matching afterwards. The user can only influence the matching process by setting thresholds, providing sample data, or matching other sources. It is conceivable that schema matchers can benefit from an increased amount of user interaction both in terms of accuracy and performance, because the expert can 'guide' the matcher based on his domain knowledge.

Bibliography

[ACG02] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 586–597, 2002.

[ACMH03] Karl Aberer, Philippe Cudré-Mauroux, and Manfred Hauswirth. The Chatty Web: Emergent semantics through gossiping. In *Proceedings of the International World Wide Web Conference*, May 2003.

[ACMM03] Luigi Arlotta, Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Automatic annotation of data extracted from large web sites. In *Sixth Int. Workshop on the Web and Databases (WebDB 2003)*, 2003.

[ADMR05] David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 906–908, 2005.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Alt05] Altova. Data integration: Opportunities, challenges, and Altova MapForce^{TM} 2005. White paper available at http://www.altova.com/whitepapers/mapforce.pdf, 2005.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[BBB^{+}05] Alexander Bilke, Jens Bleiholder, Christoph Böhm, Karsten Draba, Felix Naumann, and Melanie Weis. Automatic data fusion with HumMer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1251–1254, 2005.

[BCV99] Sonia Bergamaschi, Silvana Castano, and Maurizio Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, March 1999.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, 2003.

[Bil04] Alexander Bilke. Instance-based schema management. In *Proceedings of the 11th Doctoral Consortium on Advanced Information Systems Engineering (CAiSE'04)*, pages 95–106, 2004.

[BKLW99] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated information systems: Concepts, terminiology and architectures. Forschungsberichte des Fachbereichs Informatik 99 – 9, Technische Universität Berlin, 1999.

[BKSS04] Catriel Beeri, Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Object fusion in geographic information systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 816–827, 2004.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BM02] Jacob Berlin and Amihai Motro. Database schema matching using machine learning with feature selection. In *Proceedings of the Conference in Advanced Information Systems Engineering (CAiSE)*, pages 452–466, 2002.

[BM03] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (KDD)*, pages 39–48, 2003.

[BMPQ04] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, and Christoph Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38–43, 2004.

[BN05] Alexander Bilke and Felix Naumann. Schema matching using duplicates. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 69–80, 2005.

[BSS03] Paola Bertolazzi, Luca De Santis, and Monica Scannapieco. Automatic record matching in cooperative information systems. In *Proceedings of the International Workshop on Data Quality in Cooperative Information Systsems (DQCIS)*, Siena, Italy, 2003.

[Bus02] Susanne Busse. *Modellkorrespondenzen für die kontinuierliche Entwicklung mediatorbasierter Informationssysteme*. Logos Verlag, 2002.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[CCL03] Cecil Eng H. Chua, Roger H. L. Chiang, and Ee-Peng Lim. Instance-based attribute identification in database integration. *VLDB Journal*, 12(3):228–243, 2003.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[CGGM03] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 313–324, 2003.

[CLR01] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 2001.

[CMM01] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. ROADRUNNER: Towards automatic data extraction from large web sites. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[Coh98] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1998.

[CRF03] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI Workshop on Information Integration on the Web (IIWeb)*, pages 73–78, 2003.

[DDH01] AnHai Doan, Pedro Domingos, and Alon Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 509–520, 2001.

[DDH03] AnHai Doan, Pedro Domingos, and Alon Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Machine Learning Journal*, 50:279–301, March 2003.

[DGMY03] Neil Daswani, Hector Garcia-Molina, and Beverly Yang. Open problems in data sharing peer-to-peer systems. In *Proceedings of the International Conference on Database Theory (ICDT)*, Lecture Notes in Computer Science 2572, pages 1–15. Springer-Verlag, 2003.

[DJMS02] Tamraparni Dasu, Theodore Johnson, S. Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 240–251, 2002.

[DLD+04] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Halevy, and Pedro Domingos. iMAP: Discovering complex semantic matches between database schemas. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 383–394, 2004.

[DLLH03] AnHai Doan, Ying Lu, Yoonkyong Lee, and Jiawei Han. Profile-based object matching for information integration. *IEEE Intelligent Systems*, 18(5):54–59, 2003.

[DR02] Hong-Hai Do and Erhard Rahm. COMA - a system for flexible combination of schema matching approaches. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 610–621, 2002.

[EVE02] Mohamed G. Elfeky, Vassilios S. Verykios, and Ahmed K. Elmagarmid. TAILOR: A record linkage toolbox. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 17–28, 2002.

[FPKT04] Ronald Fagin, Lucian Popa, Phokion G. Kolaitis, and Wang-Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 83–94, 2004.

[FS69] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[Gal86] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18(1):23–38, 1986.

[GE03] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[GGLZ04] Jarek Gryz, Junjie Guo, Linqi Liu, and Calisto Zuzarte. Query sampling in DB2 Universal Database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 839–843, 2004.

[GI89] Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.

[GIJ+01] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastasa. Approximate string joins in a database (almost) for free. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[GMPQ+97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[Gor05] Davor Gornik. Use Rational Data Architect to integrate data sources. Available at http://www-128.ibm.com/developerworks/library/ar-rdaint/, March 2005.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[HC03] Bin He and Kevin Chen-Chuan Chang. Statistical schema matching across web query interfaces. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 217–228, 2003.

[HCCH04] Bin He, Kevin Chen-Chuan, and Jiawei Han. Discovering complex matchings across web query interfaces: A correlation mining approach. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (KDD)*, pages 148–157, 2004.

[HHH+05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: From research prototype to industrial tool. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 805–810, 2005.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 321–332, 2003.

[HIST03] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*, March 2003.

[HMH01] Mauricio Hernaández, Renée Miller, and Laura M. Haas. Clio: A semi-automatic tool for schema mapping. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 607, 2001.

[HS95] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 127–138, 1995.

[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(1):9–37, 1998.

[HS03] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS)*, 28(4):517–580, 2003.

[KN03] Jaewoo Kang and Jeffrey F. Naughton. On schema matching with opaque column names and data values. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 205–216, 2003.

[Kol05] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2005.

[Kon06] Martin Konitzer. Effiziente Duplikaterkennung in heterogenen relationalen Datenbanken. Diploma thesis, Technische Universität Berlin, 2006.

[KS91] Won Kim and Jungyun Seo. Classifying schematic heterogeneity and data heterogeneity in multidatabase systems. *IEEE Computer*, 24(12):12–18, December 1991.

[LC94] Wen-Syan Li and Chris Clifton. Semantic integration in heterogeneous databases using neural networks. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1–12, 1994.

[LC00] Wen-Syan Li and Chris Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33(1):49–84, April 2000.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.

[LMK03] Kristina Lerman, Steven Minton, and Craig A. Knoblock. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research*, 18:149–181, 2003.

[LMR90] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LNE89] James A. Larson, Shamkant B. Navathe, and Ramez Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Trans. Software Eng.*, 15(4):449–463, 1989.

[LRNdST02] Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran Soares da Silva, and Juliana S. Teixeira. A brief survey of web data extraction tools. *SIGMOD Record*, 31(2):84–93, 2002.

[LRO96a] Alon Y. Levy, Anand Rajamaran, and Joann J. Ordille. Query-answering algorithms for information agents. In *Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 40–47, 1996.

[LRO96b] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 251–262, 1996.

[LSS01] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL — an extension to SQL for multidatabase interoperability. *ACM Transactions on Database Systems*, 26(4):476 – 519, December 2001.

[LT97] Daniel Lopresti and Andrew Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181:159–179, 1997.

[MAL+05] Robert McCann, Bedoor K. AlShebli, Quoc Le, Hoa Nguyen, Long Vu, and AnHai Doan. Mapping maintenance for data integration systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1018–1030, 2005.

[MBDH05] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Y. Halevy. Corpus-based schema matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 57–68, 2005.

[MBR01] Jayant Madhavan, Philipp A. Bernstein, and Erhard Rahm. Generic schema matching with Cupid. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 49–58, 2001.

[ME96] Alvaro E. Monge and Charles P. Elkan. The field matching problem: Algorithms and applications. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (KDD)*, pages 267–270, 1996.

[ME97] Alvaro E. Monge and Charles P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *2nd Workshop on Research Issues on Data Mining and Knowledge Discovery (DKMD'97)*, 1997.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 117–128, 2002.

[MH03] Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 572–583, 2003.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 77–88, 2000.

[MRB03] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A programming platform for generic model management. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 193–204, 2003.

[MZ98] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 122–133, 1998.

[NBYST01] Gonzalo Navarro, Ricardo Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):19–27, December 2001.

[NHT+02] Felix Naumann, Ching-Tien Ho, Xuqing Tian, Laura Haas, and Nimrod Megiddo. Attribute classification using feature analysis. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

[NKAJ59] H. B. Newcombe, J.M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[NL00] Mattis Neiling and Hans-Joachim Lenz. Data integration by means of object identification in information systems. In *Eigth European Conference on Information Systems*, 2000.

[NOTZ03] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 633–644, 2003.

[ÖV99] M. Tamer Özsu and Patrik Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1999.

[PAGM96] Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 413–424, 1996.

[PB03] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 826–873, 2003.

[PE95] Mike Perkowitz and Oren Etzioni. Category translation: Learning to understand information on the internet. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[PL00] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 484–495, 2000.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[PVM+02] Lucian Popa, Yannis Velegrakis, Renée Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 598–609, 2002.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 24(3):3–13, 2000.

[RDM04] Erhard Rahm, Hong Hai Do, and Sabine Massmann. Matching large XML schemas. *SIGMOD Record*, 33(4):26–31, 2004.

[RNHS06] Armin Roth, Felix Naumann, Tobias Hübner, and Martin Schweigert. System P: Query answering in PDMS under limited resources. In *IIWeb*, 2006.

[Rot64] Gian-Carlo Rota. The number of partitions of a set. *American Mathematical Monthly*, 71(5):498–504, 1964.

[Seb02] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SPD92] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1(1):81–126, 1992.

[ST96] Erkki Sutinen and Jorma Tarhio. Filtration with q-samples in approximate string matching. In *CPM*, pages 50–63, 1996.

[SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[TD97] Ljupco Todorovski and Saso Dzeroski. Declarative bias in equation discovery. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 376–384, 1997.

[TH04] Igor Tatarinov and Alon Halevy. Efficient query reformulation in peer data management systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 539–550, 2004.

[TKM02] Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (KDD)*, pages 350–359, 2002.

[Ukk92] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 19–40, 1997.

[WDM04] Wensheng Wu, Clement Yu AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 95–106, 2004.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[Win95] William E. Winkler. Matching and record linkage. In Brenda G. Cox et al., editors, *Business Survey Methods*. Wiley-Interscience, 1995.

[WN05] Melanie Weis and Felix Naumann. DogmatiX tracks down duplicates in XML. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 431–442, 2005.

[YMHF01] Ling-Ling Yan, Renée J. Miller, Laura M. Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 485 – 496, 2001.