Hasso-Plattner-Institut für Softwaresystemtechnik
Fachgebiet Informationssysteme

Data Fusion and Conflict Resolution in Integrated Information Systems

Dissertation
zur Erlangung des akademischen Grades
"doctor rerum naturalium"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin "Informationssysteme"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl.-Inform. Jens Bleiholder

Potsdam, 9. November 2010

Summary

The development of the Internet in recent years has made it possible and useful to access many different information systems anywhere in the world to obtain information. While there is much research on the integration of heterogeneous information systems, most stops short of the actual integration of available data.

Data fusion is the process of combining multiple records representing the same real-world object into a single, consistent, and clean representation. The problem that is considered when fusing data is the problem of handling data conflicts (uncertainties and contradictions) that may exist among the different representations. In preliminary steps, duplicate detection techniques help in identifying different representations of same real-world objects and schema mappings are used to identify common representations if data originates in different sources.

This thesis first formalizes *data fusion* as part of a data integration process and describes and categorizes different conflict handling strategies when fusing data. Based on this categorization, three different database operators are presented in more detail: First, the *minimum union* operator and the *complement union* operator, which both are able to handle different special cases of uncertainties and implement one specific strategy. They condense the set of multiple representations by removing unnecessary NULL values. In case of *minimum union*, subsumed tuples are removed as a subsumed tuple does not contain any new information. In case of *complement union*, two tuples are combined into one, if the attribute values complement each other, i.e., if for each attribute the values coincide or there is a value in only one of the tuples. The result of both operators contains fewer NULL values. We show how the operators can be moved around in query plans. Second, we present the *data fusion* operator, which is able to implement many different conflict handling strategies and can be used to really resolve conflicts, resulting in one single representation for each real-world object. We also show how the operator can be moved around in query plans. For all three operators we give algorithms and show how to implement them as primitives in a database management system.

The feasibility and usefulness of the operators is shown by including and evaluating them in two different integrated information systems developed within our group: the HumMer system is an integrated information system that allows the user to integrate and fuse data from different data sources using a wizard-like interface. The FuSem system allows the user to apply different fusion semantics and their corresponding operators to its data, visualize the differences of the fused results and finally come up with the most satisfying result for the task at hand.

The thesis covers the process of *data fusion* in its entirety: the user chooses one or more strategies to fuse data and expresses the strategies using a relational integrated information system. Then, the system executes the operation in an efficient way and finally lets the user look at the result and explore differences between different fusion techniques/strategies.

to my parents, my sister, and aowcnlwtsmft

Acknowledgement

This PhD thesis would not have been possible without the help and support of many people whom I would like to thank.

First, I would like to thank my advisor Felix for the friendly and inspiring atmosphere in his research groups at HU Berlin and HPI Potsdam and for the opportunity to participate in the establishment of both groups. He, as advisor, and all other PhD students who started with me at HU Berlin around the same time (Jörg, Melanie, and Silke) and all other colleagues at HU Berlin and HPI Potsdam provided through lunch or coffee talks, fruitful discussions and helpful suggestions an awesome environment; without it, and them, successful and fun work would not have been possible. I would also like to thank Sascha and Frank for their help and research cooperation in the minimum union project and Ralf and Sven for the sidestep into another field.

A special thanks to Louiqa, who gave me early on (in my first year) the opportunity to spent some time abroad and to get to know a completely different area (life sciences), its research challenges and problems.

Thanks also to all the students that participated to different degrees in this research, and particularly helped with implementing the two systems, HumMer and FuSem. First of all Karsten, who has been deeply involved into the coding of HumMer and FuSem. Thanks to Peter and Daniel for help with the creation of the artificial datasets and all other students that helped with their theses in the HumMer/FuSem environment.

Thanks to Marcelo and the DAAD for giving me the opportunity to spend three months at Pontificia Universidad Católica de Chile in Santiago de Chile (practically at the other end of the world) and the resulting different views, not only towards my research.

Last, thanks to all friends who sweetened my life besides research, who have been appreciative of the ups and downs of a PhD student, who accompanied me during the time, and who have been supportive if needed. Thanks to my family for their support throughout the years; without them all this would not have been possible.

I gratefully acknowledge funding of this research: this research was mostly funded by the German Research Society (DFG grant no. NA 432), the research period in Chile was founded by a DAAD Kurzstipendium für Doktoranden.

Table Of Contents

Part I

Introducing Data Fusion

Begin at the beginning and go on till you come to the end: then stop.

(Lewis Carroll)

1

Introduction

With more and more information sources available via cheap network connections, either over the Internet or in company intranets, the desire to access all these sources through a consistent interface has been the driving force behind much research in the field of information integration. During the last three decades many systems that try to accomplish this goal have been developed, with varying degrees of success.

One of the advantages of information integration systems is that the user of such a system obtains a complete yet concise overview of all existing data without needing to access all data sources separately. Complete because no object is forgotten in the result; concise because no object is represented twice and the data presented to the user is without contradiction. The latter is usually the most difficult to achieve, because information about entities is usually stored in more than one source.

After the major technical problems of connecting different data sources on different machines have been solved, the biggest challenge remains: overcoming semantic heterogeneity, i.e., overcoming the effect of same information being stored in different ways. Two main problems are the detection of equivalent schema elements in different sources (*schema matching*) and the detection of equivalent object descriptions (*duplicate detection*) in different sources to integrate data into one single and consistent representation.

A third problem, the problem of dealing with contradictory attribute values when integrating data from different sources (*data fusion*) is first mentioned by (Dayal, 1983). Since then, in the field of information integration, the problem has often been ignored, yet a few approaches and techniques have emerged. Many of them try to "avoid" the data conflicts by handling only the uncertainty of missing values, but quite a few of them use different kinds of resolution techniques to "resolve" conflicts. This thesis takes a closer look at the data fusion step in the data integration process, a closer look at the particular problem of handling data conflicts in data integration. We start in the next section by developing an intuition of where and how data fusion can take place.

1.1 Data Fusion Application Areas

Data fusion is part of the data integration process. Therefore it can take place whenever different representations of the same real-world entity (*fuzzy duplicates*) are integrated and need to be combined. Such representations can be combined in one source alone, or can be combined across different sources. Also, there are a lot of real-world examples that can be discovered day by day, just by surfing the web. However, not all data fusion techniques are applicable equally well in all situations.

Data Fusion Scenarios In the following we describe examples and scenarios where data fusion techniques are helpful.

Company Mergers: An intuitive example where data fusion techniques are needed because of different representations of the same real-world entity coming from different sources are company mergers. When two companies merge (e.g., when recently the company *Bayer AG* bought *Schering AG*), a lot of data exists twice in the new – merged – company. As both above companies have been doing business in the same area (health care sector), this is especially true for their Customer Relationship Management (CRM) data, the data

collection on their customers (e.g., pharmacies, sales representatives, etc.). Both companies have two distinct but overlapping customer databases that need to be fused into one. This is generally done in two steps, by first detecting fuzzy customer duplicates (different representations of same real-world customers), and then using data fusion techniques to handle possible conflicts among the representations and consolidate them into one single representation. Also, much other information is duplicated in other divisions of the companies: data on suppliers, skill management data, documentation on products, etc. There is a need to integrate the data and, in case of conflicts, to handle them.

Catalog Integration: Another example scenario where fuzzy duplicates exists and data fusion techniques are helpful is the integration of product catalogs. This can be done as part of a company merger, but also when integrating product catalogs from different suppliers into ones own product catalog (e.g., resellers, retailers, online shops, etc.) or into ones own order database. Yet another example for catalog integration are websites such as IDEALO[1] that integrate product data from different shopping websites and allow users to compare prices of products. Whereas the first scenario involves the actual reproduction of the product data within ones own company (materialized approached), the example of price comparison may require the real-time or near real-time integration and fusion of products (virtual approach) in order to guarantee the most recent prices. Such a virtual approach is also termed *on-the-fly* data integration. An example for a pure virtual approach is the merging of search engine results in meta search engines such as METACRAWLER[2]. Given a query, a meta search engines queries several other search engines, finds duplicates among the results and fuses the entries into one representation in the combined search result. Particularly interesting in this context is the combination of the rankings.

Disaster Management: Data integration has started being used in disaster management scenarios after the 2004 Indian Ocean earthquake. In the disaster management scenario, different highly heterogeneous sources need to be connected to each other in order to allow for integration of all available data. An integrated view on all available data then can help organize police, firefighters, and relief organizations. Data integration in this domain also allows for missing people to be found when information of peoples whereabouts is integrated from different sources such as the police (information on missing people) and hospitals (information of found people). The SAHANA system (Careem et al., 2006) has been developed after the Indian Ocean earthquake and is the first documented data integration system especially suited for this scenario.

Data Cleansing: So far we have seen scenarios where data is integrated from two or more sources. Another broad application of data fusion techniques are typical data cleansing scenarios where one single source or table is cleansed, e.g., by detecting and removing different representations of same real-world entities (fuzzy duplicates). Fuzzy duplicates in single sources may exist due to errors when entering data (misspellings), or changes in the data over time (outdated data, people changing names, etc.). A good example is again the customer relationship domain. Entries in a customer database change frequently over time and therefore periodically should be cleansed. After fuzzy duplicates have been detected in a customer database, one needs to decide which data values to keep and which values to discard to define the combined representation. Here, data fusion techniques are helpful.

Real-world Examples Throughout our experience with real-world datasets, we have seen the need for data fusion operators, as most real-world datasets contain fuzzy duplicates. Concerning fuzzy duplicates involving missing values we can very easily find numerous examples on the web. For instance, as of January, 9th 2009, when searching for "Carla Bruni rien" on Amazon[3], we obtain five results for the 2008 Import CD "Comme si de Rien n'Etait" by Carla Bruni. Two sample representations of these five results are represented in Figure 1.1. Another example of fuzzy duplicates at the Amazon database is given in Figure 1.2. Both examples contain two representations of the same CD that complement each other, i.e., they contain overlapping information (e.g., artist and title) as well as information that is present in one, but not present in the other representation (e.g., label or price information). Please note that the CD representations are identified by their ASIN number, an artificial identifier Amazon assigns to its products. The data fusion techniques that we describe in this thesis

[1] IDEALO: `http://www.idealo.de` as seen on April, 25th 2010

[2] METACRAWLER: `http://www.metacrawler.de` as seen on April, 25th 2010

[3] `http://www.amazon.com`

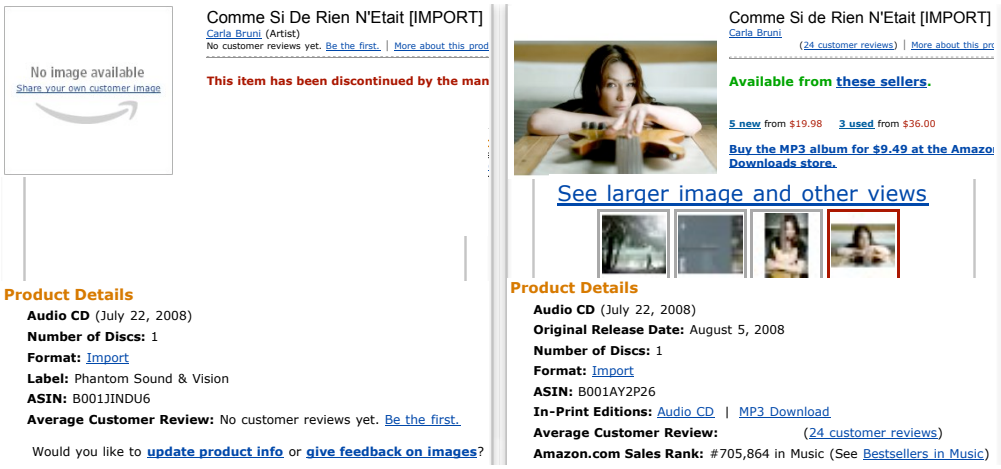


Figure 1.1: Amazon CD representations when searching for "Carla Bruni rien" on Amazon.com as of January, 9th 2009.

Figure 1.2: Complementing entries of the same audio CD at Amazon.com, as of October, 2nd 2009.

help to clean such a product database containing different representations of the same entity.

Different Techniques for Different Scenarios Due to the inherent differences of the scenarios presented so far (e.g., different characteristics of data, different tasks to fulfill) a whole set of different data fusion techniques are needed. Different scenarios require different operators. For example, using a specific data fusion technique (such as the *complementation* operator presented in this thesis) we may combine information that was not already combined in one of the sources. That way, we obtain a more complete description of an object, albeit at the risk of creating inaccurate combinations. Such a behavior is acceptable in scenarios where occasional incorrect combinations have little negative impact on the application and the combined information is valuable, e.g., in a disaster management scenario. Hence, a different technique (such as the *subsumption* operator presented in this thesis) can be used in applications where having correct (combined) information is crucial. Such a sample scenario is the integration of a customer database with a bank account database, where assigning an incorrect bank account to a person is not desirable. The next section of the introduction explores some parts of the solution space of data fusion results on the basis of a specific example to give an intuition of how data fusion can be realized.

1.2 Data Fusion Results

In this section, we use an example from the domain of disaster management to introduce several possible results of data fusion and give the reader an idea of the wide variety of different techniques and their advantages and disadvantages. Disaster management deals with the integration of several, heterogeneous and autonomous data sources in case of a disaster. A possible task in this scenario is to find people by integrating information from hospitals (people found) and the police (missing people) In this example, two overlapping relations,

representing people, are combined. We will not only consider standard relational operators, but also several more powerful ways of combining two relations.

For additional and more detailed information on state-of-the-art data fusion techniques, how they work in detail, and on data integration systems applying them, we refer the interested reader to (Bleiholder and Naumann, 2008) and Chapter 9 for a recent and more detailed survey on single data fusion techniques and systems. The examples in this chapter serve merely as a motivation for regarding the techniques described in more detail throughout this thesis and to give the reader a quick overview of the different possibilities when integrating data.

Regarding the following two tables (Table 1.1 and 1.2, which represent different information about people), the Police table contains information about missing persons and the Hospital table contains information about persons admitted to a hospital. If we want to find out if a missed person has already been found or to inform relatives of persons admitted to the hospital at their home address, we need to integrate the information stored in both of the two tables. In this scenario, our goal is then to integrate the two tables into one single table, as *concise* as possible (no duplicates, no unnecessary information), while at the same time being as *complete* as possible (all available data, not losing any important information). Such a result then gives a good overview of persons' whereabouts. For future reference we added an artificial integer label to each tuple in the tables. When looking closely at the example we may conclude that we are dealing with four different people, each with different representations. So the ideal result would consist of four distinct and consistent tuples, one tuple per person. We will see that there are in fact a variety of possible results when integrating the two tables using existing techniques from the literature.

Relation *Police*

	Name	Birthdate	Sex	Address
1	Miller	7/7/1959	m	234 Main St.
2	Miller	⊥	⊥	234 Main St.
3	Peters	1/19/1953	m	43 First St.
8	Smith	8/9/1970	m	Mass Ave.

Table 1.1: Information about missing persons as given by the police.

Relation *Hospital*

	Name	Birthdate	Sex	Blood
4	Peters	1/19/1953	⊥	AB
5	Peters	1/19/1953	m	⊥
6	Miller	⊥	f	B
7	Miller	7/7/1959	m	o
9	Smith	9/8/1970	m	A

Table 1.2: Information about persons admitted to a hospital.

When combining data of both tables into one single table, a very concise result is shown in Table 1.3. Here, each person (distinguishing people only by its Name attribute) is only contained once, with one value for each attribute. The values are chosen from the set of available values (in the sources), using conflict resolution functions. In case of the *Miller* tuples we decided to take "*m*" as the correct value for the attribute Sex and took the corresponding value for the Blood attribute. The conflict in the attribute Birthday has been resolved by choosing the value "*8/9/1970*". A result as shown in Table 1.3 is concise and complete, with all contradictions removed. Data fusion aims at producing complete and concise results like that.

In the example result of Table 1.3 we integrated the two tables as they are, without considering duplicate detection techniques used in advance. In particular, we only used the Name attribute to identify real-world identities, people in our example. This does not lead to the optimal and desired result, as all *Miller* tuples are combined into one result tuple although there seem to be two persons: Mrs. Miller (see e.g., tuple 6) and Mr. Miller (see e.g., tuple 1). Using the combination of attributes Name and Sex to identify real-world identities, would lead to a better result, however, we could run into problems when considering tuples with

Data Fusion of *Police* **and** *Hospital*

	Name	Birthdate	Sex	Address	Blood
	Miller	7/7/1959	m	234 Main St.	o
	Peters	1/19/1953	m	43 First St.	AB
	Smith	8/9/1970	m	Mass Ave.	A

Table 1.3: Combining Table 1.1 and Table 1.2 using *outer union* and conflict resolution functions.

a NULL value in either NAME or SEX such as tuple 2. A standard duplicate detection techniques that is quite perfect and assigns a unique ID in a separate column for each distinct real-world person could replace this way of identification. Instead of NAME and SEX this artificial ID is used to distinguish people. Although the examples would look slightly different when using such an artificially introduced ID, the basic problems when combining complementing tuples, removing subsumed tuples, and taking care of contradicting data would still be the same. To keep things simple, in the following examples we use attributes NAME and SEX to identify people.

A possible result that is very easily produced is shown in Table 1.4. Looking at the result we see that the schemata are aligned in such a way that corresponding attributes appear only once in the result. Also, all tuples from the sources are included in the final result. Although the result is complete, it is not concise, as there is still more than one representation for one person.

Outer Union of *Police* **and** *Hospital*

	Name	Birthdate	Sex	Address	Blood
1	Miller	7/7/1959	m	234 Main St.	⊥
2	Miller	⊥	⊥	234 Main St.	⊥
3	Peters	1/19/1953	m	43 First St.	⊥
4	Peters	1/19/1953	⊥	⊥	AB
5	Peters	1/19/1953	m	⊥	⊥
6	Miller	⊥	f	⊥	B
7	Miller	7/7/1959	m	⊥	o
8	Smith	8/9/1970	m	Mass Ave.	⊥
9	Smith	9/8/1970	m	⊥	A

Table 1.4: Outer Union result of the two example tables.

Table 1.5 shows another possible result: As you can see from the table, only three out of the four people from the sources are represented here, some of them still with contradicting data (as in the *Smith* case). Also, the schema contains the same information twice (e.g., attribute BIRTHDAY).

Join of *Police* **and** *Hospital*

	Name	Sex	P.Birthdate	H.Birthday	P.Address	H.Blood
$1 \bowtie 7$	Miller	m	7/7/1959	7/7/1959	234 Main St.	o
$3 \bowtie 5$	Peters	m	1/19/1953	1/19/1953	43 First St.	⊥
$8 \bowtie 9$	Smith	m	8/9/1970	9/8/1970	mass Ave.	A

Table 1.5: Join result of the two example tables, using NAME and SEX as join attributes.

The standard relational algebra already offers some operators that can be used to combine the information of different object representations stored in separate tables. In case of the two results above, these are the *outer union* operator and the *join* (using an equality condition on NAME and SEX). However, their result is not perfectly desirable, as they do not handle uncertainties and conflicts very well. Using other join attributes or an *outer join* variant eventually includes the fourth person in the second example (Table 1.5). This increases completeness, but may come at the cost of additional tuples/representations (decreasing conciseness) and still does not resolve the contradiction in the *Smith* representation. Eliminating these contradictions then needs to be done in an additional step. Also, the handling of exact duplicates (none in the examples), or the removal of subsumed tuples (i.e., tuples that contain information already contained in other tuples e.g., tuple 2

in Table 1.4) further enhances the result.

Another very different possibility to combine the two tables is depicted in Table 1.6. Looking at the result we see that it contains only tuples that are unique in its values for NAME and SEX. Combinations of NAME and SEX that exist multiple times in the sources (e.g., the *Smith* tuples 8 and 9) are not included. Such a result is concise and consistent as it does not contain any contradictions, but also incomplete as much information from the sources is missing in the result. This result has been created following the "Consistent Query Answering" (CQA) approach (Fuxman et al., 2005a). This technique assumes an artificially introduced unique constraint, i.e., on NAME and SEX, and returns only tuples that do not violate that constraint. In our example the constraint has been applied to the *outer union* of the two example tables, although also a *join* can be used to combine the tables beforehand.

Consistent Query Answer of *Police* **and** *Hospital*

	Name	Birthdate	Sex	Address	Blood
2	Miller	⊥	⊥	234 Main St.	⊥
4	Peters	1/19/1953	⊥	⊥	AB
6	Miller	⊥	f	⊥	B

Table 1.6: Consistent answer on the Outer Union of the two tables from above (see also Section 9.1.3).

In addition to these examples, there exists a variety of other possibilities to combine the two tables, some of which are depicted in the following Tables 1.7 to 1.11. They differ in the schema of the result as well as in the tuples that are included and how its values are chosen. We refer the reader to (Bleiholder and Naumann, 2008) and Chapter 9 for a more detailed description of how such results are generated from the sources.

From all the possible results, *minimum union* (see Table 1.10) and *full disjunction* (see Table 1.9) are nearest to an optimal result of four tuples, one tuple per person. Both remove subsumed tuples, i.e., tuples that contain information already contained in other tuples (see e.g., tuples 1 and 2). Looking more closely at these two results, we notice that the case of complementing tuples (e.g., tuples 1 and 7; tuples where attribute values complement each other) is not handled in both approaches. Defining and applying a *complement union* operation – similarly to the *minimum union* – results in a result as shown in Table 1.11.

Minimum union and *complement union* thus tackle two different aspects of uncertainties, the aspect of additional and complementing (overlapping but not contradicting) data. Both operations used in combination allow to solve most of the problems caused by NULL values when integrating data from different sources. The case of real contradictions, as can be observed in the representations for *Smith*, requires different techniques. It could be seen from two different perspectives: combining contradictions vertically (contradictions between tuples, as e.g., in most of the union approaches) and combining contradictions horizontally (contradictions between attributes, as e.g., in the join approaches).

In the following chapters we will be concerned with the two aspects of removing uncertainties (*minimum union* and *complement union*) and resolving contradictions (*data fusion*) and finally how to come up with a concise and complete integrated result.

Match Join of *Police* **and** *Hospital*

	Name	Birthdate	Sex	Adress	Blood
	Miller	7/7/1959	m	234 Main St.	B
	Miller	7/7/1959	m	234 Main St.	o
	Miller	7/7/1959	f	234 Main St.	B
	Miller	7/7/1959	f	234 Main St.	o
	Peters	1/19/1953	m	43 First St.	AB
	Smiths	8/9/1970	m	Mass Ave.	AB
	Smith	9/8/1970	m	Mass Ave.	AB

Table 1.7: Match Join of the two example tables, using NAME as identifier (see also Section 9.1.1).

Merge on *Police* and *Hospital*

	Name	Sex	P.Birthdate	H.Birthday	Address	Blood
	Peters	⊥	⊥	1/19/1953	⊥	AB
	Peters	m	1/19/1953	1/19/1953	43 First St.	⊥
	Miller	f	⊥	⊥	⊥	B
	Miller	m	7/7/1959	7/7/1959	234 Main St.	o
	Miller	⊥	⊥	⊥	234 Main St.	⊥
	Smith	m	8/9/1970	9/8/1970	Mass Ave.	A

Table 1.8: Merge on the two tables from above, using Name and Sex as identifying attributes (see also Section 9.1.2).

Full Disjunction of *Police* and *Hospital*

	Name	Sex	P.Birthdate	H.Birthdate	Address	Blood
4	Peters	⊥	⊥	1/19/1953	⊥	AB
6	Miller	f	⊥	⊥	⊥	B
1⋈ 7	Miller	m	7/7/1959	7/7/1959	234 Main St.	o
3⋈ 5	Peters	m	1/19/1953	1/19/1953	43 First St.	⊥
8⋈ 9	Smith	m	8/9/1970	9/8/1970	Mass Ave.	A

Table 1.9: Full Disjunction of Police and Hospital (see also Section 9.1.1).

Minimum Union of *Police* and *Hospital*

	Name	Birthdate	Sex	Address	Blood
1+2	Miller	7/7/1959	m	234 Main St.	⊥
3+5	Peters	1/19/1953	m	43 First St.	⊥
4	Peters	1/19/1953	⊥	⊥	AB
6	Miller	⊥	f	⊥	B
7	Miller	7/7/1959	m	⊥	o
8	Smith	8/9/1970	m	Mass Ave.	⊥
9	Smith	9/8/1970	m	⊥	A

Table 1.10: Minimum Union of the two tables Police and Hospital, handling cases of subsuming tuples.

Complement Union of *Police* and *Hospital*

	Name	Birthdate	Sex	Address	Blood
1+7	Miller	7/7/1959	m	234 Main St.	o
2+6	Miller	⊥	f	234 Main St.	B
3+4	Peters	1/19/1953	m	43 First St.	AB
4+5	Peters	1/19/1953	m	⊥	AB
2+7	Miller	⊥	m	234 Main St.	o
8	Smith	8/9/1970	m	Mass Ave.	⊥
9	Smith	9/8/1970	m	⊥	A

Table 1.11: Complement Union of the two example tables, handling cases of complementing tuples.

1.3 Contributions and Outline

In this thesis we look at the problem of actually integrating data from different data sources in the context of relational information integration processes. We hereby look closely at how different, possibly conflicting representations of the same real-world object are handled and integrated into a final representation. We are especially concerned about efficient implementations of operators that handle missing and conflicting data and their inclusion in an relational data model and integration system. The main contributions of this work are:

Classification of data fusion strategies and survey of existing techniques and systems
With this thesis we give the first extensive survey of the field of data fusion techniques and information integration systems developed so far and describe and compare their abilities and limitations. In addition we present a classification of conflict handling strategies (Bleiholder and Naumann, 2006a,b) which can be implemented by a data fusion technique or within an information integration system. The results of this survey have been published in (Bleiholder and Naumann, 2008) and are summarized in Chapter 9; the conflict handling strategies are presented as part of Chapter 3. We furthermore show how the different strategies can be implemented as part of an information integration system.

Formalization of data fusion
In this thesis we embed *data fusion* as a third main step in the more general information integration process after *schema matching* and *duplicate detection*. We define a versatile data fusion operator that can handle conflicting information and can be used to implement most of the data fusion strategies given. We show how such a *data fusion* operator can be embedded into the relational world by giving an extension to SQL to express data fusion queries (Bleiholder and Naumann, 2005, 2004; Bleiholder, 2005) and transformation rules that can be used by a relational optimizer to move the operator around in query trees. In addition, we describe two special case operators for handling missing information: *subsumption* has previously been defined and *complementation* is being been introduced in this thesis. These contributions can be found in Chapter 2 and Chapter 4. The transformation rules for data fusion are part of Chapter 6

Implementation and query optimization for subsumption and minimum union
Minimum union is a relational operator where subsumed tuples are removed from the result of an *outer union*. The *subsumption*/*minimum union* operation is one possible way of implementing a specific data fusion strategy. It is already known in the field of information integration and has also been used as part of the *full disjunction* operator or in outer join query optimization, and in the context of the CLIO schema matching system. However, an efficient implementation of the *subsumption* operation has been missing so far. This thesis fills this specific gap by presenting and comparing different implementation alternatives (Bleiholder et al., 2010a). It additionally introduces *subsumption* into relational query optimization by giving transformation rules that can be used to move the *subsumption* operator around in query plans. These contributions are presented in Chapter 5 and Chapter 6.

Implementation and query optimization for complementation and complement union
Another way of dealing with missing data and implementing a specific data fusion strategy are the *complementation* operator and the *complement union* operator. Similarly to *subsumption* and *minimum union*, *complement union* is the combination of *complementation* and *outer union*. *Complementation* has been firstly defined as part of this work (Bleiholder et al., 2010a) and combines information from tuples with complementing attribute values. We also give different implementation variants of the *complementation* operator (Bleiholder et al., 2010b) and introduce the operator into logical query optimization. The transformation rules given can be used to move the operator around in query trees. These contributions are also part of Chapter 5 and Chapter 6.

Research prototypes HUMMER and FUSEM
Lastly, this thesis contributes a large part to two research systems: the extension of SQL as presented in

Chapter 4 is used in the HumMer system (Bilke et al., 2005; Naumann et al., 2006), as well as the *data fusion* operator, also defined in Chapter 4. Another research system that has been developed within our research group is the FuSem system (Bleiholder et al., 2007). It implements different data fusion semantics and allows the user to compare them. An overview of both systems and how they employ the different data fusion techniques presented in this thesis is given in Chapter 8.

This thesis is organized as follows: It begins in the first part with an introduction into the area of data fusion by introducing some information integration scenarios where data fusion is used, a running example from one scenario and some examples of how tables can be combined (Chapter 1). The thesis is then structured into three more parts.

In the second part we describe different aspects of data fusion. We start with the general problem of combining different object representations in form of a data integration process in Chapter 2. We also define the main goal of concise and complete integration and differentiate between the term data fusion as used in the information integration context and its use in other fields. Chapter 3 then introduces the concept of data conflict, distinguishes between missing and contradicting data and defines a catalog of conflict handling strategies. We also define the concept of conflict resolution function, give examples, and show how theses functions can be used to implement conflict handling strategies and finally to resolve conflicts. In this chapter we further report on user experiments on choosing conflict resolution functions. The last chapter in this part, Chapter 4, formally introduces the basic notation that is used to define three data fusion operators: In order to handle missing information we introduce the *subsumption* and the *complementation* operators and subsequently their combination with *outer union* to the *minimum union* and *complement union* operators that are able to combine two or more tables. Whereas *subsumption* and *minimum union* have been introduced before, we define *complementation* and *complement union* as new operators in this thesis. We define the *conflict resolution* and *data fusion* operators that are able to handle contradicting information by applying conflict resolution functions on the data. The chapter closes by giving a way to embed the new operators into the Sql language.

In the third part we cover the more practical aspects of data fusion and show how to implement and use the operators. Chapter 5 gives different implementation variants for all the operators defined in the previous chapter. It introduces efficient techniques to compute *subsumption*, *complementation* and the *data fusion* operator. We then introduce in Chapter 6 the operators into logical relational optimization by giving transformation rules to move the operators around in data fusion query trees. The third part ends with the documentation of extensive experimentation in Chapter 7 that shows the feasibility and scalability of our techniques and also the differences between the different variants. At last we show the performance gain by some of the transformation rules.

The last part covers the surrounding of data fusion, namely two research prototypes that have been developed within our research group and related work. Chapter 8 first describes the HumMer and the FuSem system that are both research prototypes. Whereas the HumMer system integrates data from multiple sources and does all the necessary integration steps (*schema matching*, *duplicate detection*, and *data fusion*), the FuSem system is specifically targeted at supporting the data fusion step in data integration. Both systems serve to showcase the data fusion techniques developed as part of this thesis. Chapter 9 contains a survey on related work on data fusion techniques and data integration systems and their data fusion capabilities. We conclude in Chapter 10 and give an outlook to future research and open issues in the field of data fusion.

Part II

Describing Data Fusion

The whole is more than the sum of its parts.

(Aristotle)

2

Combining Representations of Objects

In this chapter, *data fusion* is presented in a larger context, as it is used in the field of data integration. Data integration is considered a process and combines different representations of same real-world objects. After a short introduction on how to identify real-world objects in digital systems in Section 2.1, in Section 2.2 a prototypical data integration process is presented in which *data fusion* is one of three main building blocks. Along the process we further outline the preliminaries of *data fusion*, the larger goal of data integration, and means of how to describe and compare different integrated object representations in Section 2.3. Interestingly, the term *data fusion* bears quite a few meanings throughout different fields and disciplines, not only in computer science. In Section 2.4 we present how the term *data fusion* or *fusion* is used in other fields and show their differences and similarities.

2.1 Objects, Identity and Identifiers

Data integration deals with digital representations of real-world objects. While real-world objects, such as specific persons, books, or other objects of the material world, are essentially unique, their representations in the digital world are usually not. Usually information on real-world items is stored in many different locations and systems. As distinction, we speak of an object having a real-world *identity* and of its digital representations having an *identifier*. In an ideal world, each identity would also have its own unique digital identifier. In reality, different identifiers are used to denote the same real-world identity. E.g., when storing customer data, different companies use different systems to uniquely identify their customers. Besides a customer identity being represented multiple times in different sources (using different identifiers), she may also be represented multiple times in one source. To make matters even more complicated, same identifiers can also be used to denote different identities. This is the case, when two different companies use integers as customer identifiers. That way, two different people may end up with the same customer number. So, even if an identifier is locally unique, it is not necessarily also globally unique.

To reason about identities in the digital world we need to conceptually introduce a surrogate, a globally unique digital identifier that represents the real-world object. We name this global identifier real-world ID. Local identifiers, where used, are mapped to such a real-world ID. The relation between identity and identifiers is illustrated in Figure 2.1. In the remainder of this thesis, when we speak of an identifier, in most cases we mean such a real-world ID. However, to make examples not overly complicated, we sometimes use the name of a person or other attributes as real-world ID, which works perfectly in the example at hand but does not scale to the real-world.

To illustrate the data integration process presented in the following section and the differences between its parts, we regard an example of two relations that shall be integrated. Each relation stores information on persons. The relations overlap both in the attributes and the people that are contained in the relations. E.g., both relations store the birthday, and they both contain information on a real-world person (identity) named *George Smith*. On the other hand, some information is only contained in one relation, e.g., the address, or information on the person named *Paula Miller*. Table 2.1 shows the starting point of data integration – the two source relations. Note that they are schematically heterogeneous and contain data level conflicts. The

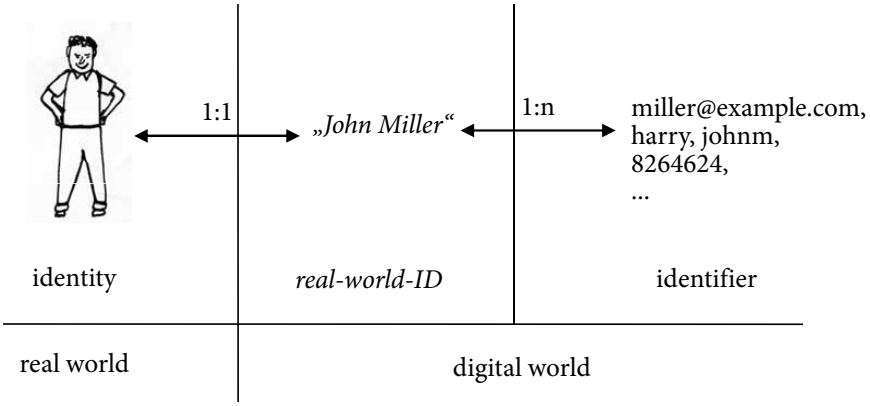


Figure 2.1: Relationship between an object of the real-world having an identity and its different representations in the digital world (identifiers).

different identities are marked by capital letters in an additional column (REAL-WORLD ID) to show that one real-world identity has multiple representations in the digital world. As we subsequently want to also refer to single representations, a tuple ID is also given.

POLICE

Tuple-ID	*Real-world ID*	**Firstname**	**Lastname**	**DOB**	**Sex**	**Address**
1	*A*	Frank	Peters	1/19/1953	M	43 First St.
2	*B*	George	Smith	8/9/1970	M	Mass Ave.
3	*C*	⊥	John-Paul Miller	4/15/1959	M	20 State Blvd.
4	*D*	Paul	Miller	7/7/1959	M	234 Main St.
5	*C*	Miller	Paul	4/15/1959	M	20 State Blvd.

(a) Source relation POLICE

HOSPITAL

Tuple-ID	*Real-world ID*	**Name**	**Birthday**	**Gender**	**Blood**
6	*A*	Frank Peters	Jan, 19^{th}1953	Male	AB
7	*A*	Frank Peters	Jan, 19^{th}1953	Male	⊥
8	*B*	George Smith	Sep, 8^{th}1970	Male	o
9	*E*	Paula Miller	Sep, 8^{th}1926	Female	B
10	*E*	Paula Miller	Sep, 8^{th}1962	Female	B

(b) Source relation HOSPITAL

Table 2.1: Source relations POLICE and HOSPITAL with information on people. Different digital representations of same identities are marked by different tuple ids, same identities are given a real-world ID and marked by same capital letters.

2.2 The Data Integration Process

Integrated information systems provide users with a unified view of multiple, heterogeneous data sources. Querying the underlying data sources, combining the results, and presenting them to the user is performed by the integration system. In the remainder we assume an integration scenario with a five step data integration process as shown in Figure 2.2 (Naumann et al., 2006).

When integrating data, as first main step, after some preprocessing (Step 1, *data preparation*), one needs to identify corresponding attributes that are used to describe the objects in the sources (Step 2, *data transformation*). The result of this step is a *schema mapping*, which is used to transform source data into a common representation. In this step, the problem of resolving naming conflicts is tackled. As a second important step, the different objects that are described in the data sources need to be identified and aligned. Here we deal with the problem of resolving identity conflicts. Using *duplicate detection* techniques (Step 3, *duplicate detection*),

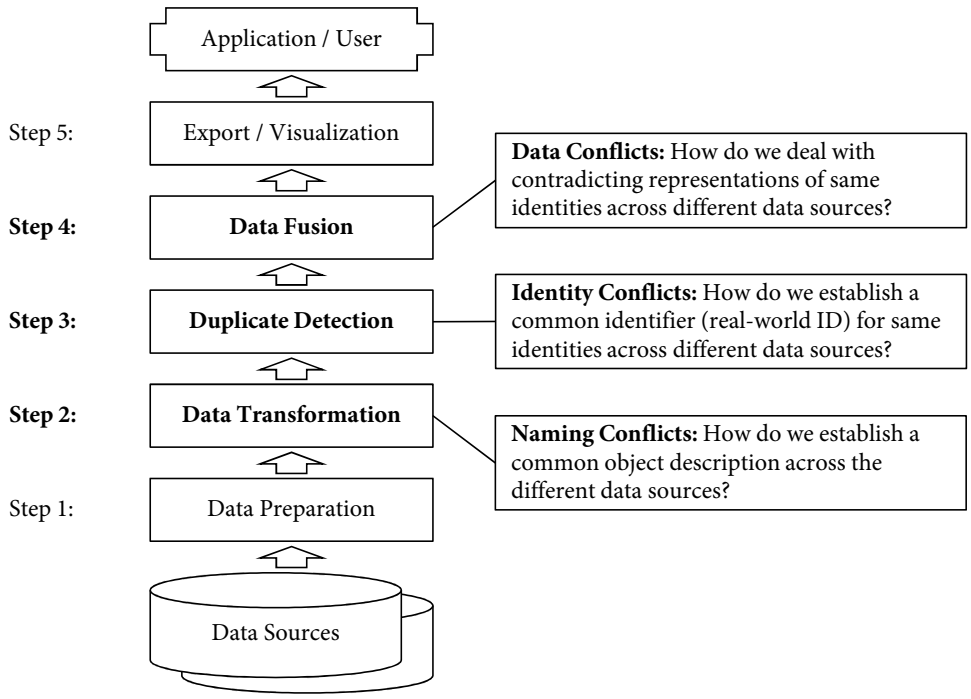


Figure 2.2: A data integration process consisting of five steps.

multiple, possibly inconsistent representations of same real-world objects are found and an object identifier is assigned. Then, duplicate representations are combined and fused into a single representation (Step 4, *data fusion*) while inconsistencies in the data are resolved (tackling data conflicts). This step is the main focus of this thesis. At last, results are shown or specially visualized to a user, or suitably exported for use in other applications (Step 5, *visualization/export*). We briefly discuss all steps in the following sections.

2.2.1 Data Preparation

Preparing source data helps improve the results of data integration, especially the resolution of naming and identity conflicts is simplified. This step and the techniques that are applied here are referred to as *data scrubbing* in the literature[1]. Among the main techniques that are used in data preparation are restructuring and standardization. Restructuring, e.g., deals with the problems of incorrectly assigned values, i.e., an attribute value that has been assigned to the wrong attribute, and embedded values, i.e., an attribute that contains more information. An example for the former would be a first name in the last name attribute and vice versa, and an example for an embedded value would be a first name that is included in the last name attribute (see Table 2.2).

Restructuring employs simple transformation rules to introduce new columns, move data from one column to another, split data into two columns, etc. Potter's Wheel (see Section 9.2.3 or (Raman and Hellerstein, 2001) for an overview) is an example for a tool that is able to accomplish such tasks. Restructuring can also be done with standard commercial ETL tools or in a restricted way also with standard DBMSs.

Standardization deals with different representation formats for the same type of information. Two examples for standardization are depicted in Table 2.2: time and date format standardization and unit conversion for measurements. Other examples include titles/addresses or names of cities: "New York City", "New York, NY", or "The Big Apple" are just three different ways of denoting the city of New York. Standardization of the spelling of cities could benefit the following steps in the integration process. If the amount of different possible values of an attribute is very large (e.g., addresses, bank accounts, phone numbers), reference datasets are used to standardize spellings. Usually, postal services offer tools or dedicated datasets with a standard spelling of

[1]The term *data cleansing* is also used in this context, although it is used more often as generic term to denote all different kinds of techniques to improve data quality.

First Name	Last Name
Doe	John
Jane Woo	⊥

⇒

First Name	Last Name
John	Doe
Jane	Woo

(a) Restructuring

Name	Birthday	Weight
John Doe	June, 5th 1976	3400g
Jane Woo	23.6.77	2.956kg

⇒

Name	Birthday	Weight
John Doe	06/05/1976	3400g
Jane Woo	06/23/1977	2956g

(b) Standardization

Table 2.2: Examples for simple data preparation techniques: restructuring and standardization.

cities, streets, and the corresponding postal code[2]. Using such a reference dataset could also be used to verify dependencies between values in different attributes, such as the dependency between zip code and city.

Police

Firstname	Lastname	DOB	Sex	Address
Frank	Peters	1/19/1953	Male	43 First St.
George	Smith	8/9/1970	Male	Mass Ave.
John-Paul	Miller	4/15/1959	Male	20 State Blvd.
Paul	Miller	7/7/1959	Male	234 Main St.
Paul	Miller	4/15/1959	Male	20 State Blvd.

(a) Relation Police

Hospital

Name	Birthdate	Gender	Blood
Frank Peters	1/19/1953	Male	AB
Frank Peters	1/19/1953	Male	⊥
George Smith	9/8/1970	Male	o
Paula Miller	9/8/1926	Female	B
Paula Miller	9/8/1962	Female	B

(b) Relation Hospital

Table 2.3: Relations Police and Hospital after the data preparation step. In comparison to the original relations, names have been restructured, date formats and permitted values for sex/gender information has been standardized.

To illustrate, Table 2.3 shows the source relations of Table 2.1 after the data preparation step. As one can see, names have been restructured and the date format of the birthdays has been adjusted as well as the allowed values for the sex/gender attributes. These basic data scrubbing operations result in tables that are better suited for the subsequent steps of *data transformation* and *duplicate detection*.

2.2.2 Data Transformation

Integrated information systems usually have to deal with heterogeneous schemata at the sources. In order to present query results to the user in a single unified schema, schematic heterogeneities must be handled. Source data needs to be transformed to conform to the global schema of the integrated information system. Two approaches are common to bridge heterogeneity and thus specify data transformation: schema integration and schema mapping. The former approach – schema integration – is driven by the desire to integrate a known set of data sources. Schema integration regards the individual schemata and tries to generate a new minimal and understandable schema that is complete and correct with respect to the source schemata. Batini et al. (1986) give an overview of the difficult problems involved and the techniques to solve them. The latter approach – schema mapping – assumes a given target schema, i.e., it is driven by the need to include a set of sources in a given integrated information system. A set of correspondences between elements of a source schema and elements of the global schema are generated to specify how data is to be transformed (Popa et al., 2002; Melnik et al., 2005). Such a mapping can be created manually, however *schema matching* techniques semi-automatically find correspondences between two schemata. Schema matching techniques (see e.g., (Melnik et al., 2002; Madhavan et al., 2001; Bilke and Naumann, 2005)) have been classified based on what input information the methods use (Rahm and Bernstein, 2001). There is much ongoing research in both the areas of schema matching and schema mapping, see e.g., (Euzenat and Shvaiko, 2007) for an overview.

[2] e.g., `http://www.usps.com/business/addressverification/welcome.htm?from=business&page=addressver` as seen on November, 16th 2008

The goal of both approaches, schema integration and schema mapping, is the same: transform data of the sources so that it conforms to a common global schema. Given a schema mapping, either to an integrated or to a new schema, finding such a complete and correct transformation is a considerable problem (Fagin et al., 2005). The data transformation itself, once found, can be performed offline, for instance as an ETL process for data warehouses, or online, for instance in virtually integrated federated databases. After this step in the data integration process all objects of a certain type are represented homogeneously.

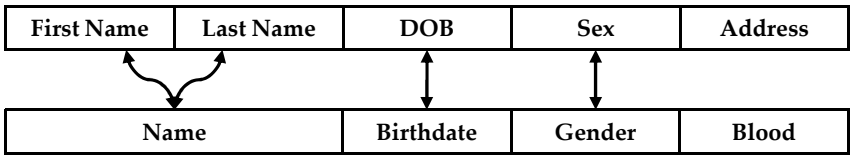


Figure 2.3: Mapping between the example source relations Hospital and Police.

As an example, Figure 2.3 shows the correct mapping between the two example relations as it is created manually or found by schema matching algorithms. "DOB" is correctly matched with *"Birthdate"* and *"Sex"* is mapped to *"Gender"*. The Figure also contains a 2:1 mapping between *"First Name/Last Name"* from one source and *"Name"* from the other source.

Police ⊎ Hospital

Source	Name	Birthdate	Gender	Address	Blood
P	Frank Peters	1/19/1953	Male	43 First St.	⊥
P	George Smith	8/9/1970	Male	Mass Ave.	⊥
P	John-Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
P	Paul Miller	7/7/1959	Male	234 Main St.	⊥
P	Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
H	Frank Peters	1/19/1953	Male	⊥	AB
H	Frank Peters	1/19/1953	Male	⊥	⊥
H	George Smith	9/8/1970	Male	⊥	o
H	Paula Miller	9/8/1926	Female	⊥	B
H	Paula Miller	9/8/1962	Female	⊥	B

Table 2.4: Relations Police and Hospital transformed into one relation, by using the mapping from Figure 2.3 and subsequently the *outer union* operator.

Using the mapping from Figure 2.3, we are now able to transform the data from the two source tables into one single table. This is done by first renaming/restructuring the attributes of the Police relation and then combining both relations by *outer union*. The transformed table from our running example is shown in Table 2.4. We additionally provide information on the source relation in a separate column. As the attributes Address and Blood are not matched, tuples are padded with null values if necessary.

2.2.3 Duplicate Detection

The next step of the data integration process is the step of duplicate detection (also known as record linkage, object identification, reference reconciliation, and many others). The goal of this step is to identify multiple representations of the same real-world object and to assign object identifiers.

At first sight, duplicate detection is simple: Compare each pair of representations using a similarity measure and apply a threshold. If a pair of representations is more similar than a given threshold, both representations are declared to be fuzzy duplicates and the same object identifier is assigned to both representations. In reality there are two main difficulties to be solved: effectiveness and efficiency.

Effectiveness is mostly affected by the quality of the similarity measure and the choice of a good similarity threshold. A similarity measure is a function determining the similarity of two representations. Usually the similarity measure is domain-specific, for instance designed to find duplicate customer representations.

Domain-independent similarity measures usually rely on string-distance measures, such as the Levenshtein-distance (edit-distance) (Levenshtein, 1965). The similarity threshold determines when two representations are considered fuzzy duplicates. A too low threshold will produce a high recall (many, at best all, real duplicates are found) but also a low precision (many non-duplicate pairs are wrongly declared to be duplicates). A too high threshold results in high precision, but low recall. Tuning the threshold is difficult, domain- and even dataset-specific.

sort key	**Name**	**Birthdate**	**Gender**	**Address**	**Blood**
FP53M	Frank Peters	1/19/1953	Male	⊥	AB
FP53M	Frank Peters	1/19/1953	Male	43 First St.	⊥
FP53M	Frank Peters	1/19/1953	Male	⊥	⊥
GS70M	George Smith	9/8/1970	Male	⊥	O
GS70M	George Smith	8/9/1970	Male	Mass Ave.	⊥
JM59M	John-Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
PM26F	Paula Miller	9/8/1926	Female	⊥	B
PM59M	Paul Miller	7/7/1959	Male	234 Main St.	⊥
PM59M	Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
PM62F	Paula Miller	9/8/1962	Female	⊥	B

Table 2.5: Sorting representations by a sort key. As sort key, the first letter of first name, last name, the year of the birthday and the sex is used.

Efficiency is an issue because datasets are often very large. Thus calculating the similarity of all pairs of representations can become an obstacle. Another obstacle of efficient duplicate detection is the complexity of the similarity measure itself, for instance because the expensive Levenshtein-distance is part of the similarity function. The first obstacle is overcome by an intelligent partitioning of the objects and comparison of pairs of objects only within a partition. A prominent example of this technique is the sorted neighborhood method (Hernández and Stolfo, 1998). The second obstacle can be alleviated somewhat by efficiently computing upper bounds of the similarity and computing the actual distance only for pairs whose bounds are higher than the upper bound (Weis and Naumann, 2004).

The result of the duplicate detection step is the assignment of an object identifier to each representation. Two representations with the same object identifier indicate duplicates. Note that more than two representations can share the same object identifier, thus forming *duplicate clusters*. It is the goal of data fusion to fuse these multiple representations into a single one. However, no duplicate detection technique is perfect and so the assigned object identifier is only an estimate of the real-world ID.

ID W=2	*ID W=4*	*Real-world ID*	**Name**	**Birthdate**	**Gender**	**Address**	**Blood**
1	*1*	*A*	Frank Peters	1/19/1953	Male	⊥	AB
1	*1*	*A*	Frank Peters	1/19/1953	Male	43 First St.	⊥
1	*1*	*A*	Frank Peters	1/19/1953	Male	⊥	⊥
2	*2*	*B*	George Smith	9/8/1970	Male	⊥	O
2	*2*	*B*	George Smith	8/9/1970	Male	Mass Ave.	⊥
3	*3*	*C*	John-Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
4	*4*	*E*	Paula Miller	9/8/1926	Female	⊥	B
5	*5*	*D*	Paul Miller	7/7/1959	Male	234 Main St.	⊥
6	*3*	*C*	Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
7	*4*	*E*	Paula Miller	9/8/1962	Female	⊥	B

Table 2.6: Using a window size of 2 (left ID column), not all present duplicates are found. A window size of at least 4 (right ID column) is needed to be able to correctly find all duplicates present in the sources. For each window size, the resulting object identifier is given in an additional column.

To proceed with the example we now find duplicate representations using one possible algorithm, the sorted-neighborhood-method (Hernández and Stolfo, 1995). Table 2.5 shows the first step of the sorted-neighborhood-method applied to our example data. A sort key is created by appending the first letter of the

first and the last name, the year of the date of birth and the first letter of the GENDER attribute. As we can see, after sorting by the key, some representations of identities, such as *Frank Peters* or *George Smith* are sorted next to each other, whereas others (e.g., *Paula Miller*) are not. In a second step, a sliding window technique is applied, so that each tuple is only compared to the other tuples in the window. In real-world scenarios, a window size of 20 is usually sufficient. In our example (see Table 2.6), a window size of 4 is already large enough to find all existing duplicates. As we also see, a window size of only 2 is not sufficient. The resulting object identifiers are provided as additional columns (column $ID\ W = 2$ for a window size of 2 and column $ID\ W = 4$ for a window size of 4).

2.2.4 Data Fusion

After the step of data transformation all objects are described in the same way. After duplicate detection we are also able to group together different representations of same real-world objects. The last issues that remain are the differences between the representations of an object, the different attribute values, such as different addresses, different phone numbers or misspellings in the names. The *data fusion* step in the data integration process deals with these data conflicts and comes up with a final representation of all objects that suit the users need when integrating the data.

Thus, data fusion tackles the problem of resolving data-level conflicts in a single table after schemata have been matched and duplicates have been detected. It creates the final representation for each distinct real-world object. However, a final representation does not need to be a single tuple, but may also consist of more than one, or even zero tuples. Different possible strategies of accomplishing this step in the process and come up with a final representation are described in more detail in the subsequent chapters. A conflict handling strategy, as presented in the next chapter, is a high-level guideline, separated from any technical implementation, and guides the user in handling conflicts. For example, relying on a primary source system and take the data from there, is a valid strategy to handle conflicts. Along with a strategy, applied to source tables, comes a result in form of a result table. Chapter 1 already introduced many different results and gave hints to the techniques, (relational) operators, or algorithms involved. Some of these techniques are handled in more detail in this thesis, especially the *minimum union*, the *complement union*, and the *data fusion* operator.

ID W=4	*Real-world ID*	**Name**	**Birthdate**	**Gender**	**Address**	**Blood**
1	*A*	Frank Peters	1/19/1953	Male	43 First St.	AB
2	*B*	George Smith	9/8/1970	Male	Mass Ave.	O
3	*C*	John-Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
4	*E*	Paula Miller	9/8/1962	Female	⊥	B
5	*D*	Paul Miller	7/7/1959	Male	234 Main St.	⊥

Table 2.7: Result after data fusion using the object ids ($ID\ W = 4$) from duplicate detection.

One possible final result table of our example can be seen in Table 2.7. Conflicts are resolved by applying the *data fusion* operator, which aggregates the different object representations of the example into one single representation, preferring NON-NULL values over NULL values, preferring values from relation HOSPITAL over values from relation POLICE, and taking longer values for the *Name* attribute and newer birthdates.

There exists some overlap and interdependence in the things that are done in the steps of data preparation and data fusion. Some of the issues that are mentioned in the section on data preparation (e.g., different date formats) can of course also be considered data-level conflicts and thus handled later during data fusion. Among the things that can be handled later (see, e.g., Table 2.2) are the conflicts that arise because of non-standardized values, but also the conflicts due to unstructured data. We introduced the step of data preparation and thus allow the resolution of some issues already early on in the data integration process because it helps the subsequent steps of data transformation and duplicate detection. In particular, we allow *conflicts*[3], which resolution does not depend on or uses the object identifier information created during duplicate detection, to

[3] We cannot really speak of a conflict here, because the information on the real-world object it belongs to is missing in the data preparation step. However, it later may become a conflict.

be done early during data preparation. However, if the data preparation step is missing, all these *conflicts* are handled during data fusion.

2.2.5 Export and Visualization

Just as the first step in our data integration process consisted of preparing data from the sources, the last step consists of preparing the data for use in an application or to be shown to a user. In the easiest case, nothing needs to be done, as the output from the data fusion step can directly be used by the application. However, another transformation into the schema used by the application is possible and could be done during this step. Here, the same techniques as described earlier (e.g., restructuring, standardizing) could be used.

Name	Birthdate	Gender	Address	Blood
Frank Peters	1/19/1953	Male	43 First St.	AB
George Smith	9/8/1970	Male	Mass Ave.	O
John-Paul Miller	4/15/1959	Male	20 State Blvd.	⊥
Paula Miller	9/8/1962	Female	⊥	B
Paul Miller	7/7/1959	Male	234 Main St.	⊥

Table 2.8: Result with data level conflicts being color coded: Green indicates one single value (light green) or same values in different representations (darker green), orange indicates a conflict between a value (uncertainty) and a NULL value and red marks real contradictions.

Concerning visualization, one possible technique is to color code the result. Table 2.8 shows an example for such a color coding: A green background indicates one single value (light green) or same values from different representations (darker green). An orange background indicates a conflict between a value and a NULL value and red marks real contradictions of two or more different values. Chapter 8 shows more examples for visualizations of data fusion results.

2.3 Complete and Concise Data Integration

A data integration process has two broad goals: Increasing the completeness and increasing the conciseness of data that is available to users and applications. An increase in completeness (all information on objects is included) is achieved by adding more data sources (more objects, more attributes describing objects) to the system. An increase in conciseness (information on objects is given only once and without contradictions) is achieved by removing redundant data, by fusing duplicate entries and merging common attributes into one. This distinction (completeness vs. conciseness) is along the lines of related work such as (Motro, 1986; Naumann et al., 2004; Scannapieco et al., 2004).

In analogy to the well known precision/recall measure from information retrieval, one can define measures of conciseness/completeness when regarding unique and additional object representations in the considered universe and the actual dataset that is being measured (see Figure 2.4 on page 23 for notation).

2.3.1 Completeness and Conciseness

In analogy to recall, completeness of a dataset, such as a query result or a source table, measures the amount of data in that set. The amount is measured relative to all data present in the universe of discourse. Completeness can be measured both in terms of the number of tuples (extensional, data level) and the number of attributes (intensional, schema level).

Extensional completeness is the number of unique object representations in a data set (a) in relation to the overall number ($a + c$) of unique objects in the real-world, e.g., in all the sources of an integrated system (see Figure 2.4 for visualization and notation). It measures the percentage of real-world objects covered by that dataset. We assume that we are able to identify same real-world objects, e.g., by an identifier created during *duplicate detection*.

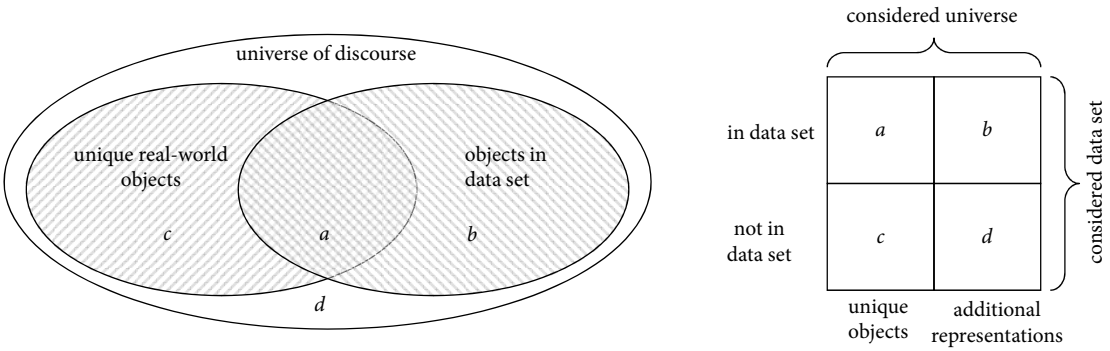


Figure 2.4: Measuring completeness and conciseness in analogy to precision and recall, letters *a*-*c* denoting the number of objects in that region, e.g., *b* being the total number of objects in the considered data set, *c* being the number of all unique real-world objects, and *a* being the number of unique real-world objects that are in the considered data set.

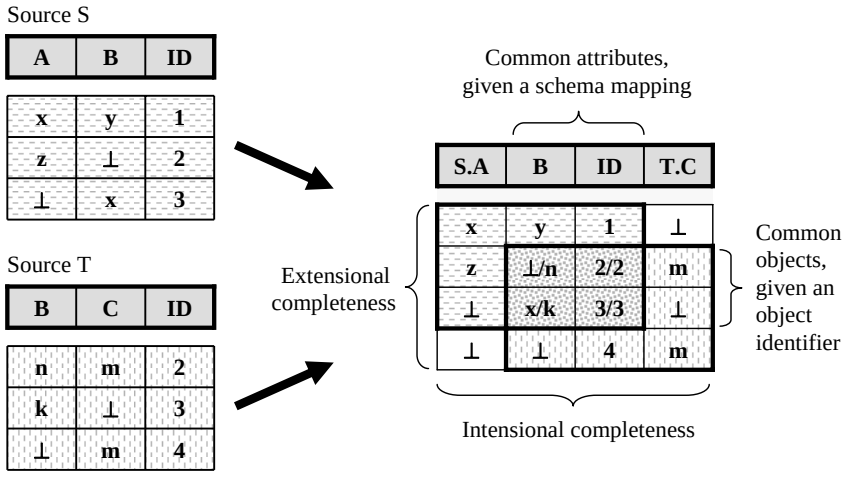


Figure 2.5: Visualizing extensional and intensional completeness when integrating two data sources (source *S* and *T* with their respective schemas (A, B, ID) and (B, C, ID)) into one integrated result, using information from schema matching (common attributes, here identified by same name) and duplicate detection (common objects, here identified by same values in the *ID* column).

$$\text{extensional completeness} = \frac{\| \text{ unique objects in dataset } \|}{\| \text{ all unique objects in universe } \|} = \frac{a}{a + c}$$

The example in Figure 2.5 shows the combination of two sources. Given a real-world ID in the ID column, there exist in total four real-world objects (four distinct values in column ID). The combination of the two sources as given in Figure 2.5 has an extensional completeness of 1 (=4/4) and is therefore maximal in the example, whereas both sources *S* and *T* are extensionally incomplete (extensional completeness: 3/4). In general, an increase in extensional completeness is achieved by adding more unique objects.

Intensional completeness is the number of unique attributes in a data set in relation to the overall number of unique attributes available. An increase is achieved by integrating sources that supply additional attributes to the relation, i.e., additional attributes that could not be included in one of the schema mappings between the sources considered so far. The result in Figure 2.5 has an intensional completeness of 1 (=4/4), because it contains four unique attributes (S.A, B, ID, T.C) out of all four unique attributes (S.A, S.B, S.ID, T.B, T.ID, T.C, with mappings S.B ↔ T.B and S.ID ↔ T.ID). Likewise, sources *S* and *T* have an intensional completeness of 3/4.

In analogy to precision, conciseness measures the uniqueness of object representations in a dataset. In the

terms of Figure 2.4, *extensional conciseness* is the number of unique objects in a dataset (a) in relation to the overall number of object representations in the dataset ($a + b$).

$$\text{extensional conciseness} = \frac{\| \text{ unique objects in dataset } \|}{\| \text{ all objects in dataset } \|} = \frac{a}{a+b}$$

The example result in Figure 2.5 results in an extensional conciseness of 1(=4/4), containing four different objects (as given by object identifier ID) out of four different object representations (tuples) in the result. Similarly, *intensional conciseness* measures the number of unique attributes of a dataset in relation to the overall number of attributes. Again, intensional conciseness follows as 1 (=4/4), as all four attributes in the result are different.

2.3.2 Four Degrees of Integration

To further illustrate the notions of completeness and conciseness when integrating data sources, Figure 2.6 visualizes four different ways of combining the example tables (sources S and T) from Figure 2.5 into one result. The generalization to more than two sources is conceptually trivial but difficult to visualize.

S.A	S.B	S.ID	T.ID	T.B	T.C
x	y	1	⊥	⊥	⊥
z	⊥	2	⊥	⊥	⊥
⊥	x	3	⊥	⊥	⊥
⊥	⊥	⊥	2	n	m
⊥	⊥	⊥	3	k	⊥
⊥	⊥	⊥	4	⊥	m

(a) No mapping and no object identifier used

S.A	B	ID	T.C
x	y	1	⊥
z	⊥	2	⊥
⊥	x	3	⊥
⊥	n	2	m
⊥	k	3	⊥
⊥	⊥	4	m

(b) Mapping (S.B↔T.B, S.ID↔T.ID) used, but no object identifier used

S.A	S.B	ID	T.B	T.C
x	y	1	⊥	⊥
z	⊥	2	n	m
⊥	x	3	k	⊥
⊥	⊥	4	⊥	m

(c) Partial mapping (S.ID↔T.ID) used, object identifier (S.ID, T.ID) used

S.A	B	ID	T.C
x	y	1	⊥
z	n	2	m
⊥	x	3	⊥
⊥	⊥	4	m

(d) Full mapping (S.B↔T.B, S.ID↔T.ID) and object identifier (S.ID, T.ID) used

Figure 2.6: Four degrees of integration, using to various degrees the information on schema mappings and object identifiers.

Without schema mapping information and the knowledge of object identifiers, the best that an integrating system can do is produce a result as seen in Figure 2.6(a). While this result has a high completeness, it is not concise. Knowledge about common attributes, given by the schema mapping, allows results of the shape as seen in Figure 2.6(b). Incidentally, this shape is the result of an *outer union* operation on the two source relations (mapping attributes with same names). We call such results intensionally concise: No real-world property is represented by more than one attribute, the full schema mapping is used.

Knowledge about common objects, e.g., using a globally consistent ID, such as the ISBN for books, or using an globally consistent object identifier created by duplicate detection methods, allows results as seen in

Figure 2.6(c). Here, the only known common attribute is ID (mapping inferred from ID being object identifier in both sources *S* and *T*). We see further on how such a result can be formed using the *full outer join* operation on the IDs of the two source relations. We call such results extensionally concise: No real-world object is represented by more than one tuple.

Finally, Figure 2.6(d) shows the result after mapping same attributes (using a schema mapping) *and* same objects (using an object identifier). The main feature of this result is that it contains only one tuple per represented real-world object *and* each row has only one value per represented attribute. Such a result (intensionally and extensionally concise) is the ultimate goal of data fusion. As could be noted by the watchful reader, the result in Figure 2.6(d) contains only the value x as the value for attribute B and object with $ID = 3$, although k could also be a valid value (contained in source T). Although it is contained in both sources, the ID attribute is a special case: it cannot contain data conflicts because it is used to identify real-world objects. Dealing with these data conflicts in the overlapping region of the two sources (marked by the checked pattern in the results), is integral part of data fusion.

When considering related work in Chapter 9 and the examples from Chapter 1, we see that most integration techniques and integrated information systems increase extensional as well as intensional completeness. One can regard this behavior as the primary feature of most integrating information systems. However, only few also consider the problem of conciseness.

2.4 The Term (Data) Fusion in Other Fields

The term *(data) fusion* is also used in fields other than in the field of data integration. We already described in detail what fusion means in the data integration context. In this section we cover some other fields where the term is used with a different meaning.

2.4.1 Fusion in Information Retrieval

The field of information retrieval covers the retrieval of relevant documents out of a large collection of documents. Search engines are common examples for information retrieval systems. Queries in modern search engines consist of a set of (natural language) keywords. The search engine then uses these keywords to retrieve relevant documents. A single document is considered relevant, if one of the keywords of the query is contained in the document. It is considered even more relevant if more than one keyword is contained and usually different keywords are weighted differently. The final result that is produced by the system is a ranked list of documents, the documents ranked in decreasing relevance. There exist a number of techniques how to compute similarities between query and documents and thereby to determine relevance of a document. See e.g., (Baeza-Yates and Ribeiro-Neto, 1999; Salton and McGill, 1986; Witten et al., 1999; Rijsbergen, 1979; Mitra and Chaudhuri, 2000; Crestani et al., 1998; Kobayashi and Takeda, 2000; van Rijsbergen, 2001; Fuhr, 2001) for surveys, introductions, and more detailed information on the field of information retrieval.

One way of improving on the result of a single search engine is to query multiple search engines with the same query and integrate the result. This is done by meta search engines, such as `www.metacrawler.com` or `www.metager.de`. Computing a single ranking for the meta search engine given the different rankings of the search engines integrated is referred to as *data fusion* or *rank merging* (see e.g., (Tsikrika and Lalmas, 2001; Lam and Leung, 2004)). The approaches are manifold and range from simply averaging the rank positions, over computing a totally new ranking using some or all parts of the found document to totally different ways of coming up with a new ranking.

Rank merging in information retrieval is similar to data fusion in data integration as in both cases different representations of same objects contain conflicting data values in their describing attributes. However, the problem of rank merging in information retrieval cannot easily be transformed into a data fusion problem as used in relational data integration. It requires the transformation of the single search results into a relational format and transforming the rank merging algorithm to fit into the integration process. The main challenge is to transform the more complex algorithms into a combination of already existing relational operators, which is not always possible. However, simply considering the algorithm as a black box operator is always possible,

although using supplementary information in addition to information stored in the relations may be difficult or infeasible. Contrarily, the easiest case of rank merging, where only the rank position is used in determining the combined rank, rank merging maps easily to data fusion in data integration. A ranking is transformed into a relation with two columns: one that contains the URL and one that contains the rank position. Rank merging is then accomplished as a simple grouping/aggregation: by grouping on the URL and averaging on the rank position.

2.4.2 Fusion in Sensory Systems

The term data fusion is also used - along with sensor fusion or information fusion - in a community that spans several fields and areas and brings together researchers from different fields out of geography, mathematics, fuzzy systems, imaging, networking, and various fields of military research. Their common ground is the combination of information from different sensors to create new information. The field of geographical imaging is e.g., concerned with overlapping satellite images from the same region and is interested in deciding whether a specific piece of land is covered by forest or plains. In medical imaging the common ground is e.g., the combination of several photos to come up with a better picture to detect diseases. Military networking needs e.g., to combine different types of information from sensors (e.g., microphones, cameras or scent sensors in tanks, drones) to create a tactical overview on a battlefield. The field is highly diversified and connects people from such diverse research areas as artificial intelligence, optics, logics, and fuzzy system research.

A common activity in these fields is the aggregation of numerical data, e.g., temperatures coming from sensors, into only one data value. There is quite some work on aggregation operators and its properties and possible applications, as well as how to use fuzzy sets to model properties of things that are not known for sure (Cholewa, 1985; Li and Wu, 2004; Dubois and Prade, 2004, 1988). Fusion in this sense is similar to data fusion as used in data integration, as in both cases possibly conflicting data is handled (e.g., conflicting temperatures for the same location from different sensors). Some researchers have also tried to apply the techniques used in the field of fuzzy sets and aggregation functions to the problem of merging databases (Cholvy and Moral, 2001; Cholvy and Garion, 2002, 2004).

Fusion as used in the other sense, as creating new knowledge (e.g., the location of participants on the battlefield) by fusing sensor data is different to the meaning of data fusion in data integration as the concept of different representations of same real-world objects is missing. The term knowledge discovery would be a better way of describing the activities here. As (Oxley and Thorsen, 2004) point out when comparing the terms fusion and integration by applying category theory, there is an ongoing debate on the adequateness of the term data fusion in this case.

2.4.3 Fusion in Philosophy

In philosophy, mereology (Wikipedia, 2009; Varzi, 2009) is the theory of part-whole relationships. It tries to formalize parts and wholes and their relationships to each other. The theory is based on the notion of *parthood*, a reflexive, transitive, and antisymmetric relation, thus forming a partial ordering and being the basis of a mereological system. Examples for parts and wholes considered include: pages being part of a book, door being part of the house, but also joy being part of the game. However, not every relation that is reflexive, transitive, and antisymmetric constitutes a mereological system. Given a partial ordering, concepts such as *proper parthood*, *equality*, *overlap*, or *underlap* are defined, among others, and depending on the mereological system in question, are also included in the system.

One of these possible additional concepts is the concept of mereological *sum* or mereological *fusion*, which models the idea that if there exist some objects, then there exists a whole (called *sum*, or *fusion*) that consists exactly of these objects (a book is the fusion of its parts: its pages and cover). That means that any number of objects can be combined into something different, which is called the *fusion*. In that sense, the *fusion* in mereology is analog to the union in set theory. In contrast *data fusion* as used in data integration means a) the process of combination and not the result and b) requires as precondition that objects that are combined are representations of the same real-world object. In data integration the *fusion* (in the mereological sense) of different real-world objects is not intended.

2.4.4 Data Fusion in Market Research

Beyond computer science, in market research, the term data fusion is used when referring to the process of combining two datasets on customers with some overlapping characteristics (van der Putten et al., 2002). E.g., a customer database (recipient dataset) of a company is combined with data from a survey on TV viewing habits (donor dataset). This scenario is depicted in Figure 2.7. The goal is to predict the TV viewing habits of the customers although they have never been asked. The rationale behind such a procedure is that it is often cheaper to buy or use existing survey data, than to do a new customer survey. This procedure only works when the two datasets are (a) drawn from the same distribution, i.e., the donor dataset is representative for the customer dataset and (b) when both datasets have some characteristics in common, e.g., some demographic data (age, gender, income). The main idea behind data fusion as used in market research is to join the two datasets on the common demographic data by using a fuzzy *similarity join*. Often, an algorithm which is best described as a *k-nearest-neighbor-join* is used. In a first step, for every customer from the recipient dataset the k most similar items from the donor dataset are determined. This is done by using the common characteristics and a more or less complex similarity measure. For example, critical characteristics, such as sex, should be equal (e.g., to not predict female characteristics, such as a pregnancy, for men), whereas other characteristics only need to be similar, such as age. The second step of data fusion consists in determining the predictions for the customer characteristics from the donor dataset. Here averages, existing values, or distributions are used.

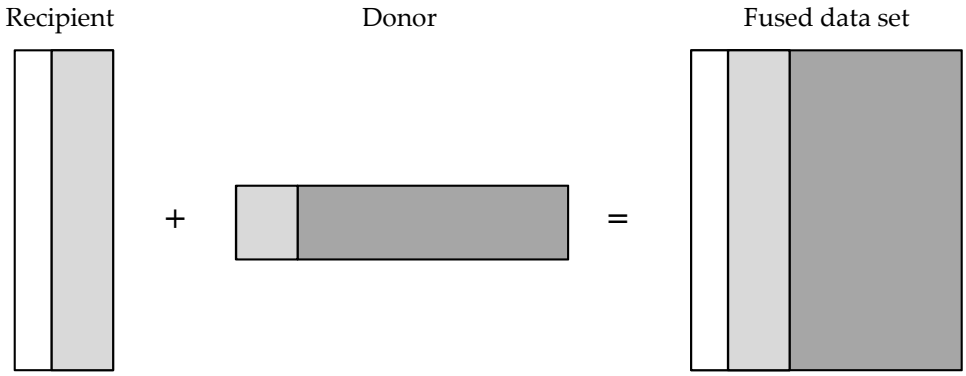


Figure 2.7: Data fusion as used in market research. Characteristics from a donor dataset are added (combined, fused) with data from a recipient dataset by joining on common characteristics.

One difference to data fusion in data integration is that data fusion in market research is a combination of what is separated in data integration: *duplicate detection* and *data fusion*. The *duplicate detection* part finds its equivalence in the fuzzy similarity measure that is used to join objects. However, in market research, no real duplicate detection is done, as – although the concept of identity exists – it is necessarily mitigated. The *join* does not pair representations of same real-world objects but only the most similar ones. It follows that every customer has a duplicate. The resolution of conflicts, however, can then be done by using the same techniques as used in data fusion in data integration.

There are three ways of dealing with difference: domination, compromise, and integration. By domination only one side gets what it wants; by compromise neither side gets what it wants; by integration we find a way by which both sides may get what they wish.

(Mary Parker Follett)

3

Dealing with Conflicts

In the following sections, we cover conflicting information as present in data integration. In particular, we focus on data conflicts and present means how to deal with them. Concerning data conflicts, in Section 3.1 we define two types of data conflicts, namely *uncertainties* (Section 3.1.1) and *contradictions* (Section 3.1.2). This section is followed by a detailed description of high-level conflict handling strategies in Section 3.2 and how they can be implemented on a lower level by our approach using conflict handling functions. Finally, we briefly cover how to combine and choose strategies and functions by reporting results from an user experiment, where fuzzy duplicates have been manually combined using conflict resolution functions.

3.1 Conflicting Data in Data Integration

In a data integration scenario, different ways of representing same real-world objects in the sources result in three types of conflicts. First, there are *schematic conflicts*, i.e., different attribute names are used to describe objects; or data sources describing the same objects are differently structured. Second, there are *identity conflicts*, i.e., same real-world objects are described multiple times in one or multiple data sources, or put in other words, multiple representations in one or more data sources have the same real-world identity. As we have seen in Chapter 2.2, these two types of conflicts are resolved in the first two phases of the data integration process (*data transformation* and *duplicate detection*). As a third type of conflict, *data conflicts* (conflicting information) remain, which are not resolved until the *data fusion* step. Such data conflicts are present, if for example for the same real-world identity (e.g., a student), semantically equivalent attributes, from one or more sources, do not agree on their attribute values (e.g., source 1 reporting "23" as the student's age, source 2 reporting "25").

Definition 1 (Conflict-Free Attributes)
An attribute A is defined as conflict-free*, if its set of values (including the special* NULL *value and duplicates removed) contains only one value. E.g., for an attribute* AGE*, the values* $\{23, 23, 23\}$ *(=* $\{23\}$*) are* conflict-free*, the values* $\{23, 24, 23\}$ *(=* $\{23, 24\}$*) and* $\{23, \perp, 23\}$ *(=* $\{23, \perp\}$*) are not* conflict-free. □

Definition 2 (Conflict-Free Objects and Relations)
Given representations of the same real-world object O as a set of tuples T. These representations of O are defined as conflict-free *if all attributes of the tuples in T are* conflict-free*. In addition, a relation R is defined as* conflict-free*, if all real-world objects represented in the relation are* conflict-free*.* □

Definition 3 (Attribute-Level Data Conflict)
An attribute-level data conflict *is present (in attribute A), if attribute A is not* conflict-free*.* □

Definition 4 (Data Conflict)
Given representations of a real-world object O. A data conflict *is present in object O, if the representations of O are not* conflict-free*. Similarly, a relation R contains a* data conflict *if there is a* data conflict *present for at least one real-world object O contained in the relation.* □

In the following we present examples for and distinguish two types of data conflicts: a) 'uncertainties' caused by NULL values (missing information), and b) 'contradictions' caused by different attribute values (contradicting information). In the remainder we use the terms *conflicting data* or *inconsistent data* interchangeably to both mean the same: the presence of data conflicts as defined above.

Data conflicts in the data sources exist at different granularities, depending on how the real-world objects are represented in the sources. In the most obvious representation (used in most examples throughout this thesis), real-world objects are represented in a table, a tuple of the table representing one object whose properties are described by the values in the columns. With this model, contradictions and uncertainties appear at attribute level, the values in the columns being in contradiction or being uncertain. The above definition of a *data conflict* and the remainder of this thesis assume this point of view. However, modeling objects at the granularity of tables and therefore properties being represented by tuples, results in looking at whole contradicting or uncertain tuples. That said, modeling properties of objects at the level of tables or even of data sources is also possible and results in looking at uncertain/conflicting tables or data sources. Please note that the granularity of the inconsistencies in the real-world object does not change, only the granularity of the representation in the data sources. While in Section 3.2.2 we consider and handle conflicts at attribute-level only, the strategies presented in Section 3.2.1 are also valid for other granularities.

3.1.1 Missing Data

Missing data in relational DBMS, is typically modeled by NULL values (commonly denoted by the symbol $\perp$)[1]. When adopting NULL values to represent missing information, we need to shortly present the different possible meanings (the different semantics) of NULL values in relational DBMS. The literature (Ullman et al., 2001) typically distinguishes three different semantics of NULL values:

1. Value *unknown*: There normally exists a value that makes sense, but that value is not known. An example for the *unknown* semantics is an unknown date of birth of a person.

2. Value *inapplicable*: There does not exist a value that makes sense in this context. An example for the *inapplicable* semantics is the information on spouse/husband for unmarried people.

3. Value *withheld*: There exists a value that makes sense, but it is not allowed to access it. A secret phone number is an example for the *withheld* semantics.

In the remainder of this thesis we assume most NULL values in a data integration scenario being *unknown* values. But even with considering NULL values as being *inapplicable* or *withheld*, the conflict handling strategies and techniques presented in the next sections and chapters remain valid.We now introduce a special type of data conflict, caused by missing information.

Definition 5 (Uncertainty)
We define an uncertainty *as a special kind of* attribute-level data conflict. *An* uncertainty *is present in an attribute A, if it is not* conflict-free *and its set of values (including the special* NULL *value and duplicates removed) contains only the* NULL *and one other* NON-NULL *value, e.g., the values* $\{23, \perp, 23\}$ *(*$= \{23, \perp\}$*).* □

Example 1 (Uncertainties)
As already mentioned in the introduction to this section, we are dealing with a data conflict if two representations of a student contain different information on the age of that particular student. If two representations report on the age of students in attribute AGE*, and one reports* 23 *as the student's age, and the other contains a* NULL *value in attribute* AGE *for that same student, then an uncertainty is present. The following table contains information about people, attribute* NAME *being an identifier:*

[1] Although there is a historically interesting debate whether introducing the concept of a NULL has been the right choice (see e.g., (Darwen and Date, 1995)).

Name	*Address*	*Date of Birth*	*Phone*
Alice	*Main Street 10*	*3/23/1976*	*555-9876*
Alice	*Main Street 10*	⊥	*555-1234*
Alice	*Main Street 10*	⊥	*555-1234*
Bob	*High Street 41*	⊥	*555-1234*
Bob	*High Street 42*	*8/12/1980*	⊥
Bob	*High Street 43*	*8/12/1980*	*555-6754*

The table contains two examples for uncertainties *in the representations of Alice and Bob. First, the three dates of birth of Alice in the three representations of Alice are different. As there is only one* NON-NULL *value (*3/23/1976*) and two* NULL *values, an* uncertainty *is present. Second, the three dates of birth of Bob are also different. This is also an example of an* uncertainty*, a* data conflict *between* NULL *and the* NON-NULL *attribute value* 8/12/1980*. Although there is a* NULL *value present among Bobs phone numbers, there is no* uncertainty *present in the phone attribute as the other representations contain two (and not just one) additional phone numbers.* □

In typical data integration scenarios, uncertainties are caused by missing information, i.e., sources overlap partially and therefore attributes are completely missing in one source, which accounts for the introduction of NULL values when several sources are integrated, e.g., by using *union* or *join*. However, NULL values already present in the sources are a second source for uncertainties. Semantics of NULL do not play a role when detecting uncertainties. However, as we see further on, it does play a role when handling uncertainties.

The reason for considering uncertainties as a special case of data conflicts is that they are generally easier to cope with than contradictions. To illustrate this point, consider the two different possible cases of uncertainty patterns when considering tuples, i.e., *subsumption* and *complementation*:

Subsumption: Whereas subsumption is formally defined in Section 4.2 we briefly introduce the concept here. A tuple subsumes another if it contains less NULL values than the other, but otherwise has same attribute values. In the following table, the first tuple subsumes the second:

Name	Address	Date of Birth	Phone
Alice	Main Street 10	3/23/1976	555-3427
Alice	Main Street 10	⊥	⊥

Complementation: The concept of complementation is also formally defined later on in Section 4.3. To understand the example it is sufficient to consider two complementing tuples as two tuples that have at least one corresponding attribute value in common, with all other corresponding attribute values either being equal or NULL. In the following table, the two representations for Alice complement each other:

Name	Address	Date of Birth	Phone
Alice	Main Street 10	⊥	555-3427
Alice	Main Street 10	3/23/1976	⊥

Strategies for handling data conflicts of the uncertainty type involve the elimination of NULL and duplicate attribute values as can be seen further on in Section 3.2. So in the two cases above the single tuple $(Alice, MainStreet10, 3/23/1976, 555-3427)$ would hold the same information in just one representation as in the two representations in the examples.

3.1.2 Contradicting Data

In contrast to uncertainties, NULL values are not important when considering contradictions.

Definition 6 (Contradiction)
A contradiction *is a special kind of* attribute-level data conflict*. A* contradiction *is present in an attribute A, if it is not* conflict-free *and its set of values (including the special* NULL *value and duplicates removed) contains at least two different* NON-NULL *values, e.g., the values* $\{23, 23, 24\}$ *(=* $\{23, 24\}$*).* □

Example 2 (Contradictions)
The example from the beginning of this Chapter (different values in the Age *attribute of a student) is a typical example for a contradiction. We consider the following table containing information about people and attribute* Name *being an identifier:*

Name	*Address*	*Date of Birth*	*Phone*
Alice	*Main Street 10*	*3/23/1976*	*555-9876*
Alice	*Main Street 10*	⊥	*555-1234*
Alice	*Main Street 10*	⊥	*555-1234*
Bob	*High Street 41*	⊥	*555-1234*
Bob	*High Street 42*	*8/12/1980*	⊥
Bob	*High Street 43*	*8/12/1980*	*555-6754*

The table contains three examples for contradictions in the representations of Alice and Bob. First, the three addresses for Bob are distinct and contradict in the street number. Second, the phone number of Alice is also not unique. As there are two distinct non-null *values (*555 – 9876 *and* 555 – 1234*), this is also a contradiction. Third, the phone number of Bob is also not unique, the three representations consisting of two distinct* non-null *values and a third* null *value. As there are two distinct* non-null *values, this last example is also a contradiction.* □

Contradictions are mainly introduced when partially overlapping sources report differently on the same properties of the same objects. See (Rahm and Do, 2000) and (Kim et al., 2003) for an overview of causes for contradictions in data and examples.

3.2 Conflict Handling

The following sections cover different aspects of conflict handling. Section 3.2.1 starts with a characterization and classification of conflict handling strategies. Then, we present conflict resolution functions as a common abstraction on a conceptual level to implement different strategies (Section 3.2.2) and characterize them by some properties (Section 3.2.3). Finally, Section 3.2.4 bridges the gap between strategies and functions and gives criteria that can be used when choosing a strategy. The section also reports on experiments as part of a manual deduplication effort to detect and fuse duplicates among audio compact discs.

Figure 3.1 depicts the interrelation between strategies and functions. Whereas strategies are on a very high abstraction level, they can be expressed using conflict resolution functions. On the technical level, when conflict handling strategies are actually used in (or as part of) a system or an application, one can rely on functions as building blocks for conflict handling or directly implement one of the strategies. Keep in mind that not all strategies can be implemented by functions. However, as we show further on, the most interesting can by implemented by functions. The step into the implementation level will be done in the next chapter (Chapter 4).

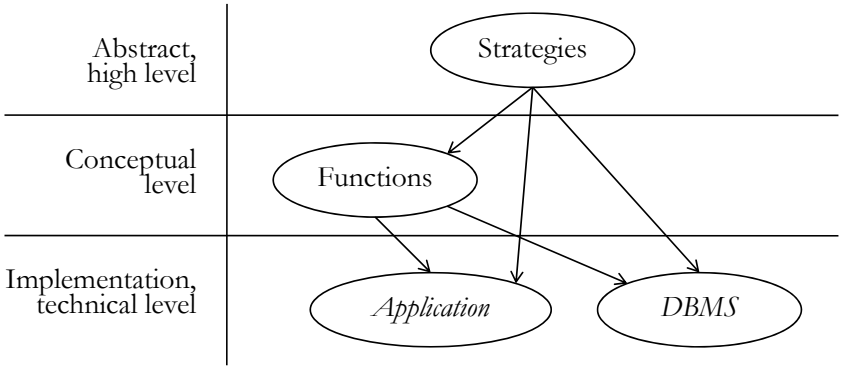


Figure 3.1: Strategies, functions, and their relation.

3.2.1 Conflict Handling Strategies

Conflict handling in data integration can be seen as a series of decisions, and a final decision on the correct value of an attribute. When dealing with inconsistent data, the first decision is whether or not the present data conflicts need to be acted upon. If we plan to remove data conflicts, the second decision usually is the decision for specific means of changing or interacting with the data until conflicts have been removed. Quite often this includes a direct decision on what attribute value to choose from a list of possible values.

Conflict handling strategies are high level strategies to handle conflicting data (Bleiholder and Naumann, 2006b). They model a general intuition of what to do in case of conflicting data. Some strategies even describe a precise final decision to take a certain value, to combine values or to invent a new value. In that way they describe a single, consistent representation that is created during data fusion.

There are several simple strategies to handle conflicting data, some of which are repeatedly mentioned in the literature. They can be classified as seen in Figure 3.2, and fall into three main classes, two of them consisting of several subclasses. The first division of strategies into the three classes is based on the way they handle (or do not handle) data conflicts present in the data: *conflict ignorance*, *conflict avoidance*, and *conflict resolution*.

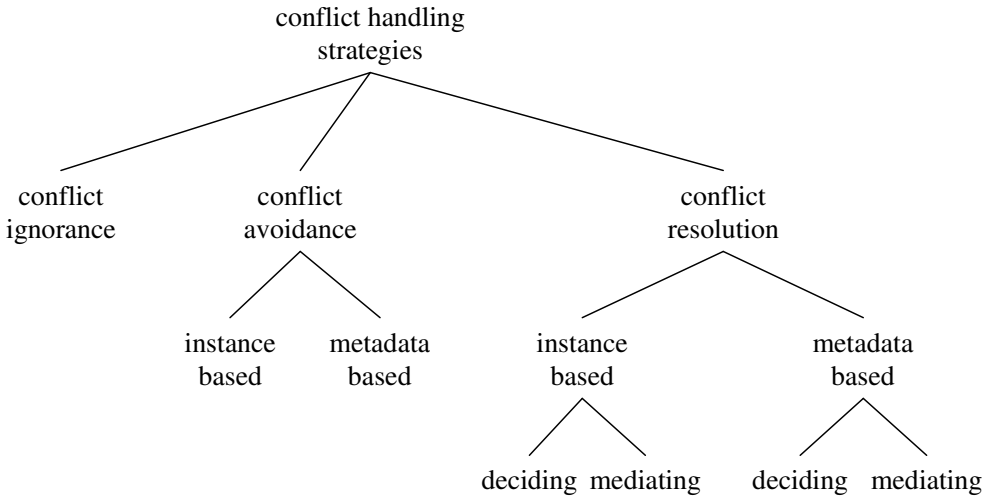


Figure 3.2: A classification of strategies to handle inconsistent data.

Conflict Ignorance

Conflict ignorance describes a class of strategies that do not include a direct decision at all concerning data conflicts. When employing such a strategy one not even needs to look for, detect or be aware of data conflicts in the data, as this information is not needed and not used. These strategies are applicable at all times and in all situations and are easy to implement. Two representatives are the Pass it on and the Consider all possibilities strategy:

Pass it on. This strategy simply takes all (possibly conflicting) representations and their attribute values, passes them on to the user or another application without changing them and lets the user or application decide how to handle possible conflicts. The decision of how to handle inconsistencies is deferred to another authority. After applying this strategy, data conflicts are still present in the data.

Consider all possibilities. In terms of completeness and conciseness as introduced in the last chapter, this strategy tries to be as complete as possible by enumerating all eventualities and giving the user the choice among all ”possible worlds” (Burdick et al., 2005), i.e., all possible combinations of attribute values. Using this strategy occasionally involves creating combinations of attribute values that are not already present in the source data. After applying this strategy, data conflicts are still present in the data.

Conflict Avoidance

Conflict avoiding strategies do not resolve single specific data conflicts, but nevertheless handle inconsistent data. They do not regard the specific conflicting values before deciding on how to handle inconsistencies. These strategies take a quick decision on whether to handle inconsistencies at all and if yes, which attribute values to use in that case. Because the decision is often done before regarding the actual values, these strategies are not always aware of a conflict. Therefore the name of conflict avoidance. They are more efficient from a computational point of view than the conflict resolving strategies considered later on, because the decision can be reached faster without looking at all conflicting values. On the other hand they lose precision because not all available information that can be valuable in resolving conflicts is taken into account.

This class of strategies can further be divided into two subclasses, one that takes metadata into account when taking a decision (*metadata based*) and one that does not (*instance based*). Two instance based strategies are Take the information and No gossiping:

Take the information. This strategy takes existing information and leaves aside null values. It is the natural way of dealing with uncertainties. The concept of subsumption, which filters out unnecessary null values and is used in the *minimum union* operator (Galindo-Legaria, 1994), and the use of Coalesce and *outer join* in the *merge* operator (Greco et al., 2001) are good examples for the use of this strategy.

No Gossiping. If you are unsure how to handle inconsistencies, why not leave them out and report only on the sure facts? This is the strategy used by the consistent query answering approaches (Arenas et al., 1999; Fuxman et al., 2005a). Here, only consistent answers are included in the result of a query, leaving aside all inconsistent ones. Because the decision is based on the data and inconsistent answers are ignored, this strategy is an instance-based conflict avoiding strategy. It is not, as one may think, a conflict ignoring strategy, because it correctly identifies conflicts and is aware of the conflicts.

An example of metadata based conflict avoidance is Trust your friends:

Trust your friends. The intuition behind this strategy is to trust somebody else to either provide the correct value or the correct strategy. Whom to trust is decided once and carried out for all data values, no matter if there is a conflict or not. This strategy can prefer data from one source over data from other sources and can be observed in the *TSIMMIS* (Papakonstantinou et al., 1996) and *Hermes* (Subrahmanian et al., 1995) systems. Source preference is given by the user, but can also be determined automatically by choosing the cheapest, most reliable, largest source, or by using other quality criteria as in the *Fusionplex* system (Motro et al., 2004a).

The main feature in conflict avoidance is that there is first a fundamental decision on whether to handle inconsistencies or not without considering the actual conflicting values. If they are handled, the decision on what actual data value to take or not to take is done first, before looking at the specific values and returning a result. Therefore, even if there is a conflict in the data, a system using these strategies does not necessarily know about it, noticing a specific conflict is avoided.

Conflict Resolution

In contrast to the previous classes, conflict resolution strategies do regard the data and metadata before deciding on how to resolve a conflict. This is computationally more expensive than other types of strategies but provides means to resolve the conflict as optimally as possible.

In contrast to the conflict ignoring and conflict avoiding strategies, conflict resolution strategies can be further subdivided into *deciding* and *mediating* strategies. The main characteristic of a deciding strategy is that it chooses its value from all the already present values (the domain of the possible values consists of all the conflicting values). Depending on the class, the choice of a value depends only on the data values (*instance based*) or takes some other information (*metadata based*) into account as well. Mediating strategies on the

other hand may choose a value that does not necessarily exist among the conflicting values. Such strategies may decide on a new value, that has not been present among the conflicting values.

Therefore, deciding strategies usually enable and easily allow for computing data lineage (Buneman et al., 2001), in particular *where-lineage*. In all cases it is clear where the value originates and therefore can be traced back to its origin. Data lineage information is usually not present with mediating strategy as values can be created arbitrarily and do not necessarily correspond to one of the existing values. Examples for instance-based deciding strategies are:

Cry with the wolves. The intuition behind this strategy is that correct values prevail over incorrect ones, given enough evidence. It reflects the principle of choosing the most common value among the conflicting values. Appropriate tie breakers are necessary.

Roll the dice. This strategy considers all values and picks one at random. Although this may not seem to be a very intelligent decision, it is still a valid strategy to resolve conflicts. Lacking any further information and input to decide upon a value, a random value is a good choice. It is still required that one is aware of a conflict, but it has the advantage of being computationally cheap.

An example of an instance-based mediating strategy is:

Better bend than break. This strategy follows the principle of compromise and does not prefer one value over the other. Instead it tries to invent a value that is as close as possible to all present values. Other possible principles used here are the computation of a value that minimizes some error, or taking the average value, rounding a value, or finding the least common denominator among the conflicting values.

Examples for deciding strategies based on metadata are:

Keep up to date. This strategy uses the most recent value and requires some additional timestamp data (recency information). This information can be present in the tables as a separate attribute or can be provided by other means, such as data lineage facilities. In a data stream environment there is a naturally given order of the tuples so that recency metadata lies in the data itself.

Ignorance is bliss. Decision making strategies based on ignorance proved to be considerably successful (Gigerenzer and Todd, 1999). Choosing among conflicting values based on the fact that some value is *known*[2] – in contrast to other unknown values – is another example for a metadata based deciding strategy. It is metadata based, because knowledge of a value is based on some external knowledge, e.g., an expert having knowledge of the value or some list or table containing the value.

Oops, I did it again. The kind of metadata that is used in this strategy is of historical nature. A value is chosen if the decision for this value has historically been proven successful, i.e., it is chosen if it has been chosen before. Implementing this strategy requires some sort of memory, a list of past decisions. This kind of strategy is used in the context of one-reason decision making (Gigerenzer and Todd, 1999).

An example for a mediating strategy based on metadata is:

Better bend than break II. If we use metadata to implement the Better bend than break strategy we reach a mediating strategy based on metadata. An example would be to take the lowest common ancestor according to a given taxonomy. Other operations involving a taxonomy or even an ontology are possible, such as taking a synonym, or another value that connects the conflicting values.

[2] *known* in the sense of valid: this is a valid value, it has been seen somewhere else, but it still might be the incorrect value.

3.2.2 Conflict Resolution Functions

In order to implement conflict handling strategies at attribute level we introduce the concept of conflict resolution functions. Conflict resolution functions are the building blocks on the conceptual level to implement conflict resolution strategies and are for use in systems or applications. As can be seen later on, the functions can be used together with a variety of approaches, including the *data fusion operator* (Bleiholder and Naumann, 2005). The main idea of this operator is to group multiple representations of one entity and apply a conflict resolution function to each column and thus – by applying the function to all (different) attribute values of the representations – fuse the data into a single representation.

Conflict resolution can be seen as a more general case of aggregation as known from the standard aggregation functions in SQL. The concept of attribute-level conflict resolution can be formalized as applying a function on the set of conflicting values (possibly with additional parameters). The function returns the resolved value as output. Such a function is defined on an input domain; both the domain and the functions possess some properties. After defining the concept of conflict resolution, some functions and their properties are presented. As we will see, conflict ignorant and conflict avoiding strategies also fit into the framework and can be realized using functions. Conflict resolution functions always handle conflicts between a set of values that are expected to contain conflicting values. The set may contain duplicate values.

Definition 7 (Conflict Resolution)
Conflict resolution is defined as the application of a function f_{cr} on a set of (possibly) conflicting values c resulting in a single resolved value s. □

We consider single- or multi-column functions, each of which may use additional information (metadata).

Definition 8 (Single-Column Conflict Resolution Function)
An n-ary single column conflict resolution function is a function f_{cr} defined on a domain D and maps a set C of n conflicting input values to one output value of the same or another domain S. Conflicting values ($c_i \in C$ are from domain D) are resolved to a solution s from domain S:

$$f_{cr} : D \times \ldots \times D \mapsto S \tag{3.1}$$
$$f_{cr}(C) = s, C = \{c_1, \ldots, c_n\}, s \in S, c_i \in D, \forall i = 1 \ldots n$$

□

Definition 9 (Multi-Column Conflict Resolution Function)
An n-ary multi-column conflict resolution function is a function f_{cr} defined on m domains D_j and maps a set C of n input m-tuples to one output value of another domain. The idea here is that conflicts are resolved in a column by using additional knowledge from other columns as well. The correspondences between values from different columns should not be lost, therefore the useof m-tuples:

$$f_{cr} : (D_1, \ldots, D_m) \times \ldots \times (D_1, \ldots, D_m) \mapsto S \tag{3.2}$$
$$f_{cr}(C) = s, s \in S$$
$$C = \{(c_1^1, \ldots, c_1^m), (c_2^1, \ldots, c_2^m), \ldots, (c_n^1, \ldots, c_n^m)\} c_i^j \in D_j, \forall i = 1 \ldots n, \forall j = 1 \ldots m$$

□

Definition 10 (Metadata Conflict Resolution Function)
For a single column conflict resolution function additional metadata information is given as a separate parameter m of domain M:

$$f_{cr} : D \times \ldots \times D \times M \mapsto S \tag{3.3}$$
$$f_{cr}(C, m) = s, C = \{c_1, \ldots, c_n\}, s \in S, m \in M, c_i \in D, \forall i = 1 \ldots n$$

□

Additional metadata information used by multi-column conflict resolution functions is modeled analogously. Input and output domains of conflict resolutions functions can be of the following types:

- **Numerical**: Numerical domains consist of numbers, the domain is ordered (numerical order) and infinite (e.g., integers: 1, 2, 3, ...).
- **Strings**: The String domain consists of words, based on characters. There exists an order (lexicographical order) and the number of values is infinite (e.g., names: Connery, Hanks, Cage, ...).
- **Date**: The date domain consists of dates (e.g., 1/1/2009, 3/16/2004, etc.) with the natural order given by a calendar. The domain is infinite.
- **Categorical**: There is no order defined on the values of the domain and the number of elements is finite. The elements can be strings or numbers (e.g., colors: {red, green, blue}, or grades: {1, 2, 3, 4, 5}).
- **Taxonomical**: A taxonomical domain consists of entities with a semi-order defined on them. The entities in the domain can be seen as structured like a tree and there is a finite number of elements (e.g., locations: Berlin, Germany, Europe, World, ...).

As we see later on, some of the functions require input data of a certain type of domain whereas other functions can be used with input data of more or all domains. The general goal is to use these functions to implement different strategies from Section 3.2.1. Table 3.1 on page 38 describes some possible functions, partly from (Bleiholder and Naumann, 2005). The properties of these functions are examined in the next section.

3.2.3 Properties of Conflict Resolution Functions

In order to understand conflict resolution functions better, we present and discuss some of their properties. Some of the properties are described in more detail in (Calvo et al., 2002) and other sources, some of the properties are relevant when implementing the functions in a research system, e.g., when considering efficient implementation of the functions, or the optimization of query plans involving functions. Table 3.2 on page 40 gives an overview of the functions and their properties that are described in the following:

- **Type**: A function is either a single- or multi-column function, depending on how many columns are used to decide on the resolved value. In addition, a function is either a parameter function, if it uses additional data (parameters), or a parameter-free function (without additional parameters). All standard aggregation functions known from SQL are examples for single-column, parameter-free functions, whereas CHOOSE DEPENDING is an example of a multi-column parameter function.
- **Applicable input domains**: To compute a conflict resolution it is important to know the different domains a function can be defined on. Functions like SUM or AVERAGE are defined on a numerical domain only, whereas MOST GENERAL CONCEPT requires a taxonomic input domain.
- **Deciding**: Deciding functions choose among the already present values, (e.g., MAX or SHORTEST) where $f_{cr}(c_1, \ldots, c_n) = c, c \in \{c_1, \ldots, c_n\}$. We assume that ties (e.g., two shortest values) are broken by a secondary criterion, e.g., the order of the values, to always get a defined result.
- **Mediating**: Similar to the concept of mediating strategies, a function is mediating, if it also allows other values as result than only the conflicting values, i.e., $f_{cr}(c_1, \ldots, c_n) = y$ with $y \in Y \supset \{c_1, \ldots, c_n\}$. Such a function (e.g., AVERAGE, CONCAT) may create a new value that is from the same domain as the conflicting values.
- **Monotonicity**: Monotonicity is defined on numerical domains as $\forall i = 1 \ldots n, x_i \neq \perp, y_i \neq \perp, x_i \leq y_i \Rightarrow f_{cr}(x_1, \ldots, x_n) \leq f_{cr}(y_1, \ldots, y_n)$. This property is only applicable if an order is defined on the domain. MIN, MAX, FIRST, and CONCAT are examples for monotonous functions.

Function	Description
COUNT	Counts the number of distinct NON-NULL values, i.e., the number of conflicting values. Only *indicates* conflicts, the actual data values are lost.
MIN/ MAX	Returns the minimal/maximal input value with its obvious meaning for numerical data. Lexicographical (or other) order is needed for non numerical data.
SUM/ AVERAGE/ MEDIAN	Compute sum, average, and median of all present NON-NULL data values.
VARIANCE/ STANDARD DEVIATION	Returns variance and standard deviation of data values.
RANDOM	Randomly chooses one data value among all NON-NULL data values.
CHOOSE(*source*)	Returns the value supplied by a specific source, may be a NULL value.
COALESCE	Returns the first NON-NULL value appearing.
FIRST/ LAST	Returns the first/last value, may be a NULL value.
VOTE	Returns the most frequent value among all NON-NULL values. Ties can be broken by a variety of strategies, e.g., by choosing randomly.
GLOBAL VOTE	Returns the most frequent value among all NON-NULL values in that attribute in the data source. Ties can be broken by a variety of strategies, e.g., by choosing randomly.
GROUP	Returns a set of all conflicting values.
SHORTEST/ LONGEST	Chooses the NON-NULL value of minimum/maximum length according to a length measure.
(ANNOTATED) CONCAT	Returns the concatenated values. May include annotations, such as the names of the data sources.
HIGHEST QUALITY	Returns the value of highest information quality. Requires an underlying quality model.
MOST RECENT	Returns the most recent value. Recency is evaluated with the help of another attribute or other metadata about tuples/values.
MOST ACTIVE	Returns the most often accessed or used value. Access statistics of the DBMS can be used in evaluating this function.
CHOOSE DEPENDING(*column,value*)	Chooses the value that belongs to a tuple with a specific given value v' in another column C'. C' and v' are given as parameters *column* and *value*.
CHOOSE CORRESPONDING(*column*)	Chooses the value that belongs to a tuple with the value already chosen for another column C'. C' is given as parameter *column*.
MOST COMPLETE	Returns the NON-NULL value of the source that contains the fewest NULL values in the attribute in question.
MOST DISTINGUISHING	Returns the value that is the most distinguishing among all present values in that column.
MOST GENERAL CONCEPT/ MOST SPECIFIC CONCEPT	Using a taxonomy or ontology this function returns the more general value (lowest common ancestor) or the more specific value (if the values are on a common path in the taxonomy).
IGNORE/ CONSTANT(c)	Returns a NULL value (IGNORE) or a constant value c of the respective domain (CONSTANT(c)).
LOOKUP(*source*)	Returns a value by doing a lookup into *source*, using the conflicting values.
COMMON BEGINNING/ COMMON ENDING	Returns the common substring at the beginning / end of all NON-NULL values.
TOKEN UNION/ TOKEN INTERSECTION	Tokenizes all NON-NULL values and returns the union / intersection of the sets of tokens. The token separator is given.

Table 3.1: Conflict resolution functions – partly from (Bleiholder and Naumann, 2005) – which can be used to implement conflict handling strategies.

- **Idempotence**: Idempotence ensures that functions can also cope with non-conflicting data, which means that $f_{cr}(c_i, \ldots, c_i)$ is always evaluated to c_i. Most functions are idempotent, exceptions include COUNT, IGNORE, and CONSTANT(c).
- **Boundary condition**: This property ensures that minimal/maximal values are always resolved to itself, i.e. $f_{cr}(b, \ldots, b) = b$ and $f_{cr}(t, \ldots, t) = t$ with b/t being the lower/upper boundary of the domain (bottom element b, top element t). It is a special case of idempotence, defined on the borders of the input domain, and used to distinguish aggregation functions in (Calvo et al., 2002).
- **Symmetry** (or **order sensitivity**): Parameters of a commutative binary function can be exchanged without influencing the result i.e. $f_{cr}(c_1, c_2) = f_{cr}(c_2, c_1)$. Symmetry extends commutativity to n-ary functions and shows if the function is sensitive to the order of the input values. A function is symmetric, if the order of the conflicting values does not change the result, i.e. if $f_{cr}(c_1, c_2, c_3) = f_{cr}(c_1, c_3, c_2) = \ldots = f_{cr}(c_3, c_2, c_1)$. COALESCE and FIRST are examples for non-symmetric functions whereas MAX or VOTE are among the symmetric functions.
- **Duplicate sensitivity**: Duplicate sensitivity indicates that the result is influenced by duplicate input values, i.e., $f_{cr}(c_1, c_1, c_2) \neq f_{cr}(c_1, c_2)$. COUNT, SUM, and VOTE are examples of duplicate sensitive functions, whereas MIN, LONGEST and MOST GENERAL CONCEPT are examples for insensitive ones. This property becomes important when considering the optimization of operator trees containing a *data fusion* operator together with *subsumption* and *distinct*.
- **Associativity** (or **decomposability**): Associativity extended to n-ary functions allows for the computation of partial results and their combination without changing the overall result. It is defined as $\forall n, m \in \mathbb{N}, \forall x_1, \ldots, x_n, y_1, \ldots, y_m : f_{cr}(x_1, \ldots, x_n, y_1, \ldots, y_m) = f_{cr}(f_{cr}(x_1, \ldots, x_n), f_{cr}(y_1, \ldots, y_m))$. Many functions are associative, exceptions are e.g., RANDOM and VOTE. Decomposability becomes important when moving *data fusion* operators in a query tree.
- **General decomposability** Extended variant of associativity/decomposability to allow the decomposition of a function into two different functions. It is defined as $\forall n, m \in \mathbb{N}, \forall x_1, \ldots, x_n, y_1, \ldots, y_m : f_{cr}(x_1, \ldots, x_n, y_1, \ldots, y_m) = f_{cr2}(f_{cr1}(x_1, \ldots, x_n), f_{cr1}(y_1, \ldots, y_m))$. As f_{cr1} and f_{cr2}, different functions are allowed. For example, the COUNT function is not associative but generally decomposable, with f_{cr1} being COUNT and f_{cr2} being SUM. This property also becomes important when considering the optimization of operator trees.
- **Neutral element**: A neutral element, if one exists, is a value from the input domain that can be omitted, because it has no impact on the result of the function, $f_{cr}(c_1, \ldots, c_n) = f_{cr}(c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n)$ with neutral element c_i. The neutral element of, e.g., SUM is 0, for COALESCE it is $\perp$.
- **Annihilator**: An annihilator a is a value that, if present in the input, determines the result, i.e., $f_{cr}(c_1, \ldots, c_i, \ldots, c_n) = c_i$ with annihilator c_i. In the literature, c_i is also called the veto-, or absorbing element. Annihilators exist with MAX/MIN (maximal/minimal value) and MOST GENERAL CONCEPT (root element of taxonomy).
- **NULL tolerance**: NULL tolerance describes how a function copes with NULL values in the input. A function is called null-tolerant, if NULL values in the input do not affect the final result, unless NULL is the only value in the input. It is null-intolerant if the result may change if NULL values in the input are added or removed. Example for null-tolerant functions are MAX and MIN, whereas COALESCE or FIRST are null-intolerant. NULL tolerance also becomes important when considering *data fusion* operators together with *subsumption* and *complementation* in optimizing operator trees.

In the literature, aggregation functions are commonly introduced with varying properties, e.g., (Detyniecki, 2001; Calvo et al., 2002; Li and Wu, 2004; Dubois and Prade, 2004). However, all approaches agree at least in the definition of aggregation functions on a numerical domain and by requiring aggregation functions to satisfy the properties of *idempotence*, *boundary condition*, and *monotonicity*. Aggregation combines values

Function	type[a]	applicable input domains[b]	mediating, deciding	monotonicity	boundary condition	idempotency	symmetry, order sensitivity	duplicate sensitivity	associativity (decomposability)	neutral element	annihilator	NULL tolerance	based on[c]	general decomposability
COUNT	S	A	M	+	–	–	+	+	–	–	–	+	D	+
MAX/ MIN	S	SN	D	+	+	+	+[e]	–	+	+[d]	+[d]	+	D	+
SUM	S	N	M	+	–	–	+	+	+	+	–	+	D	+
AVERAGE/ MEDIAN	S	N	M	+	+	+	+	+	–	–	–	+	D	–
VARIANCE/ STANDARD DEVIATION	S	N	M	–	–	–	+	+	–	–	–	+	D	–
RANDOM	S	A	D	–	+	+	–	+	–	–	–	+	D	–
CHOOSE(*source*)	SP	A	D	–	+	+	+	+	+	–	–	-	MD	+
COALESCE	S	A	D	–	+	+	–	+	+	+	–	+	D	+
FIRST/ LAST	S	A	D	+	+	+	–	+	+	–	–	-	D	+
VOTE	S	A	D	–	+	+	+[e]	+	–	–	–	+	D	–
GLOBAL VOTE	S	A	D	–	+	+	+[e]	+	–	–	–	+	MD	–
GROUP	S	A	×	×	×	×	×	×	×	×	×	–	D	×
SHORTEST/ LONGEST	S	SCT	D	–	+	+	+[e]	–	+	+[d]	+[d]	+	D	+
(ANNOTATED) CONCAT	S	A	M	+	–	–	–	+	+[f]	+[f]	–	–	D	+[f]
HIGHEST QUALITY	SP	A	D	–	+	+	+	+	+	–	–	–	MD	+
MOST RECENT	SMP	A	D	–	+	+	+	+	+	–	–	–	MD	+
MOST ACTIVE	SP	A	D	–	+	+	+	+	+	–	–	–	MD	+
CHOOSE DEPENDING	MP	A	D	–	+	+	+	+	+	–	–	–	D	+
CHOOSE CORRESPONDING[g]	MP	A	D	–	+	+	+	+	+	–	–	+/–	MD	+
MOST COMPLETE	SP	A	D	–	+	+	+	+	+	–	–	–	MD	+
MOST DISTINGUISHING	SP	A	D	–	+	+	+	+	+	–	–	–	MD	+
MOST GENERAL CONCEPT/ MOST SPECIFIC CONCEPT	SP	T	M/D	–	×	+	+	–	+	–	+	+	MD	+
IGNORE	S	A	M	+	+	–	+	–	+	+	–	+	×	+
CONSTANT(*c*)	SP	A	M	+	–	–	+	–	+	+	–	+	×	+
LOOKUP(*source*)	SP	A	M	–	–	–	–	+	–	–	–	–	MD	–
COMMON BEGINNING/ COMMON ENDING	S	S	M	–	+	+	+	–	+	–	+	+	D	+
TOKEN UNION/ TOKEN INTERSECTION	SP	S	M	–	+	+	+	–	+	+/–	–/+	+/–	D	+

[a] (S)ingle column, (M)ultiple column, with (P)arameter

[b] N(umeric), S(tring), C(ategorical), T(axonomical), D(ate), or A(ll)

[c] function uses only data (D) or also metadata (MD) in computing a result

[d] only if *top* and *bottom* elements exist

[e] depending on tie breaker

[f] not the annotated version

[g] Only in combination with a deciding function and in order to decide upon the properties, the properties of the other function also plays a role.

Table 3.2: Conflict handling functions and some of their properties, × meaning *not applicable*, a plus (+) marking *has the property* and a minus (–) *has not the property*.

Strategy	Implementing the strategy: possible functions or reference
Pass it on	Group, Concat
Consider all possibilities	(Burdick et al., 2005; Yan and Özsu, 1999)
Take the information	Coalesce, Longest
No gossiping	(Arenas et al., 1999; Fuxman et al., 2005a)
Trust your friends	Choose, Choose Depending, Highest quality, First, Most complete, Choose Corresponding
Cry with the wolves	Vote
Roll the dice	Random
Better bend than break	Average
Meet in the middle	Average, Median, Most general
Keep up to date	Most Recent, First
Ignorance is bliss	lookup(*source*), (Gigerenzer and Todd, 1999)
Oups I did it again	lookup(*source*), (Gigerenzer and Todd, 1999)
Better bend than break II	most general concept

Table 3.3: Strategies and functions that can be used to realize them.

of the same kind of different objects into a more dense representation. Conflict resolution functions on the other hand combine values of the same kind of same real-world objects. Although the goal is also a more dense representation, it has a different quality: it does not summarize values, it defines a value. Therefore, conflict resolutions function are less constrained in their domains and properties. Especially the *monotonicity* property does not hold frequently.

3.2.4 Choosing Strategies and Functions

Some strategies from Section 3.2.1 have a direct equivalent among the functions and can easily be realized by just applying this function to conflicting data. First to mention is the simple conflict ignoring strategy Pass it on, which is easily carried out by the functions group or concat. As already mentioned in Section 3.2.1, the coalesce function can be used to implement the Take the information strategy. Trust your friends is best illustrated by the choose(*source*) function, as source preference. This strategy is also a good example that there are different ways of realizing a strategy. Source preference can also be accomplished by using choose depending(*column,value*) with a column that contains a source identifier and the preferred source identifier as second parameter. Another possible choice is the highest quality function in combination with a quality measure that favors the preferred source. Further findings are summarized in Table 3.3. As can also be seen there, not all strategies can be implemented by applying a single function, e.g., No Gossiping. In these cases a reference into the literature is given, that describes the implementation of the strategy.

Choosing a specific conflict handling strategy for a certain problem is not an easy task, when one can choose among several alternatives. In most data integration scenarios, this choice is left to an expert user, as a high-level decision, which is then carried out by the integration system. The expert user needs to specify the strategy in some kind of formalism, e.g., as Sql query to be used by a system. She may also choose a strategy implicitly by using a certain integration system (if this systems only offers one strategy). The choice of a strategy highly depends on the domain and the task at hand and is driven by:

- System availability: Is there a system available that lets the user implement the strategy? Does the available system restrict the possible choices of strategies?
- Cost considerations: Is there enough money/time/disk space to use the strategy? Is a cheap result desired or is there enough budget (time, money), and is the user willing to pay for it?
- Quality considerations: Should the result contain as much information as possible or is just *some* result needed? Quality and cost often depend on each other, a high quality answer often being expensive. Is the user willing to spent some money in experts or some money and time? Among possible quality considerations are:

  - Correct results: Do we require that there are no new connections between data items in the sources (e.g., as not being guaranteed by the *match join*)?
  - Complete results: Does the task at hand require that as many information from the sources is also included in the result? Is the recall important?
  - Concise results: Do we prefer a concise result. i.e., is it acceptable if we omit some information from the sources in order to have less tuples in the result?

- Information availability: Is there enough information available to choose the strategy? For example, TRUST YOUR FRIENDS and KEEP UP TO DATE need some metadata.

- Expertise: Can an expert provide the necessary additional information that is needed by the strategy? For example, if a taxonomy is needed, is there one at hand?

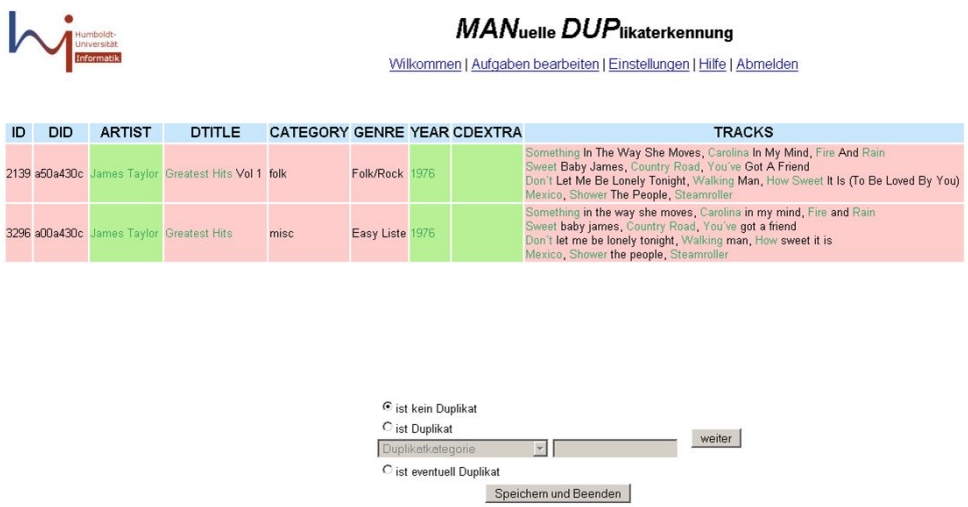
So far, choosing a strategy and implementing it has not been automated, but is mostly done manually by an expert user. Concerning the choice of different strategies/functions, in the following results of an experimental study are given that has been conducted within the information integration research group at Humboldt-Universität zu Berlin. This study can be seen as starting point towards automatically determining conflict resolution functions simply be observing user behavior when fusing data. It involves an audio CD scenario and has been conducted as part of the MANDUP project.

Function used	Number of usages	Percentage
FIRST	1387	28.9%
COALESCE	1173	24.4%
LAST	819	17.1%
CONCAT	463	9.6%
CHOOSE DEPENDING	348	7.2%
LONGEST	277	5.8%
MAX	223	4.6%
IGNORE	76	1.6%
AVERAGE	16	0.3%
MIN	9	0.2%
TOKEN UNION	6	0.1%
VOTE	5	0.1%
SHORTEST	2	0.04%
COUNT, COUNT ALL, STANDARD DEVIATION, SUM, VARIANCE, SHORTEST	0	0%

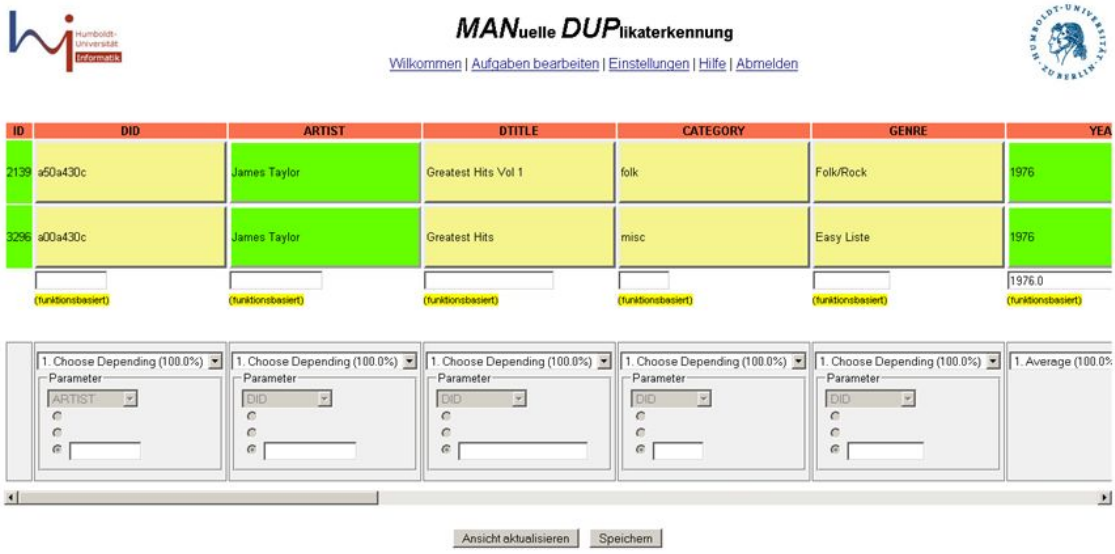
Table 3.4: Available conflict resolution functions and their usage in the experiment.

To better understand how functions are used in order to fuse different representations of same real-world objects the results of an experiment are presented that has been conducted in the context of a manual duplicate detection effort. The MANDUP system, developed at Humboldt-Universität zu Berlin, is a system that allows to manually classify pairs of tuples as different or same representations of a real-world object, an audio CD in our case. Figure 3.3 shows the two main screens of the system, where pairs of representations can be classified as duplicates/non-duplicates (see Figure 3.3(a)) and where conflicts are resolved (see Figure 3.3(b)). The system has been used to manually check a dataset of 10.000 audio-CD's. After approximately 64.000 single classifications of pairs (each pair had been checked by at least two people), 300 cases where different tuples represent the same real-world object (fuzzy duplicates) were found in the dataset. For each fuzzy duplicate, users were also asked to fuse the two representations into a single representation. Fusion was allowed per column and a total of 19 conflict resolution functions were available to the users. This resulted in approximately 600 tuple fusions, or approximately 4800 single fusion decisions. A list of the available functions and their usage in fusing the CD data is shown in Table 3.4.

The functions can be grouped into several groups according to their relative frequency: As we can see, the three most often used functions are FIRST, COALESCE, and LAST. As only two different representations



(a) Presentation of two audio CD representations and asking the user if the two representations represent the same real-world object. Pairs can be classified as duplicates, non-duplicates, or possible duplicates.



(b) Choosing a value by either clicking on the correct value (above), by entering a value (text field in the middle), or by choosing a function from a list of functions (below).

Figure 3.3: User interface of the ManDup system.

are fused in the scenario, this seems as the natural choice: to choose one of the two values available (first, last), avoiding the null value, if present (coalesce). These three functions are used in 70% of all fusion decisions. The next most often used group of functions consists of the functions concat, choose depending, longest, and max. These functions are easy to understand and try to preserve information, recent values and connections between attribute values in different attributes. ignore is the next most frequent function and can be used to ignore any value and include a null value in the result. This function can be used if all possible values seem to be incorrect. The last group of functions is less often used, including more complex and difficult to understand functions (token union), and functions that would be more suitable if more than only two representations are fused (vote, average). The functions that were available but were rarely used (count, count all, standard deviation, variance, sum, and shortest) do not fit the scenario given and are ignored in the following analysis.

Figure 3.4 shows the usage of functions among the 11 distinct people that used the system. The first column contains the overall usage, each following column contains the distribution of usage for each user. The

Function	#Usage	%Usage	user 1	user 2	user 3	user 4	user 5	user 6	user 7	user 8	user 9	user 10
Average	16	0,33%	0,0%	1,8%	1,1%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
Choose Depending	348	7,24%	1,0%	3,6%	7,5%	0,6%	1,5%	16,0%	30,0%	5,2%	60,0%	14,0%
Coalesce	1173	24,42%	27,9%	8,9%	38,9%	58,8%	0,5%	0,0%	0,0%	15,0%	0,0%	36,0%
Concat	463	9,64%	7,7%	7,1%	10,8%	7,6%	11,3%	2,1%	0,0%	8,9%	0,0%	13,8%
First	1387	28,87%	24,5%	51,8%	10,1%	15,4%	54,6%	36,1%	33,8%	46,0%	3,8%	7,5%
Ignore	76	1,58%	0,0%	1,8%	0,0%	0,0%	7,7%	4,9%	1,3%	0,2%	6,3%	0,0%
Last	819	17,05%	16,8%	21,4%	8,3%	17,4%	24,1%	38,2%	35,0%	23,5%	1,3%	6,7%
Longest	277	5,77%	5,8%	3,6%	10,3%	0,3%	0,0%	2,8%	0,0%	1,2%	21,3%	17,5%
Max	223	4,64%	16,3%	0,0%	12,4%	0,0%	0,3%	0,0%	0,0%	0,0%	1,3%	3,1%
Min	9	0,19%	0,0%	0,0%	0,1%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	1,5%
Shortest	2	0,04%	0,0%	0,0%	0,1%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
TokenUnion	6	0,12%	0,0%	0,0%	0,4%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
Vote	5	0,10%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	6,3%	0,0%

Figure 3.4: Result of an experiment with users: Usage of conflict resolution functions by user.

frequency of use is color coded, from yellow (0%) to dark green (100%). We report the following observations: Although COALESCE gives the same result as FIRST if there are two distinct NON-NULL values, Users 3, 4, and 10 clearly prefer COALESCE over FIRST in contrast to all other users. The same users also use LAST less often than the other users. User 1 seems to be the average user as she uses FIRST, LAST, and COALESCE more or less with the same frequency than all users do on average. In contrast, all other users rarely use COALESCE and stick with FIRST and LAST. The full variety of possible functions is rarely used, only by User 3 (11 out of 14 functions). All other users used at most 8 out of 14 functions, some functions only very few times. Unusual is also the high percentage of the use of the CHOOSE DEPENDING(*column,value*) function for User 9. Overall, usage of conflict handling functions differs by users, with no clear trend. However, most users tend towards deciding functions and thereby choosing a specific value.

Function	#Usage	%Usage	ID	Artist	Album	Category	Genre	Year	CDExtra	Tracks
Average	16	0,3%	0,0%	0,0%	0,0%	0,0%	0,5%	2,2%	0,0%	0,0%
Choose Depending	348	7,2%	17,1%	6,5%	5,6%	6,0%	4,1%	4,3%	9,8%	4,5%
Coalesce	1173	24,4%	42,2%	34,8%	24,4%	25,0%	15,1%	21,1%	22,6%	9,9%
Concat	463	9,6%	13,3%	0,2%	1,8%	32,7%	22,4%	1,7%	4,6%	0,3%
First	1387	28,9%	19,9%	42,3%	41,5%	26,5%	22,9%	25,7%	15,9%	36,1%
Ignore	76	1,6%	0,0%	0,0%	0,0%	0,0%	1,7%	2,8%	8,3%	0,0%
Last	819	17,0%	7,0%	14,4%	21,6%	9,5%	18,6%	15,6%	15,0%	35,0%
Longest	277	5,8%	0,0%	1,5%	3,3%	0,3%	10,8%	1,8%	17,1%	11,6%
Max	223	4,6%	0,0%	0,3%	0,7%	0,0%	4,0%	24,9%	6,8%	0,5%
Min	9	0,2%	0,2%	0,0%	1,2%	0,0%	0,0%	0,0%	0,0%	0,2%
Shortest	2	0,0%	0,3%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
TokenUnion	6	0,1%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	1,0%
Vote	5	0,1%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,8%

Figure 3.5: Results of an experiment with users: Usage of conflict resolution functions by attribute.

Figure 3.5 shows the usage of functions by the 8 different columns in the dataset. The dataset consists of CD data with the following 8 attributes: ID (a CD identifier, basically a hash on the CD content), ARTIST (the name of the artist), ALBUM (title of the album), CATEGORY (CD categorization), GENRE (genre, such as *Rock*, *Pop*, *Classical*), YEAR (year of release), CDEXTRA (additional information), and TRACKS (track titles of the CD). In column ID, the majority of decisions are decisions for a specific value, either the value of the first or the second tuple. This is reflected in the high number of usages for FIRST, COALESCE and LAST. The same holds for columns ARTIST and ALBUM. The function MAX is unusually often used for the YEAR attribute, in contrast

to the other attributes. An explanation could be that YEAR is the only non-string attribute in the data set and that most users have a clear idea of how MAX behaves with numbers. It is a more natural choice for numbers than for strings. The underlying decision is the decision for the newer release date of the audio CD. Another irregularity is the LONGEST function, which is often used with the attributes GENRE, CDEXTRA, and TRACKS. In all three cases the decision is reasonable, as in all three cases, the usage of LONGEST preserves the potentially more informative value. Attributes CATEGORY and GENRE are the only attributes where CONCAT is used in a significant number of cases, indicating that simply keeping the information stored by one representation did not seem to be sufficient, so that both values needed to be kept. Additionally these attributes are set-valued. Overall, usage of conflict resolution functions also varies with the type of data that is stored in a column.

The system also recorded a reason for each duplicate (why is this a duplicate?), given by the user that found the duplicate. Among the different duplicate reasons are: *identity (no error)*, *subsumption*, *missing data*, *typos*, *lower/uppercase problems*, etc. We also checked for differences in usage of functions by different duplicate reasons. It turned out that there are no significant differences in the usage of functions.

Another type of information that was stored while collecting fusion decisions is information on computability. For every available function f_j we recorded if the chosen resolved attribute value v (by the user) for the given conflicting values c_i could have been computed by the function f_j, i.e., if $f_j(c_1, \ldots, c_n) = v$. Afterwards, we were able to compute the success rate, the percentage of overall successful fusions if the function would be used for all fusions in that attribute. Figure 3.6 shows all combinations of functions and attributes. It could be seen that the simple heuristic to just use the function that has been used most often by the users is not necessarily the best one. For example, looking at the YEAR attribute, the most often used functions among all users was the FIRST function (see Figure 3.5). However, it computes the correct value only in 66,8% of all cases. Using MAX results in a success rate of 84,3% and would be the better choice for this column.

Function	#Comp.	ID	Artist	Album	Category	Genre	Year	CDExtra	Tracks
Average	599	0,0%	0,0%	0,0%	0,0%	0,0%	2,2%	0,0%	0,0%
Choose Depending	599	0,7%	0,5%	0,2%	0,5%	0,3%	0,3%	0,3%	0,8%
Coalesce	599	**62,6%**	81,5%	**72,6%**	52,4%	47,2%	66,8%	53,4%	55,6%
Concat	599	13,4%	0,2%	1,7%	32,7%	22,5%	1,7%	4,5%	0,3%
First	599	**62,6%**	81,5%	**72,6%**	52,4%	47,2%	66,8%	53,4%	55,6%
Ignore	599	16,0%	5,5%	5,0%	5,0%	5,5%	6,3%	17,0%	4,7%
Last	599	16,7%	80,3%	67,4%	55,4%	50,1%	66,1%	49,9%	52,1%
Longest	599	60,1%	**81,6%**	71,1%	51,3%	57,9%	78,5%	64,1%	57,9%
Max	599	26,0%	80,5%	68,4%	**57,1%**	**63,8%**	**84,3%**	**67,6%**	45,9%
Min	599	53,3%	81,3%	71,6%	50,8%	33,6%	48,6%	35,7%	**61,8%**
Shortest	*364*	*61,0%*	*79,7%*	*72,8%*	*53,6%*	*36,3%*	*47,0%*	*28,3%*	*46,7%*
TokenUnion	*358*	*11,7%*	*4,7%*	*4,2%*	*3,4%*	*5,9%*	*8,4%*	*20,9%*	*5,3%*
Vote	599	34,1%	79,0%	71,3%	51,6%	62,8%	83,5%	67,4%	51,3%

Figure 3.6: Success rates for all combinations of functions and attributes.

Applying the simple heuristics of using the function with the highest success rate for each column (COALESCE or FIRST, LONGEST, COALESCE or FIRST, MAX, MAX, MAX, MAX, MIN, marked in bold in Figure 3.6) results in an overall success rate of 68,9%. In contrast, using the most frequently used functions (COALESCE, FIRST, FIRST, CONCAT, FIRST, FIRST, COALESCE, FIRST, see Figure 3.5) results in only 59,1% correct decisions. Overall, recording user decisions in fusing data and extrapolating a choice of functions for each attribute does not result in the best possible result. A simple idea to improve on the simple heuristics is the following: if we look at combinations of functions, we can consider the optimization problem of finding the best combination of functions. In addition we can consider different partitionings of the relation (e.g., using function f for the first half, function g for the second half), or functions based on attribute values (e.g., using function f for "Rock" CD's and function g for "Classical" CD's).

The limits of my language mean the limits of my world.

(Ludwig Wittgenstein)

4

Expressing Data Fusion

So far, we presented conflict handling strategies to handle conflicting information in general and also introduced the more specific concept of conflict handling functions to resolve conflicts within a group of conflicting values. As a next step we want to bridge the abstraction gap between these two concepts and define operators that can be used to implement one or more of the strategies by using conflict handling functions.

We first formally introduce the reader into the setting of relational databases as used throughout this thesis and then define three new operators for use within the relational algebra. Two of these operators implement two different flavors of the TAKE THE INFORMATION strategy, and the third operator is a versatile operator that employs conflict handling functions to implement many of the conflict avoiding and resolution strategies mentioned in the last chapter. After defining the operators we introduce a set of keywords to extend SQL with the three operators.

4.1 Basic Definitions and Notation

In the following we assume a relational data model as introduced by (Codd, 1979). As there are some slight differences between descriptions and notations of the relational model in the literature we basically follow (Ullman et al., 2001) and (Abiteboul et al., 1995). In particular, we adopt what is specified by (Abiteboul et al., 1995) as the *named* perspective. We also employ the algebraic perspective instead of the logic-basic approach to accessing data stored in the relational model. Notation for operators largely follows (Ullman et al., 2001).

Data in the relational model is stored in the form of relations, where one relation can be visualized as a table, as seen in the following table, a slightly changed version of the POLICE relation from Section 1.2.

POLICE:

Name	Birthdate	Sex	Address
Miller	7/7/1959	m	234 Main St.
Miller	⊥	⊥	234 Main St.
Peters	1/19/1953	m	43 First St.
Smith	8/9/1970	m	Mass Ave.
Smith	8/9/1970	m	Mass Ave.

Definition 11 (Schema)
Let $A = \{a_1, a_2, \ldots, a_m\}$ be a set of m attributes. Every attribute has an assigned domain *(given by a function $dom(a_i)$), such as integer, string, date, etc. The set of attributes and its assigned domains together with the relation name define the* schema *$S = (name, A, dom)$ of a relation.* □

In our example the schema S of the relation above has the name POLICE and the set of attributes $A = \{Name, Birthdate, Sex, Address\}$ with assigned domains $dom(Name) = string$, $dom(Sex) = \{m, f\}$, $dom(Birthdate) = date$, $dom(Address) = string$. So in our example $S = (Police, \{Name, Birthdate, Sex, Address\}, dom)$. The set of attributes from S form the column headers of the table. A short notation for the set of attributes and its domains is $Name(string)$, $Birthdate(date)$, etc.

Definition 12 (Tuple)
A tuple *t is a set of m attribute/value combinations, $t = \{(a_1, v_1), (a_2, v_2), \ldots, (a_m, v_m)\}$. A* tuple *adheres to a schema S, so the m attributes a_j form the set A and the values v_j are of the corresponding* domains *$dom(a_j)$.* □

Tuples are the rows of the table, so for example the first line below the header from the table above forms the tuple $t_1 = \{(Name, \text{Miller}), (Birthdate, 7/7/1959), (Sex, \text{m}), (Address, 234 \text{ Main St.})\}$. We use the shorthand (Miller, 7/7/1959, m, 234 Main St.) (omitting the attribute names) if the order of the attributes is clear from the context. We allow for missing information in the tuples, i.e., attributes having no value. Missing information is modeled by introducing a special value, the NULL value ($\perp$), which is not part of any domain. So in fact, values v_j could be from $dom(a_j) \cup \{\perp\}$. As we will see further on it is useful to also implicitly model missing information instead of doing so explicitly by using the special value $\perp$. We implicitly model missing information in tuples by missing attribute/value combinations, i.e., if v_j is NULL, then there is no attribute/value pair (a_j, v_j) in t. If not clear from the context, we use $\alpha(t)$ to denote the implicit modeling of missing information. For example, the second tuple from the relation above is explicitly represented as $t_2 = \{(Name, \text{Miller}), (Birthdate, \perp), (Sex, \perp), (Address, 234 \text{ Main St.})\}$ and implicitly represented as $\alpha(t_2) = \{(Name, \text{Miller}), (Address, 234 \text{ Main St.})\}$.

Definition 13 (Relation)
A relation *$R = (S, T)$ of size n is defined by a schema S and a set $T = \{t_1, t_2, \ldots, t_n\}$ of n tuples, where each tuple t_i adheres to schema S. When the schema and the set of tuples are clear from the context, we denote relations only by R.* □

We reference the above relation by its name POLICE and give the relation as a table as depicted at the beginning of this section. Under bag semantics, we further allow exact duplicates among the tuples in T. Two tuples t and t' are exactly equal, denoted by $t = t'$, if they conform to the same schema and if their attribute/value sets are equal, i.e. if $\forall i, j, (a_i, v_i) \in t, (a_j, v_j) \in t' : a_i = a_j \rightarrow v_i = v_j$. Under set semantics, exact duplicates in T do not exist.

Definition 14 (Database)
A database *DB consists of k relations: $DB = \{R_1, \ldots, R_k\}$.* □

We rely on the standard operators present in the relational algebra (Abiteboul et al., 1995; Ullman et al., 2001) and assume the reader to be familiar with these basic relational operators. In particular we use *select-project-join-rename* queries as basic building blocks and additionally *union*, *grouping/aggregation*, *cartesian product*, and *sort*. As additional join variants we consider *left-*, *right-*. and *full outer join*. Under bag semantics, the *distinct* operator removes exact duplicates. Table 4.1 lists the basic relational operators and its symbols used throughout this thesis.

We further assume that we know which attributes in same or different relations are semantically equivalent (given by a schema mapping). For simplicity, semantically equivalent attributes have the same name. Concerning object identity we first assume a world of distinct real-world objects. If necessary and not already given, we employ duplicate detection methods to assign a global identifier to each distinct real-world object. Such an identifier can then be used in data fusion.

Based on these definitions, assumptions and basic notation, in the following sections, we formally define three new operators and give examples of their behavior. Note that examples on how these operators behave are also given in Section 1.2. The first two operators assume an "unknown" semantics of NULL values, as both aim at filling up NULL values with existing data. The third also assumes "unknown" semantics, but can also cope with other semantics, the only restricting factor here being the conflict resolution functions used.

4.2 Subsumption and Minimum Union

Minimum union is one of two alternative operators that both aim at resolving a special type of data conflict, called uncertainty (see Section 3.1.1 on page 30), realizing the TAKE THE INFORMATION strategy. *Minimum union* combines an *outer union* with the removal of tuples that are contained in other tuples, so called *subsumed*

Operator	Notation	Description
selection	σ_c (R)	Selects tuples from R that fulfill condition c
projection/renaming	π_L (R)	Extended projection as in (Ullman et al., 2001), including renaming and basic computations, given in L and applied on relation R
sort	τ_o (R)	Sorts the tuples in a specified order o
distinct	δ (R)	Removes exact duplicates from relation R
grouping/aggregation	γ_L (R)	Grouping and aggregation as given in attribute list L and applied on relation R
natural join	R $\bowtie$ S	Joins two relations R and S on attributes with same names
join	R $\bowtie_c$ S	Joins two relations R and S using join condition c
cartesian product	R $\times$ S	Computes the cartesian product (cross product) of two relations R and S
left outer join	R $\mathbin{⟕}_c$ S	Computes a left outer join of R and S using join condition c
right outer join	R $\mathbin{⟖}_c$ S	Computes a right outer join of R and S using join condition c
full outer join	R $\mathbin{⟗}_c$ S	Computes a full outer join of R and S using join condition c
union	R $\cup$ S	Union of relations R and S

Table 4.1: Notation used for relational operators, based on the notation used in (Ullman et al., 2001)

tuples (Galindo-Legaria, 1994). We define the operator as follows, first introducing *outer union* and *subsumption* as its building blocks:

Definition 15 (Outer Union)
The binary outer union *operator ($\uplus$) combines two relations and is an extension of the* union *operator to tables with arbitrary schemata. The output schema is the union of the input schemata, where duplicate attributes are removed according to given equivalences (same name, if nothing else is given). Likewise, the resulting tuple set is the union of all input tuple sets. Given two relations R and S and implicit modeling of missing information, then the* outer union *$T = R \uplus S$ is given by*

$$\begin{aligned} T &= (S_R \cup S_S, T_R \cup T_S) \\ &= ((newname, \{a_i^r\} \cup \{a_j^s\}, \{dom(a_i^r)\} \cup \{dom(a_j^s)\}), \{\alpha(t_i^r)\} \cup \{\alpha(t_j^s)\}) \end{aligned}$$

□

The two relations R and S with attribute sets $A_R = \{A, B, C, D\}$ and $A_S = \{C, D, E, F\}$ result in $T = (R \uplus S)$ having attribute set $A_T = \{A, B, C, D, E, F\}$. Combining more than two relations is easily possible as *outer union* is commutative. If we employ explicit modeling of missing information, tuples must be padded by NULL values in the attributes that are contained in the attribute set of the respective other relation.

Definition 16 (Tuple Subsumption (Galindo-Legaria, 1994))
A tuple $t_1 \in T$ subsumes *another tuple $t_2 \in T$ ($t_1 \sqsupset t_2$), if (1) t_1 and t_2 have the same schema, (2) t_2 contains more* NULL *values than t_1, and (3) t_2 coincides in all* NON-NULL *attribute values with t_1.* □

Under implicit modeling of missing information, tuple subsumption is equal to set containment. A tuple t_1 *subsumes* another tuple t_2 ($t_1 \sqsupset t_2$), if $t_1 \supset t_2$. Tuple subsumption is a transitive relationship, so if $t_1 \sqsupset t_2$ and $t_2 \sqsupset t_3$, then also $t_1 \sqsupset t_3$. Tuple subsumption is neither symmetric nor reflexive.

Definition 17 (Subsumption Operator)
We use the unary subsumption operator β to denote the removal of subsumed tuples from a relation R:

$$\beta(R) = \left\{ t \in R \mid \neg \exists t' \in R : t' \sqsupset t \right\}$$

□

Note that equal tuples (exact duplicates) do not subsume each other and are therefore not removed by β. Under set semantics, exact duplicates do not exist anyway. The extension to bag semantics is straightforward:

Two tuples t and t' with $t = t'$ are both removed by β, if they are subsumed by a third tuple t''. If a tuple t is subsumed by two tuples t' and t'' with $t' = t''$, the tuple t is removed by β. Under bag semantics, the *distinct* operator (δ) can additionally be used to explicitly remove exact duplicates, either before or after removing subsumed tuples. The following example shows the removal of subsumed tuples under bag semantics:

Relation R

tuple	**A**	**B**	**C**	**D**
1	a	b	c	d
2	a	⊥	c	d
3	b	b	⊥	d
4	b	⊥	⊥	d
5	b	⊥	⊥	d

Relation β **(R)**

tuple	**A**	**B**	**C**	**D**
1	a	b	c	d
3	b	b	⊥	d

In the example, Tuple 2 is removed, because it is subsumed by Tuple 1 and Tuples 4 and 5 are subsumed because they are subsumed by Tuple 3. The remaining tuples are Tuples 1 and 3.

Definition 18 (Minimum Union)
The binary minimum union *operator ($\oplus$) is the combination of* outer union *and* subsumption *where subsumed tuples are removed from the result of the* outer union *of the two input relations:*

$$A \oplus B = \beta\left(A \uplus B\right)$$

□

The *minimum union* operator is commutative and associative. The following example shows tuples from two tables that are combined by *outer union* and where subsumed tuples are removed subsequently:

Relation R

tuple	**A**	**B**	**C**	**D**
1	a	b	c	d
2	a	⊥	c	d
3	a	b	⊥	d

Relation S

tuple	**C**	**D**	**E**	**F**
4	c	⊥	⊥	⊥
5	c	d	⊥	⊥
6	d	⊥	e	f

Relation $R \oplus S$

tuple	**A**	**B**	**C**	**D**	**E**	**F**
1	a	b	c	d	⊥	⊥
6	⊥	⊥	d	⊥	e	f

Please note that we allow for both subsumed tuples in the same source (e.g., Tuple 2, 3, 4) and subsumed tuples among sources (e.g., Tuple 4, 5).

4.3 Complementation and Complement Union

Complement union is another operator that aims at resolving a special type of data conflict, called uncertainty (see Section 3.1.1), thus realizing the Take the information strategy. It combines an *outer union* with the combination of tuples that complement each other. We define the operator as follows, first introducing *complementation* and the concept of maximal complementing sets as its building blocks:

Definition 19 (Tuple Complementation)
A tuple t_1 complements a tuple t_2 ($t_1 \gtrless t_2$) if (1) t_1 and t_2 comply to the same schema, (2) values of corresponding attributes in t_1 and t_2 are equal or one of them is null *, (3) t_1 and t_2 are neither equal nor do they subsume one another, and (4) t_1 and t_2 have at least one attribute value combination in common.* □

We introduce condition (3) to strictly separate equality, subsumption and complementation, and introduce condition (4) to assure that tuples are not completely unrelated. When assuming implicit modeling of missing information, two tuples t_1 and t_2 complement each other, if for the set $t_c = t_1 \cup t_2$ of all (a_j, v_j) from both tuples it holds that $\forall (a_l, v_l), (a_m, v_m) \in t_c : a_l = a_m \rightarrow l = m \wedge t_c \supset t_1 \wedge t_c \supset t_2$ (assuring at most one value

per attribute, or a NULL value). The set t_c is called the *complement* of the two tuples. Tuple t_c is created by coalescing the values of each attribute a_i from t_1 and t_2. Both t_1 and t_2 are from the same T, assuring schema compliance. Similarly, we can construct the complement out of three or more tuples, if all three or more tuples pairwise complement each other. Tuple complementation is a symmetric relationship (if $t_1 \gtrless t_2$, then $t_2 \gtrless t_1$). It is not reflexive and in contrast to subsumption, tuple complementation is not transitive.

In order to define the complementation operator we first introduce maximal complementing sets:

Definition 20 (Maximal Complementing Set)
A complementing set CS of tuples of a relation $R = (S, T)$ is a subset of tuples from T where for each pair t_i, t_j of tuples from CS it holds that $t_i \gtrless t_j$. A complementing set S_1 is a maximal complementing set *(MCS) if there exist no other complementing set S_2 s.t. $S_1 \subset S_2$.* □

A tuple t_i can be in more than one *MCS* (e.g., if t_i complements t_j and also t_k, but t_j does not complement t_k), and in general there are multiple *MCS* for a relation R. A relation R can be uniquely divided into a set of *MCS*s. All tuples of a maximal complementing set MCS_i can be combined into one tuple, the complement $t_c i$.

Definition 21 (Complementation Operator)
The unary complementation *operator κ replaces each existing maximal complementing set MCS_i of a relation R by the single complement tuple t_{c_i} of all tuples in MCS_i.* □

Note that equal tuples (exact duplicates) do not complement each other and are therefore retained after application of κ. Under set semantics, exact duplicates do not exist anyway. Under bag semantics, exact duplicates do exist. The extension of the *complementation* operator to bag semantics is again straightforward: Given the case that two duplicate tuples t_i and t_i' with $t_i = t_i'$ both complement t_j. As t_i and t_i' do not complement each other, this results in two maximal complementing sets ($\{t_i, t_j\}$ and $\{t_i', t_j\}$). However, because $t_i = t_i'$, both maximal complementing sets finally result in the same complement. Under bag semantics, the *distinct* operator (δ) can additionally be used to explicitly remove exact duplicates.

The following examples demonstrates the application of *complementation*:

Relation *R*

tuple	**A**	**B**	**C**	**D**
1	a	b	c	d
2	a	b	c	⊥
3	a	⊥	c	d
4	a	b	⊥	d
5	a	⊥	⊥	e
6	b	⊥	⊥	d

Relation κ **(R)**

tuple	**A**	**B**	**C**	**D**
1	a	b	c	d
2+3+4	a	b	c	d
2+5	a	b	c	e
6	b	⊥	⊥	d

In the example, Tuples 2, 3, and 4 form a maximal complementing set, as well as Tuples 2 and 5. They result in their complement, two different tuples. Tuples 1 and 6 do not complement any other tuple and are therefore retained as is. Whereas the order of β and δ can be exchanged, assuming bag semantics, the order of κ and δ is relevant. This is best shown in the example above (righthand side, κ (R)) when looking at the first two tuples, which are the same. This is the result of all tuples being part of the set (2, 3, and 4) being subsumed by Tuple 1. So, considering bag semantics, to solve all issues related to *complementation* and *subsumption* and additionally remove exact duplicates, κ needs to be applied before β and δ.

Definition 22 (Complement Union)
The binary complement union *operator ($\boxplus$) is the combination of* outer union *and* complementation. *Complementing tuples in the result of the* outer union *of the two input relations A and B are combined:*

$$A \boxplus B = \kappa\left(A \uplus B\right)$$

□

Complement union is commutative, but not associative. The following example shows tuples from two tables that are combined by *outer union* and where complementing tuples are combined:

Relation R

tuple	**A**	**B**	**C**	**D**
1	a	⊥	c	d
2	⊥	b	c	⊥
3	⊥	c	c	d

Relation S

tuple	**C**	**D**	**E**	**F**
4	c	e	⊥	f
5	c	⊥	⊥	g
6	c	e	e	⊥

Relation $R \boxplus S$

tuple	**A**	**B**	**C**	**D**	**E**	**F**
1+2+6	a	b	c	d	⊥	g
1+3+6	a	c	c	d	⊥	g
2+4+6	⊥	b	c	e	e	f
2+5+6	⊥	b	c	e	e	g

Similar to *subsumption*, the definition forms complements of complementary tuples from the same source (Tuples 1 and 2 from the example) and from different sources (Tuples 2 and 6 from the example).

4.4 Conflict Resolution and Data Fusion

Next, we define a versatile operator (the *data fusion* operator) that is designed to also solve inconsistencies and not only tackle uncertainties such as the two operators defined in the previous sections. The operator also includes *subsumption* and *minimum union*, i.e., in its simplest form, it performs the removal of subsumed tuples or performs a *minimum union*. The relationship is comparable to the relationship between *distinct* (δ) and *aggregation* (γ) (Ullman et al., 2001). However, the *data fusion* operator goes beyond *subsumption* and *minimum union* and is also able to resolve inconsistencies. In order to define the operator itself, we first define the *conflict resolution* operator as one building block:

Definition 23 (Conflict Resolution Operator)
The conflict resolution *operator is denoted* $\lambda_{CR,O}\ (T)$ *and is defined as the application of a set of m conflict resolution functions* $CR = \{cr_1, cr_2, \ldots, cr_m\}$ *on an ordered set of n tuples* $T = \{t_1, t_2, \ldots, t_n\}$ *that comply to an input schema* $IS = (name^I, A^I, dom^I)$ *with l attributes* $A^I = \{a_1^I, \ldots, a_l^I\}$. *The result of* conflict resolution *is one single tuple t that complies to an output schema* $OS = (name^O, A^O, dom^O)$, *with m attributes* $A^O \subseteq A^I$. *Conflicts are resolved attribute-wise, i.e., for each* cr_i *there is a corresponding* $a_i^O \in A^O$. *Attribute* a_i^O *is produced by* cr_i. *Given the ordered set of tuples T, for each attribute* $a_i^I \in A^I$, *an ordered set of values* V_i *is defined, s.t.,* $V_i = \bigcup_{j=0}^{n} v_i$ *with* $(a_i, v_i) \in t_j, t_j \in T$. *The order of the tuples in T is given by a sort order O, a set of k attributes* $O = \{a_1, \ldots, a_k\} \subseteq A^I$, *that are used to order tuples in each* V_i. *The result of* conflict resolution *is then the result of applying all conflict resolution functions in CR on the sets* V_i. *Thus, the final result of* conflict resolution *is a single tuple t, s.t.,* $t = (cr_1(V_{j_1}), cr_2(V_{j_2}), \ldots, cr_m(V_{j_m})) = \{(a_1^O, cr_1(V_{j_1})), (a_2^O, cr_2(V_{j_2})), \ldots, (a_m^O, cr_m(V_{j_m}))\}$ *with* $V_{j_k}, j_k = 1, \ldots, n$ *being the needed input attribute for function* cr_k. *In case of multi-column conflict resolution functions, all necessary* V_{j_k} *are use, e.g.,* $cr_1(V_{j_1}, V_{j_2})$. □

Conflict resolution on a tuple set T can be seen as combining a *sort* on that tuple set T and an order observing *aggregation*, where not only the standard aggregation functions are allowed (see Section 3.2.2 on page 36 for a list of possible function). Requiring an ordered set T specifically allows for influencing conflict resolution functions that are symmetric (see symmetry/order sensitivity in Section 3.2.3 on page 37). In case of functions with additional input (metadata conflict resolution functions), additional parameters to the conflict resolution functions are first passed to the operator and then to the respective functions.

We now define two variants of the *data fusion* operator:

Definition 24 (Data Fusion Operator without Removal of Subsumed Tuples)
The unary data fusion *operator* $\phi_{F,CR,O}\ (R)$ *on a relation R with input schema* $IS = (R, A^I, dom^I)$ *is defined as follows: As parameters, it uses a set* $F = \{f_1, \ldots, f_k\} \subseteq A^I$ *of k identifying attributes that are used to identify an object, a set* $CR = \{cr_1, \ldots, cr_l\}$ *of l conflict resolution functions to create l corresponding attributes* $CA = \{ca_1, \ldots, ca_l\} \subseteq A^I$, *and a sort order O, a set of m attributes* $O = \{a_1, \ldots, a_m\} \subseteq A^I$ *by which tuples are ordered. Conflicts are resolved attribute-wise, i.e., there is one corresponding* ca_i *for each* cr_i, *where the conflict resolution function* cr_i *produces the final result for attribute* ca_i. *The input schema of the operator is the schema*

of the relation R. F and CA are disjoint, and $CA \cup F \subseteq A^I$. *The result schema of the operator consists of all the attributes from F and CA. We first define a decomposition of R, a set of p tuple sets* $\mathcal{T} = \{T_1, \ldots, T_p\}$, *where each* T_i *is defined as the set of tuples that have the same attribute values of attributes in F. If* $F = \emptyset$, *then all tuples form one large group. Then, under bag semantics, exact duplicates are removed from each* T_i. *Finally, on each* T_i, *a* conflict resolution *operator is applied, thus, for each* T_i, *we define a result tuple* t_i. *The final result of the* data fusion *operator is then the union of all result tuples* t_i, *where each* t_i *is padded with the values for the attributes from F, i.e.,* $T = \bigcup t_i$, *with* $t_i = \lambda_{CR,O}\ (T_i) \cup \bigcup_k \{(a_k, v_k)\}, a_k \in F$. □

The second variant of the *data fusion* operator includes the removal of subsumed tuples:

Definition 25 (Data Fusion Operator with Removal of Subsumed Tuples)
The unary data fusion *operator with removal of subsumed tuples* $\phi^{\beta}_{F,CR,S}\ (R)$ *on a relation R is defined as the application of a* data fusion *operator* $\phi_{F,CR,S}\ (R)$ *on a relation R, where additionally subsumed tuples are removed from the* T_i, *i.e., a result tuple* t_i *is defined as* $t_i = \lambda_{CR,O}\ (\beta\ (T_i)) \cup \bigcup_k \{(a_k, v_k)\}, a_k \in F$. □

The semantics of the *data fusion* operator is illustrated as follows: First, the source relation is divided into groups of tuples using the attributes in F to identify groups. Second, only in case of ϕ^{β}, subsumed tuples are removed from each group. Then, a *conflict resolution* is executed on each group separately. For this, the parameters CR and S are directly fed into the conflict resolution together with all tuples of a group.

The *data fusion* operator is defined as a unary operator. So, in order to fuse data from multiple source relations, all tuples from all source relations involved need to be combined prior to that, in order to form a single relation. Here, for data integration, an *outer union* can be used, as well as a *join* or any other suitable operator.

We now give more details on the single steps of the *data fusion* operator. Executing a fusion consists of two phases: First, of preparing for resolving conflicts by grouping tuples representing the same real-world object, and second, of resolving conflicts to come up with a single consistent representation for each real-world object.
Step 1: Preparing for conflict resolution. First, all tuples that describe the same real-world object are grouped together. This is done by doing a grouping on the column(s) given in the list of identifying attributes $F = \{f_1, \ldots, f_k\}$. We hereby assume that we are able to rely on a globally unique and consistent identifier that we can use to do the grouping. This identifier may be produced by detecting duplicates and assigning equal keys to the same real-world objects or using multiple columns as key. For this reason, in the data integration process, duplicate detection needs to be done prior to the fusion process.
Step 2: Increasing conciseness and resolving conflicts. Then, in case of the second variant of the *data fusion* operator, subsumed tuples are removed per group. In case of the first variant, subsumed tuples are kept. Interestingly, removing subsumed tuples per group as needed in our case does not yield the same result as removing subsumed tuples from the entire table. This is due to possible NULL values in the grouping attributes and is described in more detail in Section 6.2.3. All remaining tuples of one group are then fused together into only one single tuple, at the same time resolving inconsistencies and data conflicts. This is done by applying *conflict resolution* on the tuple set, by feeding the remaining parameters, the list of conflict resolution functions and the sort attributes together with each tuple set into the *conflict resolution* operator $\lambda_{CR,S}$.

In this section we defined operators that are able to handle uncertainties, as well as conflicting data. They can be customized to perform a variety of different conflict resolutions. In the next section we give extensions to SQL in order to include our operators.

4.5 Extending SQL to Express Data Fusion Operators

For ease of use in a relational database management system we extend SQL to express *subsumption* (β), *minimum union* ($\oplus$), *complementation* (κ), *complement union* ($\boxplus$), and *data fusion* (ϕ, ϕ^{β}). The prototype system that we developed within our research group (see Section 8) supports this proposed syntax.

First, we extend SQL to express the removal of subsumed tuples (β) and the *complementation* operator (κ). This is done by introducing two new keywords, namely `SUBSUMED` and `COMPLEMENTED`, that can be used in the same place as `DISTINCT`. It adds an additional operator (β or κ) at the top of the query tree. The three

keywords cannot be used together, as the order is important. So, in order to e.g., first handle subsumed and then complementary tuples, the order needs to be given by using *subsumption* in a subquery[1].

Extending SQL to allow for *outer union*, *minimum union*, and *complement union* is done by introducing the three new keywordss `OUTER UNION`, `MINIMUM UNION`, and `COMPLEMENT UNION`. They are used in the same way as `UNION` is already used and connects two subqueries.

We next present an intuitive expression of the *data fusion* operator (ϕ, ϕ^{β}) by means of the FUSE BY statement. The FUSE BY statement represents a simple way of expressing queries that fuse multiple tuples describing the same object into one tuple while resolving uncertainties and contradictions. It is based on the standard SQL syntax for `SELECT-FROM-WHERE` queries and also resembles in syntax and semantics the GROUP BY statement.

The syntax diagram of the FUSE BY statement is shown in Figure 4.1. The statement consists of three main parts: 1. The `SELECT` part that specifies the conflict resolution, 2. the `FUSE FROM` part giving the source and 3. the `FUSE BY` part which holds information to identify objects. `FUSE BY SUBSUMED` is used in order to specify the second variant of the *data fusion* operator that includes the removal of subsumed tuples.

Tuples going into the fusion process are from the tables given in the `FROM` clause. In this place standard joins are possible, as well as as are arbitrary subselects. If `FUSE FROM` is used, it indicates combining the given tables by *outer union* instead of *cross product*, saving specifying complex subselects in most cases as can be seen further on. Please note that when using `FUSE FROM`, tuples are ordered in the order of the tables specified by default. Using `FUSE FROM A, B`, all tuples from A are considered before the tuples from B.

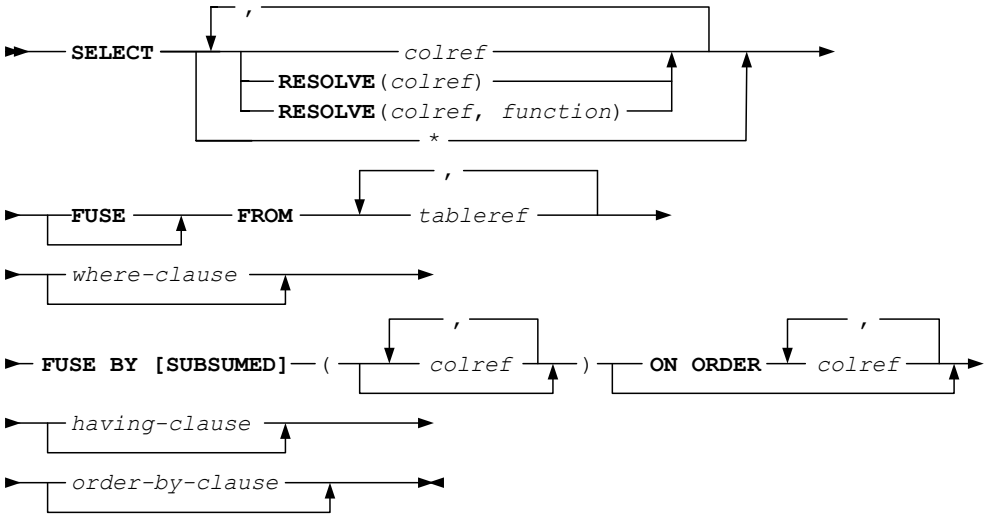


Figure 4.1: Syntax diagram of the FUSE BY statement

Similar to the `GROUP BY` clause, the `FUSE BY [SUBSUMED]` clause defines which object representations are considered to represent the same real-world objects, and are therefore fused into one single tuple. The attributes given here serve as identifier. `ON ORDER` influences the order in which tuples are considered when resolving conflicts. All attributes that do not appear in the `FUSE BY` clause may contain data conflicts. The keyword `RESOLVE` in the `SELECT` clause marks these columns and also serves to specify a conflict resolution function (*function*) to resolve conflicts in this column. The wildcard '*' or not specifying a conflict resolution function results in a default conflict resolution behavior. Keep in mind that both the `HAVING`-clause and `ORDER BY` clause can be used additionally and keep its original meaning.

The parameters of the data fusion operator $\phi_{F,CR,O}$ (or $\phi^{\beta}_{F,CR,O}$ respectively) map as follows to the syntax of the FUSE BY statement: Columns that are given in set F are listed in the `FUSE BY` clause and can be included as is in the `SELECT` clause. Conflict resolution functions as given in CR are included together with the keyword `RESOLVE` in the `SELECT` clause. Finally, the sort order from O is included in the statement as the `ON ORDER` clause.

A small example for a FUSE BY statement using the example relations is:

[1] SELECT COMPLEMENTED * FROM (SELECT SUBSUMED * FROM basetable)

```
SELECT Name, RESOLVE(Address, longest)
FUSE FROM Police, Hospital
FUSE BY (Name)
```

This fuses data from the Police and Hospital relations, leaving just one tuple per person. People are identified by their name and conflicting Address values are resolved by taking the longest address, assuming that it contains more information.

Figure 4.2 shows a query that is used to produce a table similar to the one in Figure 1.3 from Section 1.2. Please note that the order of the tables and the order by Birthdate influences the values chosen, e.g. the address of *Miller*.

```
SELECT Name, RESOLVE(Birthdate, max), RESOLVE(Sex),
       RESOLVE(Blood, vote), RESOLVE(Adress)
FUSE FROM Police, Hospital
FUSE BY (Name) ON ORDER Birthdate
```

Figure 4.2: Example query that produces the result from Table 1.3 on page 7

The following Figure 4.3 shows three more examples of Fuse By queries.

```
SELECT *
FROM T1
FUSE BY (A)
```

(a) Query 1: Resolves data conflicts in Table T1

```
SELECT *
FROM T1
FUSE BY ()
```

(b) Query 2: Removes subsumed tuples from Table T1

```
SELECT *
FUSE FROM T1, T2
FUSE BY ()
```

(c) Query 3: Combines two tables by *minimum union*

Figure 4.3: Three simple Fuse By statements

Query 1 (Figure 4.3(a)) groups the tuples of Table T1 by the values in column A. All other columns (replacing wildcard `*`) of table T1 may contain conflicting data that is resolved by the default conflict resolution function coalesce. This way, the statement behaves like an ordinary `GROUP BY` with a coalesce aggregation. Fusion by more than one column is possible, replacing attribute A with all desired columns. In Query 2 (Figure 4.3(b)) there is no column present in the `FUSE BY` clause. All tuples are treated equally as one large group. Thus, subsumed tuples are removed. Conflicts are not resolved and this corresponds to the result after applying the *subsumption* operator. The same result is obtained by the statement `SELECT SUBSUMED FROM T1`, directly expressing the *subsumption* operator. More than one table given in the `FUSE FROM` clause are combined by *outer union*. In this case, columns with same names from different tables fall together, columns present in only some tables are padded with null values. If needed, a renaming of columns can be done as a subselect in the `FUSE FROM` clause. Query 3 (Figure 4.3(c)) first combines the two tables T1 and T2 by *outer union*. It completes missing values in columns by null values and then removes subsumed tuples. Together with coalesce as default conflict resolution function this corresponds to the result of a *minimum union* operator. The same result is obtained by the statement `(SELECT * FROM T1) MINIMUM UNION (SELECT * FROM T2)`. Examples with three or more tables look and behave similarly.

Part III

Using Data Fusion

An algorithm must be seen to be believed.

(Donald Knuth)

5

Implementation of Data Fusion

So far we have considered strategies and operators for different data fusion tasks. This chapter now focuses on the implementation of these strategies and operators. In the following Sections 5.1 to 5.3, algorithms that implement the *subsumption* (β), *complementation* (κ), *conflict resolution* (λ), and *data fusion* (ϕ) operators are described in more detail. For *subsumption* (Section 5.1) and *complementation* (Section 5.2), several alternative algorithms are given with references to Chapter 7 where they are evaluated. Last, alternatives are discussed, how some of the techniques can be realized by using SQL statements and constructs (Section 5.4). Some of these algorithms are implemented in the two research systems HUMMER and FUSEM which are presented subsequently in Chapter 8.

5.1 Subsumption and Minimum Union

After having defined the necessary concepts for *subsumption* and *minimum union* in the last chapter (see Section 4.2), we now examine different implementation alternatives for computing *subsumption*. To implement *minimum union*, *subsumption* is simply combined with *outer union* by executing one after the other. The combination of *outer union* and *subsumption* to form *minimum union* is discussed in more detail in Section 6.2.1 when considering optimization rules. Although the need for efficient and general algorithms for *subsumption* has been acknowledged, there exist few approaches so far (see Section 5.1.6 for some related work). In this section, a baseline algorithm (*Simple Subsumption*) is given first, which serves as basic implementation and is used as building block and for comparison. Following the baseline algorithm three more advanced implementations of the *subsumption* operator are presented in Sections 5.1.2, 5.1.3, and 5.1.4: First, we consider an algorithm based on an indexing technique that saves runtime by precomputation of an index structure. Second, a partitioning technique in the spirit of hash-based duplicate elimination (Ullman et al., 2001) is presented that can be applied in addition to the *Simple Subsumption* or any other subsumption algorithm to potentially further save runtime (*Partitioning* algorithm). Last, the section concludes with a brief mention of an alternative algorithm for computing *subsumption*, namely the *Null-Pattern-Based Subsumption* algorithm, that potentially improves on the worst-case time complexity of the *Simple Subsumption* algorithm. Two of these three different implementation techniques can be found in (Bleiholder et al., 2010b).

5.1.1 Baseline Algorithm

The naive way of removing all subsumed tuples from a given relation R consists of comparing all pairs of tuples from R and including a tuple in the result only if it is not subsumed by any other tuple in R. Clearly, this is not an efficient solution, and there are two ways of speeding it up: First, pairwise comparisons of a given tuple to other tuples are stopped as soon as a tuple that subsumes the one at hand is found. Second, tuples are grouped in buffers along the process in such a way that two tuples are in the same buffer, if one (possibly) subsumes the other. Then, tuples need only be compared to tuples in the same buffer and not to tuples in different buffers. The actual number of buffers needed is not predefined and depends on the tuples in the relation and their respective subsumption relationships. In the remainder of this chapter, we refer to this

improved naive algorithm (see Algorithm 1) as the *Simple Subsumption* algorithm, or *SMPS* for short. The algorithm is blocking, as all tuples from R need to be processed before the first tuple can be sent to the output. Its worst-case time complexity is $\mathcal{O}(n^2)$ with n being the number of tuples in R.

Algorithm 1: The baseline Algorithm: *Simple Subsumption* (*SMPS*).

Input: Relation R of tuples $t_i \in T$ (with attributes $a_j \in A$)
Output: Relation β (R)

```
1:  Create an empty list of buffers B, where each buffer b_j can hold tuples;
2:  for all tuples t_i of the input relation R do
3:      inAnyB ← false;
4:      for all buffers b_j in B do
5:          conflict ← 0;
6:          for all tuples s_k in b do
7:              if t_i and s_k conflict in at least one attribute value (conflicting values) then
8:                  conflict ← −1;
9:              else
10:                 conflict ← number of NULL values in t_i;
11:             end if
12:             if conflict = −1 then
13:                 Break;
14:             end if
15:         end for
16:         if conflict > −1 then
17:             Add tuple t_i to buffer b_j if there is no t′ in b_j that subsumes t_i, if necessary also removing all t″ from b_j
                that are subsumed by t_i;
18:             inAnyB ← true;
19:             Break;
20:         end if
21:     end for
22:     if inAnyB = false then
23:         Create new tuple buffer b′;
24:         Add t_i to b′;
25:         Add b′ to B;
26:     end if
27: end for
28: Write all tuples t_i in all buffers b_j in B to the output;
```

5.1.2 Employing a Bitmap Index

Bitmap indexes are a special type of index structure that are used in DBMS (Ullman et al., 2001). We show in this section how we can employ such a bitmap index structure to remove subsumed tuples. The main idea is to decide for each tuple t of a relation R whether it is subsumed by another tuple or not by doing a quick check on the index structure. Once the index structure exists, we gain the advantage of the operator not being blocking: for each t from the input we can immediately decide, if it is part of the output or not. However, the disadvantages are a larger memory consumption (to hold the index), and additional runtime (to compute the index structure).

Example 3 (Bitmap Index Structure)
To better understand how using a bitmap index helps in removing subsumed tuples, we consider the six example tuples seen below. The relation consists of three columns A, B, and C. In addition, some information is given whether a tuple is subsumed and by what other tuple. Last, we assume an ordering of the tuples, illustrated by a separate column ID, which contains a tuple identifier:

ID	**A**	**B**	**C**	*subsumed by tuple with ID*
1	1	2	2	-
2	1	⊥	1	5,4
3	2	1	⊥	-
4	1	2	1	-
5	1	1	1	-
6	1	⊥	⊥	1,2,4,5

□

Considering Example 3 above, the algorithm first precomputes the following bitmap index structures, one index structure for each column:

A	
1	110111
2	001000

B	
1	001010
2	100100

C	
1	010110
2	100000

To the left, we see the index structure for column *A*. There are only two distinct NON-NULL values in column *A*, so there are only two bitmaps to be stored. The length of the bitmap is the number of tuples in the relation. There is a bit set (marked by a 1) in the bitmap for value "1" if the value is present in the corresponding tuple. The bitmap `110111` therefore indicates that every tuple except tuple with *ID* 3 has "1" as its value in column *A*.

Whenever there is a NULL value in a column, we store this information in a separate NULL index structure, also as a bitmap index:

A	
⊥	000000

B	
⊥	010001

C	
⊥	001001

The basic idea is then to only consider tuples that contain NULL values as only these can be subsumed by other tuples and need to be removed. More precisely, the algorithm looks at tuples t that have NULL values in an attribute and tries to find other tuples where all other NON-NULL attribute values of t coincide. All other tuples are not considered further and are written directly to the output.

For all potentially subsumable tuples (all that contain at least one NULL value) we search for all other tuples that have the same values in the attributes that are NON-NULL. This is done by matrix/vector multiplication of bit matrices and vectors. As a result we obtain a vector that gives us the *ID* values of the relevant tuples that do subsume or are equal to the tuple in question. We then delete the tuple (if it is subsumed by at least one tuple) or keep it (if it is only equal). More precisely, the *Index-Based Subsumption* algorithm (see Algorithm 2 on page 63) executes the following computations to remove subsumed tuples from a relation R:

- The algorithm considers all tuples t_i from relation R, one after the other, and checks, if it has a NULL value, i.e., it iterates through all ⊥ entries in the indexes, in parallel.
- If it finds at least one 1 entry in one of the NULL index structures (bit set, tuple t_i has a NULL value), then let $\vec{n}$ be the ith transposed unit vector. For all other tuples: put t_i into the result (t_i is not subsumed). Let k be the attribute where $t_i[k]$ is NULL.
- For all other attributes j that do not have a 1 at this position (i.e., $t_i[j]$ is not a NULL value) do the following:
  - Let the bit index of the attribute j be matrix I_j, and
  - compute $\vec{a}_j = (I_j \vec{n}^T)^T I_j - \vec{n}$
- Let $\vec{A}_i$ be $\bigwedge_j \vec{a}_j$
- if $|\vec{A}_i| \geq 1$ (more than one entry of this vector is 1) then t_i is equal to or subsumed by at least one other tuple. Check all t_j with $\vec{A}_i[j] = 1$ whether they are equal to t_i. If so, insert t_i into the result (t_i is not subsumed).

Considering Example 3 again, in order to determine the relation without subsumed tuples the algorithm scans through all $\perp$ entries from left to right in parallel. It first places the tuple with ID 1 into the output as there is no NULL value present. Next, it stops at the tuple with ID 2, as the NULL index structure for B has a bit set. Then, $\vec{n}$ is set to be (010000) and vectors $\vec{a_A}$ and $\vec{a_C}$ are determined (and finally $\vec{A_2}$) as there is only a NULL value in column B.

$$\vec{a_A} = (I_A\vec{n}^T)^T I_A - \vec{n} = (\begin{pmatrix}1&1&0&1&1&1\\0&0&1&0&0&0\end{pmatrix}\begin{pmatrix}0\\1\\0\\0\\0\\0\\0\end{pmatrix})^T\begin{pmatrix}1&1&0&1&1&1\\0&0&1&0&0&0\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0\end{pmatrix}$$

$$= (\begin{pmatrix}1&0\end{pmatrix}\begin{pmatrix}1&1&0&1&1&1\\0&0&1&0&0&0\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0\end{pmatrix}$$

$$= (\begin{pmatrix}1&1&0&1&1&1\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0\end{pmatrix} = \begin{pmatrix}1&0&0&1&1&1\end{pmatrix}$$

$$\vec{a_C} = (I_C\vec{n}^T)^T I_C - \vec{n} = (\begin{pmatrix}0&1&0&1&1&0\\1&0&0&0&0&0\end{pmatrix}\begin{pmatrix}0\\1\\0\\0\\0\\0\\0\end{pmatrix})^T\begin{pmatrix}0&1&0&1&1&0\\1&0&0&0&0&0\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0\end{pmatrix}$$

$$= (\begin{pmatrix}1&0\end{pmatrix}\begin{pmatrix}0&1&0&1&1&0\\1&0&0&0&0&0\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0\end{pmatrix}$$

$$= (\begin{pmatrix}0&1&0&1&1&0\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0\end{pmatrix} = \begin{pmatrix}0&0&0&1&1&0\end{pmatrix}$$

In the next step, $\vec{A_2}$ is computed by anding $\vec{a_A}$ and $\vec{a_C}$ to be (000110). Finally, tuples with ID 4 and 5 are checked (as indicated by the two bits set in $\vec{A_2}$) for equality with tuple with ID 2 with the result that they are not equal and that tuple with ID 2 is subsumed and does not belong to the output relation. Algorithm 2 details the entire algorithm.

Although this is an elegant way of expressing the removal of subsumed tuples, it comes with some disadvantages. First, the runtime complexity is dominated by the complexity of matrix multiplication, which is $\mathcal{O}(n^3)$ for a straightforward implementation and $\mathcal{O}(n^{lg7})$ for Strassens Algorithm (Cormen et al., 2001). Second, the index structure needs to be computed from scratch, if *subsumption* is applied to intermediate relations (contrary to base relations, where precomputation and storage is possible). Still, even if it would only be applied to base relations there would still be the problem of index management and especially the problem of how to update the index structure when inserting new tuples into the base relation. However, as advantage comes that the basic operations are just bit operations and a few value comparisons, instead of the complex subsumption checks needed in the baseline algorithm. Further on in Chapter 7 we observe that the *Index-Based Subsumption* algorithm can – at present – not compete with the other algorithms presented in this chapter. Additionally, runtime is expected to depend on the number of NULL values in the relations, which is also confirmed by the experiments (see also Chapter 7). However, the *Index-Based Subsumption* algorithm is an idea that is worth further investigation.

Algorithm 2: The two-pass *Index-Based Subsumption* algorithm.

Input: Relation R of tuples $t_i \in T$ (with attributes $a_j \in A$)
Output: Relation β (R)

```
1:  Read tuples t_i from relation R and create bitmap indexes;
2:  Create output buffer out ← ∅;
3:  for all tuples t_i from relation R do
4:      if at least one t_i[k] is ⊥ then
5:          Ā_i ← 0⃗, length number of tuples;
6:          for all columns j do
7:              if t_i[j] ≠ ⊥ then
8:                  Let n⃗ be the ith transposed unit vector;
9:                  Let the bit index of attribute j be matrix I_j;
10:                 Compute ā_j = (I_j n⃗^T)^T I_j − n⃗;
11:                 Compute Ā_i ← Ā_i ∧ ā_j;
12:             end if
13:         end for
14:         if |Ā_i| > 0 then
15:             cnt ← 0;
16:             for all tuples t_k indicated by Ā_i to be candidates (A_i[k] = 1) do
17:                 if t_k equals t_i then
18:                     cnt = cnt + 1;
19:                 end if
20:             end for
21:             if |Ā_i| = cnt then
22:                 Insert t_i into out;
23:             end if
24:         else
25:             Add t_i to out;
26:         end if
27:     else
28:         Insert t_i into out;
29:     end if
30: end for
31: Write all tuples t_j ∈ out to the output;
```

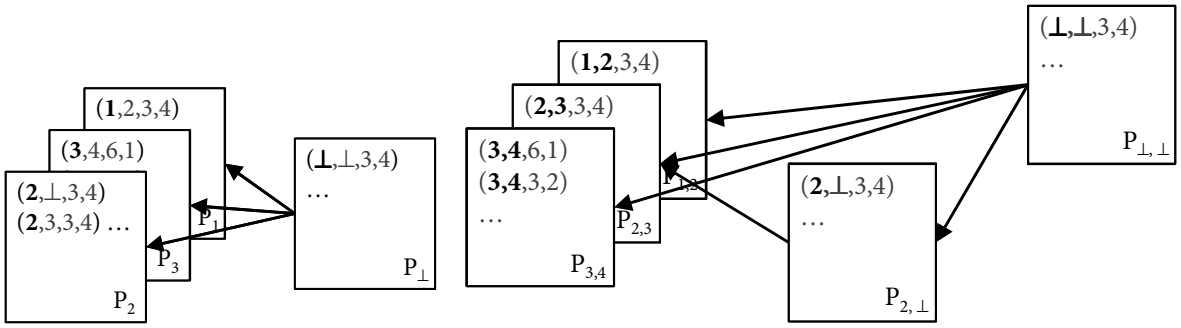
5.1.3 Input Partitioning by Column Values

To improve the runtime for computing *subsumption*, we now consider to partition the input using a simple criterion in such a way that tuples where one does not subsume the other do not fall into the same partition. Then, the input of any subsumption algorithm like the *Simple Subsumption* algorithm is reduced to the size of a partition, which potentially results in fewer tuple comparisons and hence better runtime. Further on, we also consider this technique in combination with another subsumption algorithm, the *Null-Pattern-Based Subsumption* algorithm (Section 5.1.4). See Algorithm 3 on page 64 for a description of the general *Partitioning* algorithm.

Essentially, we select a partitioning criterion, such as a column c and partition all tuples of the input relation according to their c-values. If there are d distinct NON-NULL values in c, we obtain d partitions P_i (one for each value of attribute c). We also create an additional partition containing all tuples whose c-value is NULL and which we denote as the NULL partition $P_\perp$. As a side note, the input partition here is different to the grouping used in the baseline algorithm because partitions may very well contain conflicting tuples. The grouping used in the baseline algorithm is thus more fine-grained.

From the definition of subsumption follows that there cannot be tuples t, t' such that t subsumes t' and the two tuples belong to different NON-NULL partitions. Therefore, we can handle each NON-NULL partition P_i separately. The NULL partition $P_\perp$, in addition to having subsumed tuples removed, needs special treatment, because there can be tuples t, t' such that t subsumes t', where t' belongs to $P_\perp$, but t does not. Therefore,

we need to compare each tuple in $P_\perp$ with each tuple from all NON-NULL partitions. Such a partitioning is visualized in Figure 5.1(a).



(a) Partitioning by one column (b) Partitioning using two columns

Figure 5.1: Example partitioning by one or two columns and comparison scheme given by the arrows.

In case only one column is selected to partition the relation, the *Partitioning* algorithm performs the following steps: First, it produces a partitioning of the input. Second, it removes all subsumed tuples within the NULL partition $P_\perp$ by applying an appropriate subsumption algorithm $\mathcal{A}$ (e. g., the *Simple Subsumption* algorithm). It also reads each NON-NULL partition P_i and removes all subsumed tuples within it by applying $\mathcal{A}$. After removing all subsumed tuples from P_i, the remaining tuples constitute $P_i' = \beta(P_i)$ and are part of the output. Third, the algorithm has to check if there are tuples in the remaining NULL partition $P_\perp'$ that are subsumed by tuples in $P_1', \ldots, P_d'$. Therefore, each tuple t in $P_\perp'$ is compared to the tuples in the remaining NON-NULL partitions $P_1', \ldots, P_d'$ and, in case t is subsumed, it is removed from $P_\perp'$. Hence, after considering all tuples in $P_\perp'$, the remaining tuples in it are also part of the output.

Algorithm 3: The *Partitioning* algorithm.

Input: Relation R of tuples $t_i \in T$ (with attributes $a_j \in A$)
Output: Relation β (R)

```
1: Use schema S and statistics to choose partitioning column c ∈ A;
2: Partition set of input tuples T from relation R by column c into d NON-NULL partitions and one NULL partition;
3: Let P⊥ be the NULL partition, and the d Pi's the NON-NULL partitions;
4: Choose a subsumption algorithm A to remove subsumed tuples;
5: Remove subsumed tuples in the NULL partition P⊥, i.e., P⊥' ←β (P⊥);
6: for all d NON-NULL partitions Pi do
7:     Pi' ←β (Pi) using algorithm A;
8:     for all tuples t' ∈ P⊥' do
9:         if t' is subsumed by any tuple t ∈ Pi' then
10:            Remove t' from P⊥';
11:        end if
12:    end for
13:    Write all tuples tj in Pi' to the output;
14: end for
15: Write all tuples tj in P⊥' to the output;
```

Choosing more than just one column to partition the input results in one complete NULL partition (containing all tuples with NULL values in all selected columns) and several partial NULL partitions (containing tuples where only some of the selected columns have NULL values). Tuples from the complete NULL partition need to be compared to tuples from all other partitions, whereas tuples from a partial NULL partition P need to be compared only to tuples from those partitions P' that satisfy the following conditions:

1. There is at least one selected column where P has a NULL value, but P' does not.

2. The value of a selected NON-NULL column in P coincides in P and P', i. e., if the value of a selected column is NON-NULL in both partitions, it must be the same.

3. For each selected NULL column in P, the NON-NULL column value in P' is arbitrary.

An example for a partitioning by two columns is shown in Figure 5.1(b). According to the given conditions, tuples from the complete ($P_{\perp,\perp}$) or the partial NULL partitions (e.g., $P_{2,\perp}$) only need to be compared with all tuples from appropriate adjacent partitions to the left that may contain subsuming tuples (following the arrows). Therefore, to remove tuples from $P_{2,\perp}$ we need to consider only $P_{2,3}$.

The subsumption algorithm $\mathcal{A}$ that is used inside the *Partitioning* algorithm to implement the *subsumption* operator β can be implemented in different ways. In the experiments (see Chapter 7) the *Partitioning* algorithm is combined both with the *Simple Subsumption* algorithm (*Partitioning(SMPS)*) and the *Null-Pattern-Based Subsumption* algorithm (*Partitioning(NPBS)*).

The overall runtime of the *Partitioning* method depends both on the runtime of the algorithm used for computing *subsumption* and the value distribution of the partitioning attribute c. In the following, we consider only the case where just one column is selected for partitioning. In case more than one column is selected, a similar argument holds. For an input relation of n tuples, let $T_{\mathcal{A}}(n)$ be the runtime of algorithm $\mathcal{A}$ that is used inside the *Partitioning* algorithm. Then, the overall runtime complexity of the *Partitioning* method with d NON-NULL partitions is bounded by

$$\begin{aligned} & T_{\mathcal{A}}(|P_\perp|) && \text{to compute } \beta\ (P_\perp) \\ + & \sum_{1\le i\le d} T_{\mathcal{A}}(|P_i|) && \text{to compute all } \beta\ (P_i) \\ + & \sum_{1\le i\le d} |\beta\ (P_i)| \cdot |\beta\ (P_\perp)| && \text{to compare } \beta\ (P_\perp) \text{ with all } \beta\ (P_i) \end{aligned} \tag{5.1}$$

where $|P_i|$ is the number of tuples in partition P_i. Using the *Simple Subsumption* algorithm as algorithm $\mathcal{A}$ and estimating $|\beta\ (P_i)|$ by the average size n/d of a partition (distributing n tuples over d partitions), the overall runtime of the *Partitioning* algorithm evaluates to $\mathcal{O}(n^2) + \mathcal{O}(d \cdot (\frac{n}{d})^2) + \mathcal{O}(d \cdot \frac{n}{d} \cdot n) = \mathcal{O}(n^2)$. In the above formula, the last term becomes more complicated when considering more than one column for partitioning. The impact of the value distribution is more difficult to quantify. When c is a key attribute, i.e., unique and not NULL, all three terms above are zero and no comparisons are necessary. When all values of attribute c are equally distributed, no term dominates the others, which still leads to satisfactory results. For each NON-NULL partition that has significantly larger size than others, the second term exhibits peaks for the respective partitions. Depending on how many and how big the peaks in the number of comparisons are, these peaks may dominate runtime. The worst case however occurs when the NULL partition contains significantly more tuples than the remaining partitions. Indeed, this creates a peak for every ith product in the third term and thus definitely dominates runtime. Avoiding peaks in the histogram is the goal of the third rule in our heuristic (see below) to select c. Experiments in Chapter 7 confirm this analysis.

We apply a heuristic approach that chooses a single partitioning column c based on the following rules applied in that order:

1. Choose a key column (minimal partition sizes $|P_i|$, thus minimizing the first term),

2. Choose a column that does not contain NULL values (minimizing the third term), and

3. Choose a column c where the product of the number of distinct values in c and the maximum frequency of a value in c is minimal. This rule prefers a column where values are distributed equally over a skewed column, where one or only a few values appear with a much higher frequency than others.

To apply the above heuristic, we rely on both the schema S of the input relation and on statistics about value distributions that were collected beforehand.

When considering more than just one column for partitioning, the number of partitions increases or at least stays the same when increasing the number of columns. At the same time, the average size of each partition

decreases. In Chapter 7 experiments show that it is generally beneficial to choose as many columns as possible for partitioning (maximum m columns, with m attributes in schema S).

Nevertheless there are cases where it might be better to use less than the maximal possible number of columns for partitioning. This is the case, if we can save the time for partitioning, i.e., if we can access tuples in order, for example by using an existing index. Also, given a fixed number of k columns for partitioning, one can formulate the optimization problem of finding the combination of k columns that minimizes the number of subsumption comparisons and by that the runtime. In Chapter 7, experiments show that we can already reach a good solution for this optimization problem by applying a greedy heuristic: First, we compute for each column c of the relation the term $D_c = d_c \cdot (\frac{n}{d_c})^2 + (d_c \cdot |P_\perp|)$. Here, d_c is the number of all partitions in column c, $\frac{n}{d_c}$ is the average size of a partition and $|P_\perp|$ is the size of the NULL partition. Then, D_c approximates the runtime of the *Partitioning* algorithm, using the *Simple Subsumption* algorithm as $\mathcal{A}$ and one column for partitioning. Second, we choose the k smallest columns with respect to D_c.

5.1.4 Employing Patterns of Null Values

We only briefly present the *Null-Pattern-Based Subsumption* algorithm here as it has been developed as part of a joint project with Sascha Szott and others from our research group and refer the reader to (Bleiholder et al., 2010a) for a more detailed description.

The main idea of this algorithm is to use information about NULL values in the tuples in a more complex way than it has been used in the *Partitioning* algorithm. Simply by looking at patterns of NULL values, we can exclude tuples from the set of possibly subsuming tuples and avoid many tuple comparisons. Specifically, tuples are sorted into buckets, one bucket holding tuples that have the same NULL pattern, i.e., the same distribution of NULL values in their attributes.

Considering an example, Table 5.1 shows tuples from the disaster management domain along with their corresponding bucket on the right. Bucket $B_{(11110)}$ for example holds all tuples (first and third tuple) where all attributes except the last one have NON-NULL values. Two tuples where one subsumes the other are sorted into different buckets.

Example Data

ID	**Name**	**DOB**	**Sex**	**Address**	**Blood**	*Bucket*
1	Miller	7/7/59	m	12 Main	⊥	$B_{(11110)}$
2	Miller	⊥	⊥	12 Main	⊥	$B_{(10010)}$
3	Peters	1/1/53	m	34 First	⊥	$B_{(11110)}$
4	Peters	1/1/53	⊥	⊥	AB	$B_{(11001)}$
5	Peters	1/1/53	m	⊥	⊥	$B_{(11100)}$
6	Miller	⊥	f	⊥	B	$B_{(10101)}$
7	Miller	7/7/59	m	⊥	o	$B_{(11101)}$

Table 5.1: Bucket assignment in the example scenario.

The algorithm removes subsumed tuples as follows: In a first step, the algorithm sorts all tuples into a bucket data structure (see Figure 5.2) and then starts comparing tuples in order to determine if one subsumes the other. To improve efficiency, the set of buckets under consideration is pruned based on the definition of subsumption.

The pruning scheme is also depicted in Figure 5.2 where the arrows denote the subsumption relation. For example, tuples from bucket $B_{(01111)}$ can only be subsumed by tuples from bucket $B_{(11111)}$. For instance, Tuple 1 subsumes Tuple 2 over the path of buckets $B_{(11110)}, B_{(10110)}, B_{(10010)}$. Using this graph representation, only descendant buckets of bucket B_j need to be considered when looking for all tuples that can be potentially subsumed by a tuple in B_j.

The algorithm proceeds incrementally and first consider all buckets with i NULL values before considering buckets with $i+1$ NULL values. As soon as we determine that one tuple in $B_{j'}$ is subsumed by looking at all buckets to the left, we delete it from its bucket.

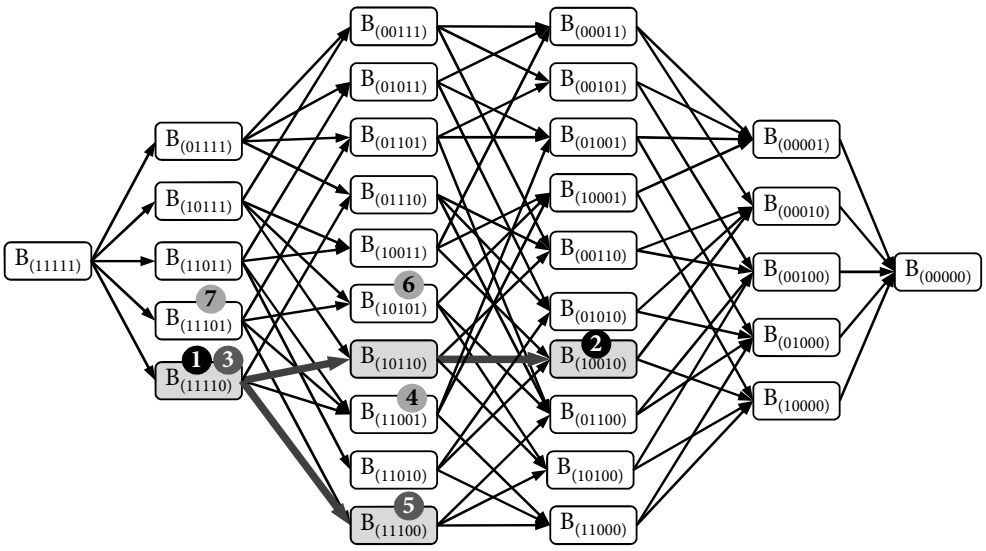


Figure 5.2: Dependency between buckets and the placement of tuples from Table 5.1 in buckets.

To efficiently determine if a tuple t' in $B_{j'}$ is subsumed by a tuple t in B_j, we use binary search in the buckets in question. As an example, consider tuple 5 in Figure 5.2, which lies in the bucket $B_{(11100)}$. The parent buckets are $B_{(11110)}$ and $B_{(11101)}$, which contain tuples 1, 3, and 7. After projecting out all NULL columns on tuple 5 and the parent buckets, we obtain $\pi(t')$ =(Peters, 1/1/53, m) for t' being tuple 5 and $\pi(B_{(11100)})$ ={(Miller, 7/7/59, m), (Peters, 1/1/53, m), (Miller, 7/7/59, m)}. After sorting $\pi(B_{(11100)})$ and applying binary search we detect that $\pi(B_{(11100)})$ contains $\pi(t')$ and conclude that there is a tuple subsuming t'. Note, that in this example by applying binary search only one comparison step is needed to find $\pi(t')$ within $\pi(B_{(11100)})$.

In terms of the number n of tuples in the input relation and the number of attributes m, the worst case time complexity of the *Null-Pattern-Based Subsumption* algorithm is $\mathcal{O}(\min\{2^m, n\} n \log n)$. Assigning each tuple to its corresponding bucket in the first step needs $\mathcal{O}(n)$ time. In the second step each of at most n non-empty buckets has to be processed. In principle, for each such bucket B, the set $\pi(B)$ has to be computed by considering all directly and indirectly connected non-empty buckets to the left of B (according to the scheme shown in Figure 5.2). As the size of $\pi(B)$ is bounded by n, the runtime of this process can be bounded by $\mathcal{O}(n)$. Afterwards, $\pi(B)$ has to be sorted, which needs $\mathcal{O}(n \log n)$ runtime. We now have to check the at most $\mathcal{O}(n)$ elements in B for subsumption by applying a binary search in $\pi(B)$ for each of it, thus needing $|B| \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$ runtime in total.

With m as the number of attributes at most 2^m buckets are built in the first step. Thus there can be at most 2^m iterations within the second step, which yields to an overall runtime of $\mathcal{O}(2^m n \log n)$. For very wide tables this can become substantial, but in our experience (e.g., customer relationship management data integration scenarios), tables usually had less than ten relevant attributes and the actual number of non-empty buckets was far below 2^m. In fact for large values of n, the total number of buckets, 2^m, is substantially smaller than n. Therefore, in most cases it is valid to consider m as a constant, i.e., $m \in O(1)$, giving an overall runtime for the *Null-Pattern-Based Subsumption* algorithm of $\mathcal{O}(n \log n)$.

Therefore, as long as $m \in o(\log(\frac{n}{\log n}))$ the asymptotic runtime of the *Null-Pattern-Based Subsumption* algorithm is smaller than that of the *Simple Subsumption* algorithm. Especially for the common case that the number m of attributes is a constant, the time complexity of the *Null-Pattern-Based Subsumption* algorithm is $\mathcal{O}(n \log n)$.

5.1.5 Null-Pattern Buckets vs. Input Partitions

Concluding the presentation of the *Null-Pattern-Based Subsumption* and *Partitioning* algorithms in the two previous sections we briefly summarize their similarities and differences. Using the *Partitioning* algorithm as a starting point, the *Null-Pattern-Based Subsumption* algorithm is an extension of a *Partitioning* algorithm

version that uses all m columns to partition the input relation. More specifically, this is done by combining tuples from all partitions that have the same NULL pattern (e. g., partition $P_{1,\perp}$ and $P_{3,\perp}$ are combined into bucket $B_{(10)}$) into a single bucket, and adding two additional features: a more intelligent processing order of the complete and partial NULL partitions and using binary search within the lexicographically sorted projections of the buckets. However, as the *Null-Pattern-Based Subsumption* algorithm is a subsumption algorithm of its own, it can also be used in combination with the input partitioning technique itself. It then acts as the subsumption algorithm $\mathcal{A}$ that is used to remove subsumed tuples from the individual partitions in the *Partitioning* algorithm.

When choosing among algorithms for implementing the *subsumption* operator we then have four different choices: a) the *Simple Subsumption* algorithm on its own, b) the *Partitioning* algorithm in combination with the *Simple Subsumption* algorithm (*Partitioning(SMPS)*), c) the *Partitioning* algorithm in combination with the *Null-Pattern-Based Subsumption* algorithm (*Partitioning(NPBS)*), and d) the *Null-Pattern-Based Subsumption* algorithm on its own. In the experiment chapter (see Chapter 7) we evaluate and compare all four possible choices.

5.1.6 Related Work on Subsumption and Minimum Union

Related work on conflict resolution, data fusion and other general approaches, techniques, and systems is widely covered in Chapter 9, briefly including subsumption where applicable. More specific related work to the definition and implementation of *subsumption* and *minimum union* is covered in the following.

The *minimum union* operator is defined in (Galindo-Legaria, 1994) and is used in many applications. For instance, *minimum union* is exploited in query optimization for *outer join* queries (Galindo-Legaria and Rosenthal, 1997). However, an efficient algorithm for the general subsumption task is still considered an open problem therein. An assumption in that work is that the combined base relations do not contain subsumed tuples, contrary to our work in the field of data integration. Another difference to our setting is the use of *join* instead of *union* to combine tables before removing subsumed tuples. Another use case of *minimum union* is its usage in Clio (Hernández et al., 2002) as one possible semantics to use a schema mapping for the creation of transformation rules (Popa et al., 2002).

Subsumption is used in (Rao et al., 2004) to create standardized *outer join* expressions, that way enabling *outer join* query optimization. The authors propose a rewriting for *subsumption* in SQL, using the data warehouse extensions provided by SQL. However, removing subsumed tuples using the proposed SQL rewriting depends on the existence of an ordering such that subsuming tuples are sorted next to each other. As subsumption establishes only a partial order, such an ordering does not always exist.

Another very efficient rewriting for *subsumption* is proposed in (Larson and Zhou, 2005). However, it is also not applicable in general, as special properties of the problem considered in (Larson and Zhou, 2005) are used, namely key properties of columns and additional conditions, such as certain columns not containing NULL values.

The problem of tuple subsumption can be transformed into the problem of minimizing tableau queries (tableau queries as introduced in (Aho et al., 1979)). The transformation works as follows: each tuple $t = \{(a_1, v_1), (a_2, v_2), \ldots, (a_m, v_m)\}$ of relation R is transformed into a literal $R(v_1, \ldots, v_m)$ of the corresponding tableau query Q, where m is the schema size and the v_j are the values of the tuple. Then, finding a minimal tableau Q' (equivalent to Q) is equivalent to removing subsumed tuples from R. There exist algorithms for minimizing tableau queries (see e.g., (Sagiv, 1983)), however, they only consider the general case, and are not able to take advantage of the characteristics of the transformed tableau query originating in our data integration scenario.

Computing *set containment joins*, e.g., (Melnik and Garcia-Molina, 2003), is a similar concept to removing subsumed tuples. Projecting on the *self set containment join* of a relation removes nearly all subsumed tuples: only those tuples t_1 that are subsumed by another tuple t_2, but also subsume another tuple t_3 ($t_3 \sqsubset t_1 \sqsubset t_2$), stay in the result and are not removed.

5.2 Complementation and Complement Union

As described in the last chapter (see Section 4.3), *complement union* aims at resolving a special type of data conflicts, namely *uncertainties*. After having defined the necessary concepts for *complementation* and *complement union*, we now look at different implementation alternatives to compute *complementation*. To implement *complement union*, we simply combine *complementation* with *outer union* by executing one after the other. This modular specification gives us more flexibility during integration; *complementation* can also be used to clean source databases before the integration process and it also allows us to combine relations by other means than *outer union*, before executing the *complementation* operator. To the best of our knowledge, (Bleiholder et al., 2010a) and (Bleiholder et al., 2010b) are the first that consider data fusion with the general semantics of *complement union* and give implementation details. In the following, first, a baseline algorithm to compute *complementation* is presented, the *Simple Complement* algorithm, and then the *Partitioning Complement* algorithm is given. Last, a third alternative algorithm is given based on the *Null-Pattern-Based Subsumption* algorithm for *subsumption*. The two latter algorithms follow similar ideas that have been used in the algorithms for *subsumption*: using input partitioning and NULL patterns. Experimental results for all algorithms are discussed in Chapter 7.

5.2.1 Baseline Algorithm

The main idea of the baseline algorithm – the *Simple Complement* algorithm – is based on the maximal complementing sets defined in Section 4.3. In a first pass the algorithm considers all input tuples and groups tuples that complement each other, thus building all sets of complementing tuples. In a second pass, the algorithm considers all these sets of complementing tuples and writes the complement of all the tuples of all maximal complementing sets to the output.

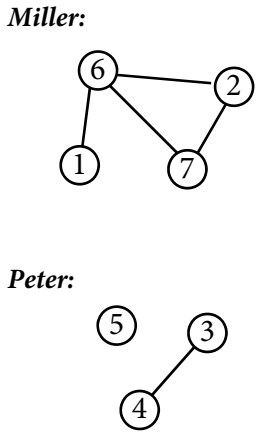
Figure 5.3(a) on page 69 shows an example of a source relation and the result after the application of *complementation*. In Figure 5.3(b) we show the complement relationships among the tuples numbered 1 to 7 (*Tuple-ID*). They are arranged in two groups (by name) for a better overview. Nodes are marked by their *Tuple-ID* attribute and represent the tuples in the example table. An edge represents a complement relationship between the two tuples. For instance, Tuples 1 and 6 complement each other, resulting in an edge between ① and ⑥ in Figure 5.3(b). In such a graph representation of the complement relationships, a maximal complementing set corresponds to a maximal clique.

Tuple-ID	**Name**	**Birthdate**	**Gender**	**Address**	**Blood**
1	Miller	5/5/61	⊥	12 Main St.	⊥
2	Miller	⊥	⊥	12 Main St.	⊥
3	Peter	1/1/53	Male	34 First St.	⊥
4	Peter	1/1/53	⊥	⊥	AB
5	Peter	1/1/53	Female	⊥	AB
6	Miller	⊥	Male	⊥	B
7	Miller	7/7/59	Male	⊥	⊥

⇓ complementation ⇓

Tuple-ID	**Name**	**Birthdate**	**Gender**	**Address**	**Blood**
1+6	Miller	5/5/61	Male	12 Main St.	⊥
2+6+7	Miller	7/7/59	Male	12 Main St.	B
3+4	Peter	1/1/53	Male	34 First St.	AB
5	Peter	1/1/53	Female	⊥	AB

(a) Example data and application of *complementation*



(b) Complement relationships between tuples

Figure 5.3: Example for the use of the *Simple Complement* algorithm.

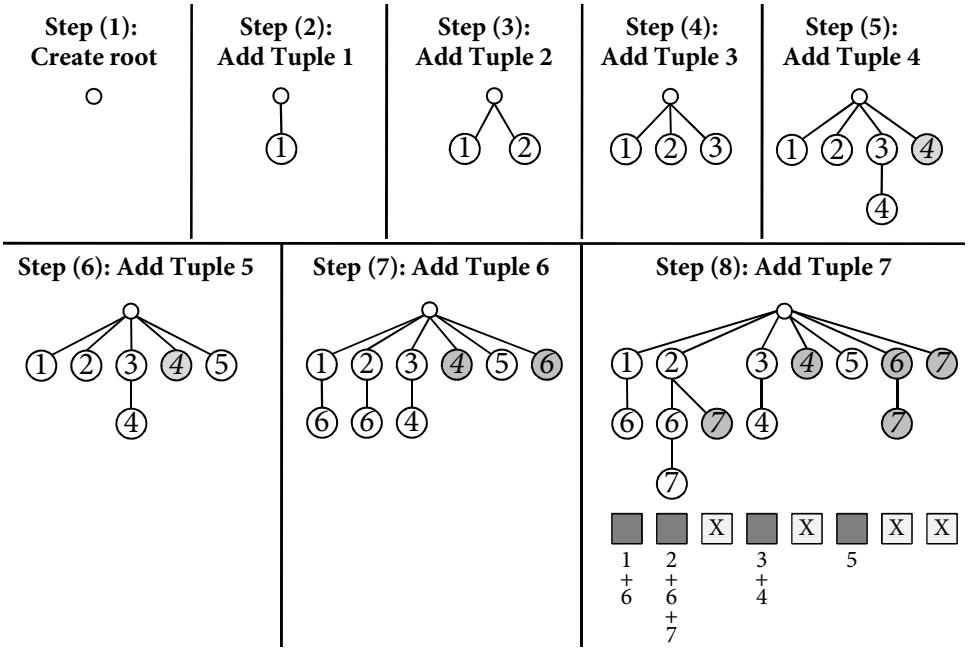


Figure 5.4: Steps (1) to (8) of the *Simple Complement* algorithm for the example data of Figure 5.3 and final output.

To compute and store the sets of complementing tuples, we subsequently insert each tuple of the relation into a tree data structure. The creation of this tree data structure is depicted in Figure 5.4 on page 70. At the first level of the tree structure, subsequently each tuple t_i of the relation is inserted. A tuple t_j is inserted into the subtree below t_i if it complements t_i. It is additionally inserted further down the subtree below t_i if it also complements the tuples that have already been inserted below t_i. Thus, we collect the sets of complementing tuples that contain that particular tuple t_i in the subtree below it. A path originating in t_i and ending at a leaf below t_i is such a set of complementing tuples. Then, among all stored sets of complementing tuples, we find the maximal complementing sets which we are interested in. The trees are internally stored as sets of paths from the root to all nodes. So for the tree structure for Tuple 3 from the example in Figure 5.4 (see Step 5, root–③–④) we store the two paths (root–③ and root–③–④) as node sets ($\{3\}, \{3, 4\}$). We keep the sets in such an order that we can traverse the tree in postorder just by scanning through the sets. To save space, instead of storing complete tuples, we store only tuple indexes and keep the assignment of index numbers to tuples separate. The algorithm is given as Algorithm 4 on page 72 and we refer to this version as the *Simple Complement* algorithm (*SMPC*).

Regarding the example run of our algorithm in Figure 5.4, the algorithm starts with an empty tree and inserts the first three tuples (Steps 1 to 4). Tuples are inserted as nodes in depth-first order, generally checking if the tuple that is inserted complements one of the already inserted tuples. As Tuples 1 to 3 do not complement each other, all are separately inserted. With Step 5 we add Tuple 4. As Tuple 4 complements Tuple 3, it is inserted below Tuple 3, forming the set of complementing tuples $\{3, 4\}$. It is subsequently also inserted below the root node. However, it is marked (italic and grey color in Figure 5.4), because it already has been added to a larger set (i.e., $\{3, 4\}$). As we see further on, marking is used to distinguish *maximal* complementing sets from other complementing sets.

Then, Tuples 5 and 6 are inserted. Tuple 6 is inserted below Tuple 1 and below Tuple 2 as it complements both tuples. It is further inserted below the root (marked). Finally, Tuple 7 is inserted in Step 8. Overall, there are four occurrences of Tuple 7 in the last step of Figure 5.4. It is inserted below the combination of Tuples 2 and 6 (path root–②–⑥), due to the fact that it complements the complement of 2 and 6. If a tuple t complements two other tuples r and s then it also complements the complement of r and s. As Tuple 7 also complements both Tuples 2 and 6 alone, it is also inserted below both of them. Due to the depth-first order, it has already been added to a larger set ($\{2, 6, 7\}$) and is therefore marked in both cases. It is also inserted below the root node, also marked.

In a second pass, the algorithm traverses the tree and identifies maximal complementing sets as the sets

formed by all tuples that lie on a root-to-leaf path that does not contain a marked leaf node. For instance, in Figure 5.4, the maximal complementing sets are those marked by a filled rectangle (below the last step). The participating original tuples are printed below. Once all maximal complementing sets have been identified, the algorithm computes and outputs the complement of each maximal complementing set. Thus, we return four tuples in our example that correspond to the complements of the sets $\{1,6\}$, $\{2,6,7\}$, $\{3,4\}$, and $\{5\}$.

The worst case time complexity of the algorithm is dominated by the first step. In the worst case, the nth tuple needs to be inserted into the subtree below all $(n-1)$ already inserted tuples. Given the size of the largest maximal complementing set by k, the size of these subtrees is at most 2^k. In the second pass the entire tree structure (n subtrees of size 2^k at most) is traversed to visit, check and – where necessary – output the leaf nodes (together with their paths from the root node). Thus, worst case runtime complexity of this algorithm (including the time needed for enumerating all MCSs and building the complements) is then quadratic in the number of tuples and exponential in the size of the largest subtree (which is equal to the size of the largest MCS and usually much smaller than the number of tuples, n), i.e., runtime complexity is $\mathcal{O}(n^2 2^k)$ where k is the size of the largest MCS and n is the number of tuples.

5.2.2 Input Partitioning by Column Values

To considerably improve runtime when computing *complementation*, we can apply the same input partitioning technique as for *subsumption* algorithms. The relation is partitioned by the different values of a fixed column c and then the algorithm is applied to each partition individually. However, the NULL partition containing all tuples with a NULL value in attribute c needs to be handled separately and different as in the case of *subsumption*, because tuple complementation (see Section 4.3) is not transitive. For each NON-NULL partition P_i the tuples from the NULL partition $P_\perp$ are added and then complementing tuples are replaced. The resulting set P_i' of tuples is then split into two groups: (1) the set of all tuples that came from the NULL partition and have not been complemented M_i and (2) all other tuples. Whereas we can output the tuples of case (2), we need to save all M_i until all P_i have been processed. Then, based on this division, all tuples from the NULL partition $P_\perp$ are identified that have not been used in *any* of the partitions, or all complemented tuples of just the NULL partition alone. This is done by computing the intersection of all M_i. We add the tuples from the intersection to the final output. We refer to this improved version as the *Partitioning Complement* algorithm. The input partitioning technique is given as Algorithm 5 on page 73. In choosing the partitioning column c we employ the same heuristics as used for *subsumption*.

We now consider runtime of the *Partitioning Complement* algorithm. The overall runtime of the *Partitioning Complement* algorithm depends basically both on the runtime of the algorithm used for computing *complementation* on the partitions and the value distribution of the partitioning attribute c (especially the size of the NULL partition). In the following, we consider only the case where just one column is selected for partitioning. In case more than one column is selected, a similar argument holds. For an input relation of n tuples, let $T_{\mathcal{A}}(n)$ be the runtime of algorithm $\mathcal{A}$ that is used inside the *Partitioning Complement* algorithm. Then, the overall runtime complexity of the *Partitioning Complement* method when having d NON-NULL partitions is bounded by

$$\begin{aligned} &\sum_{1\le i\le d} T_{\mathcal{A}}(|P_i \cup P_\perp|) && \text{to compute all } \kappa\,(P_i) \text{ (and } \kappa\,(P_\perp)) \\ &\quad + \sum_{1\le i\le d} |P_\perp| && \text{to compute the intersection of all } M_i \end{aligned} \tag{5.2}$$

where $|P_i|$ is the number of tuples in partition P_i. Using the *Simple Complement* algorithm as algorithm $\mathcal{A}$, taking $k = |P_\perp|$ as the size of the NULL partition and estimating the size of a P_i by the average size $(n-k)/d$ of a partition (distributing $(n-k)$ tuples over d partitions), the overall runtime of the *Partitioning Complement* algorithm evaluates to $\mathcal{O}(n^2 2^m) + \mathcal{O}(nd) = \mathcal{O}(n^2 2^m)$, m being the size of the largest MCS. If we consider more than just one column for partitioning, the first term gets more complex, because all different NULL partitions need to be considered.

Algorithm 4: The baseline algorithm: *Simple Complement*.

Input: Relation R of tuples $t_i \in T$ (with attributes $a_j \in A$)
Output: Relation κ (R)

```
1: Create a list structure L to store all tree structures;
2: L ← ∅;
3: Create a tuple list TL to store all tuples;
4: TL ← ∅ ;
5: for all tuples t_i from relation R do
6:     HBUW_i ← ∅;
7:     Add t_i to TL;
8:     for all tree structures g_j ∈ L do
9:         Get g_j from L;
10:        Get t_j, the representative/root tuple of g_j, from TL;
11:        if t_i complements t_j then
12:            Create new tree structure g'_j ← ∅;
13:            for all tuple sets (paths) ts_k ∈ g_j do
14:                if all tuples from ts_k and t_i complement each other then
15:                    ts'_k ← new tuple set, copy of ts_k;
16:                    if ts_k ∈ HBUW_i then
17:                        Add t_i to ts'_k (marked);
18:                    else
19:                        Add t_i to ts'_k (unmarked);
20:                    end if
21:                    Add ts'_k to g'_j;
22:                    Add ts_k to HBUW_i;
23:                end if
24:                Add ts_k to g'_j;
25:            end for
26:            Replace g_j with new tree structure g'_j in L;
27:        end if
28:    end for
29:    Create a new tree structure g_i with t_i as representative;
30:    if |HBUW_i| > 0 then
31:        Mark t_i in g_i;
32:    end if
33:    Insert g_i into L;
34: end for
35: for all tree structures g_i in L do
36:    for all tuple sets (paths) ts_k ∈ g_i do
37:        if ts_k is maximal (path to leaf) in g_i and last node is marked then
38:            Compute t_c as the complement of all tuple in ts_k;
39:            Write t_c to the output;
40:        end if
41:    end for
42: end for
```

5.2.3 Employing Patterns of Null Values

A different way of computing *complementation* is the *Null-Pattern-Based Complement* algorithm. This algorithm partitions the input relation according to the patterns of tuples' NULL values and compares only tuples with complementing NULL patterns. It is a modification of the respective algorithm for removing subsumed tuples (see Section 5.1.4). We briefly describe the idea of the algorithm that has been developed in a joint project with Sascha Szott and others (Bleiholder et al., 2010a).

We first insert all tuples of the input relation into buckets according to the presence of NULL values in their attribute sequence: A tuple t with c attributes is inserted in bucket $B(n(t))$, where $n(t)$ is the bit sequence of length c such that $n(t)[i] = 0$ iff the value of t's i-th attribute is NULL.

Algorithm 5: The *Partitioning Complement* algorithm.

Input: Relation R of tuples $t_i \in T$ (with attributes $a_j \in A$)
Output: Relation κ (R)

```
1: Use schema S and statistics to choose partitioning column c in relation R;
2: Partition set of input tuples T from relation R by column c into d NON-NULL partitions and one NULL partition;
3: Let P_⊥ be the NULL partition, and the P_i's the d NON-NULL partitions;
4: Choose a complementation algorithm A to complement tuples;
5: Create list of d tuple sets M_i;
6: for all d NON-NULL partitions P_i do
7:     P'_i ← P_i ∪ P_⊥;
8:     P''_i ←κ (P'_i) using algorithm A;
9:     M_i ← ∅;
10:    for all tuples t' ∈ P''_i do
11:        if t'[c] ≠ ⊥ then
12:            Write tuple t' to the output;
13:        else
14:            Add t' to M_i;
15:        end if
16:    end for
17: end for
18: O ← ⋂_{i=1}^{d} M_i;
19: Write all tuples t_k in O to the output;
```

Afterwards, we sort the resulting non-empty buckets in order of decreasing number of zeros within their associated bit sequences. Let $B_1, \ldots, B_k$ be this ordering (note, that this ordering does not have to be unique, as more than one non-empty bucket with the same number of NULL values can exist).

We then start by considering the tuples in B_1 and compare them to the tuples in all other buckets $B_2, \ldots, B_k$ that have a NULL pattern that complements B_1's NULL pattern. For each tuple t in B_1 we build the complements (if tuples were found that complement t), insert the complements into their appropriate buckets, and, finally, delete t. If no complementing tuple is found, we output t. If a complement's bucket is not already present in $B_2, \ldots, B_k$, a new bucket is created and it is merged into the existing ordered sequence of buckets. As the number of NULL values of the complement is smaller than the number of t's NULL values, it is guaranteed that the complement is considered in a subsequent iteration. After having considered all tuples of B_1, we continue with B_2 and proceed until all non-empty buckets were considered.

Processing the buckets in this way saves runtime since we do not build complements twice: If $t_1 \gtrdot t_2$, the tuples cannot be contained in the same bucket. Therefore, the complement of t_1 and t_2 is only computed once. For example, if t_1's bucket is processed before t_2's one, the complement is only computed when t_1 is processed. When t_2 is processed, the complement is not computed again, since t_1's bucket is not considered (and thus t_1 is not found as a complementing tuple). We report on experimental results in Chapter 7.

5.2.4 Related Work on Complementation and Complement Union

Related work on conflict resolution, data fusion and other general approaches, techniques, and systems is widely covered in Chapter 9, briefly including complementation where applicable. More specific related work to the definition and implementation of *complementation* and *complement union* is covered in the following.

Data fusion with the semantics of *complement union* has been first considered in (Bleiholder et al., 2010a) and (Bleiholder et al., 2010b). The *complementation* operator and *complement union* are defined (see also Section 4.3) and implementation details are given. However, similar concepts have been previously explored. Replacing complementing tuples by their complement in a relation is equivalent to finding all maximal cliques in a graph that has been constructed by creating one node per tuple and an edge between nodes if the corresponding tuples complement one another. Standard algorithms for finding all maximal cliques are described in (Bron and Kerbosch, 1973; Makino and Uno, 2004), although (Stix, 2004) improves on that on dynamic graphs by introducing edge weights. Work described in (Modani and Dey, 2008) enumerates all

cliques of a minimum size. Algorithms that solve the dual problem of finding independent sets (Johnson et al., 1988) can also be looked at. In the experiments section (see Chapter 7) several different implementation variants for finding cliques and independent sets are compared to our algorithms.

5.3 Conflict Resolution and Data Fusion

Whereas Section 4.4 defined the two operators for *conflict resolution* and *data fusion*, this section now gives a brief overview on some implementation issues of these two operators. In particular it presents algorithms to realize the two operators in a database system. First, an implementation of the *conflict resolution* operator is described in Section 5.3.1, before an implementation of the *data fusion* operator is described in Section 5.3.2. Implementations of the two operators combine already known algorithms in order to accomplish the two operations.

5.3.1 Implementing Conflict Resolution

During conflict resolution, a set of conflict resolution functions is applied on an ordered set of tuples (see Definition 23 on page 52). The basic algorithm is given as Algorithm 6. In a first step, if the set of tuples is not already ordered, the tuple set T is ordered according to a set of sorting attributes O. Then, attribute values are separated: all values from each attribute a_j are held in a separate list L_j. Each conflict resolution function f_k is then provided with the necessary value list L_k to compute the resolved value. If the function needs additional information, e.g., other attributes, this information is also provided. All resolved values are combined to form result tuple t_r which is finally written to the output.

Algorithm 6: The baseline algorithm for *conflict resolution*.

Input: Set of n tuples $T = \{t_i\}$ (with attributes $a_j^I \in A^I$)
Input: Set of m conflict resolution functions $CR = \{f_j\}$ (with assigned attributes $a_k^O \in A^O$, defining output attributes $A^O \subseteq A^I$), one f_k for each a_k^O
Input: Set of sorting attributes $O \subseteq A^I$
Output: Result tuple $t_r \leftarrow \lambda_{CR,S}(T)$ (with attributes $a_k^O \in A^O$)

1: Order tuples t_i from T by O;
2: Create ordered lists $L_j \leftarrow \varnothing$, one for each attribute $a_j^I \in A^I$;
3: **for all** tuples $t_i \in T$ **do**
4: **for all** attributes a_j^I of t_i **do**
5: Add value $t_i[a_j^I]$ to the end of L_j;
6: **end for**
7: **end for**
8: Create result tuple $t_r \leftarrow \varnothing$ (with attributes $a_k^O \in A^O$);
9: **for all** conflict resolution functions f_k **do**
10: Feed L_k (and additional information, e.g., additional L_j that are needed by f_k) into f_k and get result value $v_k \leftarrow f_k(L_k, \ldots)$;
11: $t_r[a_k^O] \leftarrow v_k$;
12: **end for**
13: Write t_r to the output;

Computational complexity of the conflict resolution algorithm is as follows: Sorting the tuple set can be done in $\mathcal{O}(n \log n)$ and the subsequent step has a computational complexity of at least $\mathcal{O}(nk)$ for the rearrangement of the input, where n is the number of conflicting values and k is the number of attributes. Let's assume that conflict resolution functions only need to look once at each conflicting value, which is certainly true for the simple functions, such as MIN, or MAX. This results in a computational complexity of $nk\mathcal{O}(1) = \mathcal{O}(nk)$ for the execution of all f_k. More complex functions such as MOST GENERAL CONCEPT may result in a higher computational complexity. Assuming $\mathcal{O}(\mathrm{crf})$ as the computational complexity of the execution of the conflict resolution functions, the overall computational complexity is the maximum of $\mathcal{O}(n \log n)$, $\mathcal{O}(nk)$, and $\mathcal{O}(\mathrm{crf})$.

There is some improvement possible. In the baseline algorithm, conflict resolution functions are applied on the whole set of conflicting values. There exists another way of applying conflict resolution functions: the value can be computed by successively feeding one value after another into the function and keeping an intermediate result. E.g., the maximum of a set of values can be computed by computing the maximum of the first and second value, then by computing the maximum of this value and the third value, etc. This is called the *online* version of the algorithm, as opposed to the *offline* version as described above. A function that can be computed that way is called an *online* function. However, not all functions are computable in this fashion. The online version saves the work of rearranging values in the lists L_j and allows to feed the attribute values directly into the conflict resolution functions. For functions such as FIRST this may save considerable runtime, as only the first tuple needs to be considered. However, this *online* version of conflict resolution is only possible, if all functions f_j from set CR are *online* functions.

5.3.2 Implementing Data Fusion

The basic data fusion implementation is shown as Algorithm 7 on page 75. It combines the two operations *conflict resolution* and *subsumption* with a grouping, in order to accomplish its goal of fusing different representations of same real-world objects. As a first step, the input relation R is divided into groups, using a grouping algorithm $\mathcal{G}$. Then, for each group – which will later result in one single tuple – we first remove exact duplicates. In case of the data fusion variant with removal of subsumed tuples we then also remove subsumed tuples from each group using a subsumption algorithm $\mathcal{S}$. Last, we apply the conflict resolution operator $\lambda_{CR,S}$ on the group. The tuple that is returned is written to the output.

Algorithm 7: The baseline algorithm for *data fusion*.

Input: Relation R, as set of tuples $T = \{t_i\}$ (with attributes $a_j \in A$)
Input: Set of identifying attributes $F = \{id_k\} \subseteq A$
Input: Set of conflict resolution functions $CR = \{f_j\}$
Input: Set of sorting attributes $S \subseteq A$
Output: Relation $\phi_{F,CR,S}$ (R) (or $\phi^{\beta}_{F,CR,S}$ (R) respectively)(with attributes $a_j \in A$)

```
1: Group input relation R according to F using grouping algorithm G;
2: for all Groups g_i formed by G on R do
3:     Remove exact duplicates from g_i, using a distinct algorithm D;
4:     if operator is φ^β_{F,CR,S} then
5:         Remove subsumed tuples from g_i, using subsumption algorithm S;
6:     end if
7:     Compute result tuple t_i ←λ_{CR,S} (g_i) for group g_i using Algorithm 6;
8:     Write tuple t_i to output;
9: end for
```

Computational complexity depends mainly on the algorithms chosen for the different parts of the data fusion algorithm: on the complexity of the grouping algorithm $\mathcal{G}$, the complexity of the subsumption algorithm $\mathcal{S}$ and the complexity of the conflict resolution algorithm (here, especially the complexity of the conflict resolution functions). Given that *grouping* (based on *sorting*), *deduplication* (based on *sorting*) and *subsumption* (NULL pattern based) each do not exceed a runtime of $\mathcal{O}(n \log n)$, then overall runtime is the maximum of $\mathcal{O}(n \log n)$ and the complexity of conflict resolution (maximum of $\mathcal{O}(n \log n)$, $\mathcal{O}(nk)$, and $\mathcal{O}(\text{crf})$).

When implementing *data fusion* it is important to know the different alternative implementation choices for its parts (e.g., *sort-based grouping*, *hash-based deduplication*, different variants of *subsumption*, etc.) and come up with a good combination of the algorithms that it consists of. When implementing data fusion as part of the research systems HUMMER and FUSEM we have been able to rely on a variety of implementation variants for *grouping*, *sorting*, and *distinct*, supported by the XXL framework. Finding a good combination is then done by the optimizer, given cost formulas for the parts and the data fusion algorithm frame. If the optimizer is able to estimate the cost, it can easily determine the best combination. Cost formulas for the standard implementations for *grouping*, *sorting*, and *deduplication* are well known in the literature (Ullman et al., 2001). Additionally, we briefly give some possible cost formulas for *subsumption*, *complementation* and

data fusion in the next chapter, in Section 6.3.

5.4 Data Fusion in Relational DBMS

So far we defined operators for data fusion and gave implementations of these operator for direct use (as independent operators) in a database management system. This section considers data fusion operators expressed using standard SQL. We first cover SQL rewritings for *subsumption* and *complementation* and then consider the *data fusion* operator and to what degree it is possible to express it using standard SQL constructs alone. Standard SQL as used in the next sections will include *selection*, *projection*, *join*, *union*, *difference*, *grouping*, *aggregation* and the data warehouse extensions of SQL 2003 (mainly the windowing techniques) (Eisenberg et al., 2004; ISO/IEC 9075-*:2003, 2003). We focus on the SQL dialect that IBM's DB2 implements. Other vendors with different dialects may vary.

5.4.1 Subsumption and Complementation in RDBMS

As we see in the following, *subsumption* can be rewritten as an SQL statement in two different ways: first, using a correlated subquery and second, using the data warehouse extensions of SQL. Considering a correlated subquery, we can easily translate the conditions from the definition of tuple subsumption (see Definition 16 in Section 4.2) into conditions of the subquery. In (Kruppa, 2005) the implementation of *minimum union* (and of *subsumption*) as an SQL query and a DB2 table function is explored in detail, which we summarize here.

For a relation BASETABLE with k columns a_1, a_2, ..., a_k, the SQL statement to produce a table where all subsumed tuples are removed from BASETABLE is the following:

```
SELECT DISTINCT  a1, a2, ..., ak
FROM BASETABLE as R2
WHERE NOT EXISTS (
 SELECT  a1, a2, ..., ak
 FROM BASETABLE AS R1
 WHERE
 (
       (R2.a1 = R1.a1 OR R2.a1 IS NULL)
   AND (R2.a2 = R1.a2 OR R2.a2 IS NULL)
   ...
   AND (R2.ak = R1.ak OR R2.ak IS NULL)
 ) AND (
      (R2.a1 IS NULL AND R1.a1 IS NOT NULL)
   OR (R2.a2 IS NULL AND R1.a2 IS NOT NULL)
   ...
   OR (R2.ak IS NULL AND R1.ak IS NOT NULL)
 )
)
```

The main idea is to select all tuples t_2 where no other tuple t_1 exists such that t_2 is subsumed by t_1 ($t_2 \sqsubset t_1$). So we need to show that the tuples that are selected by the correlated subquery do in fact subsume the tuple from the main query. In order to show this we denote the tuple from the main query by t_2 and the tuples from the correlated subquery by t_1. The first condition that t_2 and t_1 need to be defined on the same schema is trivially satisfied. In terms of the tuples t_1 and t_2, the condition of the subquery can be expressed as follows:

$$\left(\bigwedge_{i=1}^{k} t_2.a_i = t_1.a_i \vee t_2.a_i = \bot\right) \wedge \left(\bigvee_{i=1}^{k} t_2.a_i = \bot \wedge t_1.a_i \neq \bot\right)$$

The first part on the left models the condition of subsumption that in all NON-NULL attributes of t_2, the values of t_2 and t_1 coincide. It also follows that t_1 does not have more NULL values than t_2. The second part on the right models the condition that there is at least one attribute where t_2 has a NULL value and t_1 has

not. It follows that t_2 has more NULL values than t_1 which is the last condition from the definition of tuple subsumption.

The statement is generally applicable and its complexity is linear in the number of attributes of the table. However, the statement always involves a (self-) join, regardless of the number of attributes. Computational complexity of the statement depends on the join algorithm and other techniques that may be used by the database optimizer to speed up computation. As we will see further on in the experiments section (see Chapter 7), this generally applicable statement does not scale well.

A more efficient SQL rewriting is proposed by (Rao et al., 2004) using the data warehouse extensions of SQL. The main idea is to sort the relation BASETABLE with k columns a_1, a_2, ..., a_k in such a way that a tuple t_1 that subsumes another tuple t_2 is sorted next to it. Then, using a windowing technique, only tuples that lie next to each other need to be compared and, if necessary, kept in the output or thrown out. We denote an ordering by columns a_1, a_2, ..., a_k that accomplishes this as a favorable ordering `fav`. As long as such a favorable ordering exists, the following SQL statement removes subsumed tuples:

```
SELECT * FROM (SELECT a1, a2, ..., ak,
min(a1) OVER (ORDER BY fav ROWS BETWENN 1 PRECEEDING AND 0 FOLLOWING) AS a1_p,
min(a2) OVER (ORDER BY fav ROWS BETWENN 1 PRECEEDING AND 0 FOLLOWING) AS a2_p,
...
min(ak) OVER (ORDER BY fav ROWS BETWENN 1 PRECEEDING AND 0 FOLLOWING) AS ak_p,
rownumber() OVER (ORDER BY fav) AS rownum
FROM (SELECT DISTINCT a1, a2, ..., ak FROM BASETABLE))
WHERE rownum=1
OR a1 <> a1_p
OR a2 <> a2_p
...
OR ak <> ak_p
```

The complexity of the statement is linear in the number of attributes of the table and as we see later on in the experiments section (see Chapter 7) it scales well and results in comparable runtimes to the specialized algorithms for *subsumption*. Nevertheless it is not applicable in the general case, as there may be datasets for which the needed favorable ordering does not exist.

Concerning a rewriting for *complementation*, we take advantage of the equivalence of the problem of complementing tuples and the problem of finding cliques. Such a rewriting is much more complicated and slightly different for different DBMS, because it involves recursive queries that are implemented and supported differently in different DBMS. In the following a brief sketch of such an implementation is given. In a first step a table GRAPH is created with two columns A and B that holds the adjacency list (all edges) of the graph that represents all complementation relationships. Tuple identifiers are used for the input table and the tuple $(2, 3)$ is included in GRAPH if the two tuples with *id* 2 and 3 complement each other. An edge is inserted only once, with the A value being less than the B value. (In the example the tuple $(3, 2)$ would not be insert.) This can easily be done by self-joining the table and putting the conditions for two tuples to complement each other in the where clause. In the rewriting, the number of NULL values of a tuple is given in an additional column (as NULLCOUNT). It can easily be computed on-the-fly from the base table. The rewriting for a relation BASETABLE with k columns $a_1, a_2, \ldots, a_k$, a column `nullcount`, and a tuple identifier `id` is given in the following:

```
WITH COMPLEMENT (X,Y) AS (
SELECT A.id AS X, b.id AS Y
FROM BASETABLE AS a, BASETABLE AS b
WHERE
(
    ((A.a1 IS NULL AND B.a1 IS NOT NULL)
      OR (A.a1 IS NOT NULL AND B.a1 IS NULL) OR A.a1=B.a1)
AND ((A.a2 IS NULL AND b.a2 IS NOT NULL)
      OR (A.a2 IS NOT NULL AND B.a2 IS NULL) OR A.a2=B.a2)
      ...
AND ((A.ak IS NULL AND B.ak IS NOT NULL)
```

```
      OR (A.ak IS NOT NULL AND B.ak IS NULL) OR A.ak=B.ak)
) AND A.nullcount>0 AND B.nullcount>0 AND A.id<>B.id AND A.id<B.id
)
```

The resulting table Complement has two attributes X and Y and holds the complementation relationships of table Basetable. The next step in the process is to find cliques on this adjacency table. A rewriting is documented on the web for the Oracle DBMS. It involves recursive queries and a clever use of the `LIKE` operator to find all cliques in the adjacency table Complement (Oracle Forums, 2010). As IBM's DB2 uses a slightly different mechanism to express recursive queries, the statement can not be executed right away. Translating the rewriting into the DB2 dialect fails because of the different semantics of the `LIKE` statement: DB2 does not allow for two columns to be compared by `LIKE`. However, in newer versions, DB2 also understands the Oracle constructs for hierarchical queries used in the rewriting. So, beginning with Version 9.7. the statement above can also be executed in IBM's DB2.

5.4.2 Conflict Resolution and Data Fusion in RDBMS

Some parts of the *conflict resolution* and *data fusion* operators can be expressed by means of standard Sql's grouping and aggregation functionalities and using the data warehouse extensions. However, not all aspects of these two operators can be expressed, unless we allow for the use of user-defined aggregation functions. In that case, we could express almost all data fusion operations, as long as we can define the conflict resolution functions as a user-defined aggregation function and are able to incorporate it into the database management system. Unfortunately, user-defined aggregation is not a standard feature in RDBMS and is rarely supported (see also Section 9.1.2).

Conflict resolution is seen as the application of a function on a set of values (similar to aggregation). The data fusion operator additionally includes a grouping according to a specific set F of identifying attributes. The two variants of the *data fusion* operator (ϕ and ϕ^{β}, without and with removal of subsumed tuples inside groups) are not equal unless the columns that are used for the grouping do not contain any null values. The following rule holds, if the columns in F do not contain null values:

$$\phi^{\beta}_{F,CR,S}\,(\mathrm{R}) = \phi_{F,CR,S}\,(\beta\,(\mathrm{R}))$$

This equivalence is caused by all null values in a grouping column being combined into their own group. If a tuple from R contains a null in one of the F columns, and such a tuple is subsumed by another one, it does not form its own group when removing subsumed tuples beforehand instead of in the groups. Therefore, the variant of data fusion that also removes subsumed tuples in the groups can only be rewritten using Sql if the set of identifying columns does not contain null values. We see this rule again when considering rewriting rules for optimization data fusion queries. The rewriting for subsumption (see Section 5.4.1) is used on the input and serves as input to the rewriting of the normal variant of the data fusion operator described in the following. If *outer union* is used to combine base relations as input for data fusion, we use the following straight-forward statement to create an intermediate relation, the *outer union* of two sources:

```
WITH OUTER_UNION (ID, A1, ..., An, B1, ..., Bm, C1, ..., Ck, Q) AS
(
  SELECT ID, A1, A2, ... An,
   B1, B2, ..., Bm,
   NULLIF('','') as C1, NULLIF('','') as C2, ..., NULLIF('','') as Ck,
   '1' AS Q FROM SRC_1
UNION
  SELECT ID
   NULLIF('','') as A1, NULLIF('','') as A2, ..., NULLIF('','') as An,
   B1, B2, ..., Bm,
   C1, C2, ..., Ck,
   '2' AS Q FROM SRC_2
)
```

The statement easily extends to more than only two sources. To later be able to sort by source, or to use conflict resolution functions that need information which source a tuple comes from, we include a source identifier in the result of *outer union* as column Q. In the statement, attributes A_i are from source 1 only, attributes B_j are common to both sources and attributes C_k are from source 2 only. The schema of the result of *outer union* is the union of the schemata of the sources and the `NULLIF` function is used with IBM's DB2 for padding with NULL values. Intermediate tables are created via SQL's `WITH` statement and are called common table expressions (CTE) in IBM's DB2.

After subsumed tuples have been removed from the input, the remaining part of the *data fusion* operator $\phi_{F,CR,S}$ is rewritten in the following three steps:

- **Divide the Input.** First, the input table is divided into several different tables: one table for each of the conflict resolution functions f_i in CR. The attribute Q holding source information and the identifying attributes F are also kept in these tables, to be able to later combine the resolved values. If a conflict resolution function requires additional attributes these are also included. See also the transformation rules in Section 6.2.3 on how to divide *data fusion* into parts and recombine it later.
- **Apply Conflict Resolution.** Second, for each attribute (each table), the conflict resolution function is applied on the corresponding table. Further down SQL rewritings are given for some basic functions.
- **Combine Results.** Last, after all conflicts have been resolved the fused results are combined into one clean representation. See also the transformation rules in Section 6.2.3 on how to divide data fusion into parts and recombine it later. The final statement can then be applied on the data and results in the application of the respective data fusion operator.

Divide the Input For each attribute where conflicts are to be resolved, a column view is created, an intermediate table with all necessary attributes. Necessary attributes are 1) the attribute where conflicts are to be resolved, 2) the identifying (ID) attributes from F, 3) the source attribute Q from the input view, and finally 4) all other attributes that are needed by the conflict resolution function. The following statement is used to split the input relation (*outer union*) and to create the column views:

```
WITH ID_VIEW (ID) AS (
 SELECT DISTINCT ID FROM OUTER_UNION
),
A1_VIEW (ID, A1, Q) as (
 SELECT ID,A1, Q FROM OUTER_UNION
),
A2_VIEW (ID, A2, Q) as (
 SELECT ID, A2, Q FROM OUTER_UNION
),
...
Ak_VIEW (ID, Ak, Q) as (
 SELECT ID, Ak, Q FROM OUTER_UNION
)
```

The first view in the list that only holds ID's is used to recombine the fused tuples later on in the third step. In all views, information on the source, where the value is coming from, is included as column Q.

Apply Conflict Resolution For each column view a resolved view is declared next, that applies the corresponding resolution function on the column view. In the following such a resolved view is shown for different resolution functions.

The application of standard aggregation functions (MAX, MIN, COUNT, ...) as conflict resolution functions is straight-forward and shown first. In the following, statements for slightly more complicated resolution functions are also provided. Please bear in mind that in this section we do not consider the use of user-defined aggregation functions, but try to implement the functions by the use of SQL. Implementing the resolution

functions as user-defined aggregation functions would otherwise follow the pattern of the first statement for standard aggregation functions. Standard aggregation functions are applied to a column view in the following way:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT T.ID,
 f(T.A1) as RESOLVED_A1
FROM (
 SELECT ID, A1, Q
 FROM A1_VIEW
 ORDER BY ID,Q
) AS T
GROUP BY T.ID
)
```

In the statement defining the view, function f is one of MAX, MIN, COUNT, SUM, AVERAGE, STANDARD DEVIATION, or VARIANCE. As all these functions are *order-independent*, the ordering by Q is actually not necessary, but included to match the pattern of the other statements.

The next statements, defining various resolved views, use the data warehouse extensions of SQL 2003. These extensions introduce among other things a windowing mechanism and allow to influence the order inside a grouping. The two functions FIRST and LAST are *order-dependent* and their resolved view is defined as follows:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID,A1,
 ROWNUMBER() OVER (PARTITION BY ID) AS R
 FROM A1_VIEW
 ORDER BY ID,Q
) AS T
WHERE T.R=1
)
```

For each different ID, all existing rows are numbered using the `ROWNUMBER()` function and the corresponding partitioning. Selecting all rows with row number 1 returns the first value for each ID according to the specified ordering. This way the view definition above implements the FIRST function. The next view definition implements the LAST function:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID,A1,
 max(ID) OVER (ORDER BY ID ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING) as NEXTID
 FROM A1_VIEW
 ORDER BY ID,Q
) AS T
WHERE ID<>NEXTID OR NEXTID IS NULL
)
```

The view returns the last value according to the specified ordering. Please keep in mind that if a sortorder is specified in the S parameter of ϕ, the corresponding attributes need to be included in `A1_VIEW` and the `ORDER BY` clauses accordingly. The idea behind the statement for the LAST function is to store with each ID also the ID (as `NEXTID`) of the following tuple (according to the sort ordering). The *last* tuples for each ID are all tuples where `NEXTID` is different to ID or – if it is the last tuple at all – `NEXTID` is NULL. The view definition to implement the COALESCE function is based on the statement for the FIRST function:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID,A1,R,
 ROWNUMBER() OVER (PARTITION BY ID) AS RR
 FROM (
  SELECT ID,A1,Q,
  ROWNUMBER() OVER (PARTITION BY ID) AS R
  FROM A1_VIEW
  ORDER BY ID, Q
 ) AS T
 WHERE (A1 IS NOT NULL)
) AS TT
WHERE RR=1
)
```

Here, a partitioning and a row numbering are nested twice, the inner one filtering out all NULL values and the outer one thus returning the first NON-NULL value. If there are only NULL values available, the *ID* value is discarded. However, in the process of combining everything later on in the third step, a NULL value is reinserted, thus guaranteeing the correct semantics of the COALESCE function.

The next two view definitions compute results for LONGEST and SHORTEST. As both functions are *order-independent*, ordering by source and/or sort order *S* is not necessary. The view for the LONGEST function is defined as follows:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID, A1, L,
 ROWNUMBER() OVER (PARTITION BY ID) AS R
 FROM (
  SELECT ID, A1,
  CASE WHEN A1 IS NULL THEN 0 ELSE LENGTH(A1) END AS L
  FROM A1_VIEW
  ORDER BY ID, L DESC, A1
 ) AS TT
 ORDER BY ID, L DESC, A1
) AS T
WHERE T.R=1
)
```

The view definition for SHORTEST is the same, only the ordering by *L* is in ascending order instead of in descending order. The main idea is to first introduce an intermediate column that holds the length of each attribute value. Then, a partitioning by *ID* and ordering in ascending (shortest) or descending (longest) order ensures the correct semantics of the SHORTEST or LONGEST function.

The idea behind the view definition for the VOTE function is similar. Instead of ordering by length, values are ordered by the number of occurrences of a specific value. The defining statement is as follows:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID,A1,
 count(A1) as cnt,
 ROWNUMBER() OVER (PARTITION BY ID) AS R
 FROM A1_VIEW
 GROUP BY ID, A1
 ORDER BY ID, CNT DESC
) AS T
```

```
WHERE T.R=1
)
```

There are two functions exhibiting source preferences: CHOOSE which chooses only values from one specific source and PREFER which prefers value from one *preferred* source over values from *non-preferred* sources. Considering CHOOSE, only values from a specific source are chosen. If there are values from other sources for an *ID*, they are discarded. In this case, an *ID* may end up with a NULL value. We resolve conflicts by the following view:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID,A1,Q,
 ROWNUMBER() OVER (PARTITION BY ID) AS R
 FROM A1_VIEW
 WHERE Q='<source-ID>'
 ORDER BY ID, Q, A1
) AS T
WHERE T.R=1
```

The PREFER function prefers values from a specific source over values from the other sources for a specific *ID*. If there is no value from the preferred source, but values from other sources, these are taken instead. In a first step a column is introduced that indicates if the tuple comes from a preferred or non-preferred source. tuples are ordered in such a way that preferred tuples come first, are partitioned by ID and then the first value is taken:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID,A1
FROM (
 SELECT ID,A1,Q,P,
 ROWNUMBER() OVER (PARTITION BY ID) AS R
 FROM (
  (
   SELECT ID,A1,Q,'<preferred>' AS P
   FROM A1_VIEW
   WHERE Q='<source-ID>'
  )
  UNION
  (
   SELECT ID,A1,Q,'<nonpreferred>' AS P
   FROM A1_VIEW
   WHERE Q<>'<source-ID>'
  )
 ) AS T
 ORDER BY ID, P DESC, A1
) AS G
WHERE G.R=1
)
```

Another function that is easy to realize is the CHOOSE CORRESPONDING function. In order to realize it, the view definition of the function from the other (corresponding) column is used and the column name is added to the SELECT lists.

The statement for the CHOOSE DEPENDING function takes as resolved value for each *ID* the value that has the value <v> in the column <other_col>. In addition, as values are ordered by source column *Q*, source preference is implicitly included. The function is realized as follows:

```
WITH RESOLVED_A1_VIEW (ID, RESOLVED_A1) AS (
SELECT ID, A1
FROM (
 SELECT ID,A1,<other_col>,Q,
 ROWNUMBER() OVER (PARTITION BY ID) AS R
 FROM A1_VIEW
 WHERE <other_col>='<v>'
 ORDER BY ID, Q
) AS T
WHERE T.R=1
)
```

Combine Results. At the end, the resolved values for each column and *ID* need to be recombined to form one single table. This can be done in two ways. It can be done either by using a *full outer join* on all resolved views using the identifying attributes (including the *ID* view, in order to prevent NULL tuples from disappearing) as join attributes, or by an *outer union* followed by a simple *aggregation* in order to keep only one resolved value per *ID*. The statement for the latter is given in the following. We consider these two ways of combining results again in the section on transformation rules (see Section 6.2.3). The combination statement is the main query after the definition of all intermediate views:

```
SELECT ID, max(RESOLVED_A1) AS A1, max(RESOLVED_A2) AS A2, ... max(RESOLVED_Ak) AS Ak
FROM (
 SELECT ID, A1 AS RESOLVED_A1, NULLIF('','') AS RESOLVED_A2,
    ... NULLIF('','') AS RESOLVED_Ak FROM RESOLVED_A1_VIEW
  UNION
 SELECT ID, NULLIF('','') AS RESOLVED_A1, A2 AS RESOLVED_A2,
    ... NULLIF('','') AS RESOLVED_Ak FROM RESOLVED_A2_VIEW
  ...
  UNION
 SELECT ID, NULLIF('','') AS RESOLVED_A1, NULLIF('','') AS RESOLVED_A2,
    ... Ak AS RESOLVED_Ak FROM RESOLVED_Ak_VIEW
) AS T
GROUP BY ID
```

As only one resolved value exists per *ID* and per column, the MAX function returns this resolved value, or a NULL value, if no value exists.

A good plan, violently executed now, is better than a perfect plan next week.

(George S. Patton)

6

Optimizing Data Fusion Operations

So far, we have seen how data fusion fits into an integration framework, we have defined data fusion operations and have shown how they can be executed. We now leave the more technical area of algorithms and operator implementation and look at data fusion as an operator that fits into the world of – relational – operators. In Section 6.1 we describe typical query plans in the area of information integration, describe how our operators fit in and give a look at typical data fusion queries and their representation as query trees. We then explain how data fusion operators fit into logical query optimization: in Section 6.2 we define the search space of possible operator trees for optimization by giving transformation rules involving the newly defined operators. In Section 6.3 we show how to include the operators in physical optimization by estimating query costs and selectivities of operators.

6.1 Data Fusion Queries

In the following we use the term *data fusion queries* to denominate queries that involve data fusion operations in general and especially the operators that have been defined in the last chapters: *subsumption* and *minimum union* (β and $\oplus$), *complementation* and *complement union* (κ and $\boxplus$), and *conflict resolution* and *data fusion* (λ and ϕ). This is similar in spirit to the use of the term *fusion queries* in (Yerneni et al., 1998), where a *fusion query* combines information using *union* from distributed source relations. Whereas (Yerneni et al., 1998) only allows for *union*, *selection*, and the *projection* of only a single attribute in a single query, we extend the notion of *fusion queries* to also include other standard relational operators (such as *join* and *grouping*/*aggregation*), and the newly introduced operators mentioned above. We also extend *union* to *outer union*. However, the main characteristics that data from sources is combined by *union*/*outer union* stays the same. By introducing more sophisticated operators we especially allow for the handling of uncertainties and conflicts in the data.

To introduce *data fusion queries* we consider the scenario of integrating data from two bookstores, where one bookstore stores its fiction books in relation BOOKS_1 and it's non-fiction books in relation BOOKS_2. As a book is either fiction or non-fiction, there is no overlap between the relations, however, duplicate entries in one relation are still possible. Authors are stored in an additional relation AUTHOR. The second bookstore stores its books (authors included) in relation BOOKS_3. We consider as a typical task to integrate the book data from both bookstores, to remove subsumed tuples, combine complementing tuples and select only books that are cheaper than 10 EUR and have been written by Ernest Hemingway. Such a task can be expressed by a *data fusion query* and visualized by a query plan.

A logical query plan is depicted as a tree where the nodes of the tree are the logical query operators and edges stand for the data flow between the operators (see e.g., (Ullman et al., 2001) for a detailed explanation of query plans). Figure 6.1 depicts two different query plans that can be used in answering above question. The plan on the left (Plan A) first joins the BOOKS relations from the first bookstore with the AUTHOR relation, then combines all three BOOKS relations by *outer union* and subsequently removes subsumed tuples, combines complementing tuples and selects the desired books. In contrast, Plan B on the right pushes the *selection*s as far down as possible, and removes subsumed tuples early in the process.

We show both plans to illustrate that there are always multiple ways of integrating sources and to motivate for

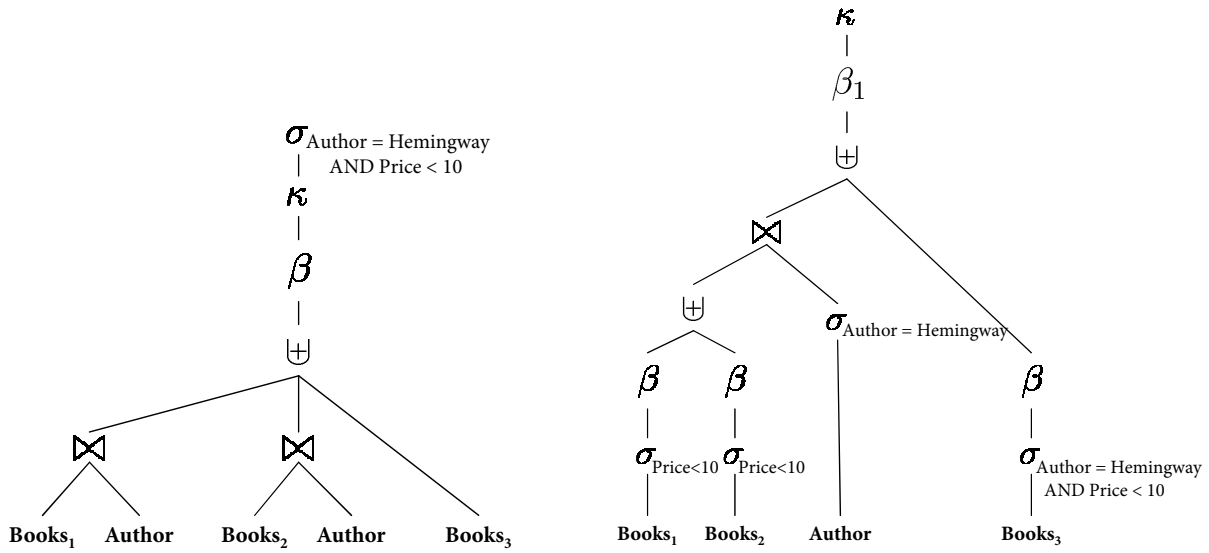


Figure 6.1: Two equivalent query plans for a data fusion query.

the next sections that introduce transformation rules that can be used to transform Plan A into the equivalent Plan B. Physical query optimization can then estimate the cost of both plans and pick the better one. The associated expressions of relational algebra are the following two:

Plan A:

$$\sigma_{A \wedge P} \left(\kappa \left(\beta \left(\left(\textsc{Books}_1 \bowtie \textsc{Author}\right) \uplus \left(\textsc{Books}_2 \bowtie \textsc{Author}\right)\right)\right) \uplus \textsc{Books}_3\right) \tag{6.1}$$

Plan B:

$$\kappa \left(\beta_1 \left(\left(\left(\beta \left(\sigma_P \left(\textsc{Books}_1\right)\right) \uplus \beta \left(\sigma_P \left(\textsc{Books}_1\right)\right)\right) \bowtie \left(\sigma_A \left(\mathrm{A}\right)\right)\right) \uplus \left(\beta \left(\sigma_{A \wedge P} \left(\textsc{Books}_3\right)\right)\right)\right)\right) \tag{6.2}$$

with conditions A being *Author='Hemingway'* and P being *Price<10*. We use this notation (expressions of relational algebra) in describing the transformation rules. A complete list of operators and their symbols as used in this thesis can be found in the Glossary.

6.2 Defining the Search Space

Up to this point, we considered efficient physical implementations of operators for data fusion (Chapter 5). We now focus on optimization at the logical level, i.e., how operators can be moved in a logical query plan to increase efficiency. In the following we present sets of transformation rules that can be used by an optimizer to create a search space of equivalent query plans that can then be searched for the most efficient one. We focus on equivalent transformations that do not change the result of the query. The main advantage of rewriting a *data fusion query* is to reduce the size of the input of the data fusion operators, as the cost of most data fusion operators primarily depends on the input size.

We focus on the basic operators *subsumption* (β), *complementation* (κ), and on the *data fusion* operators (ϕ, ϕ^{β}), and how they can be moved around in combination with other relational operators that are commonly used in data integration, namely *union*, *selection*, *projection*, and *join*. We also briefly discuss some other operators such as *grouping/aggregation*, *sort*, and *distinct*. Unless otherwise noted we assume set semantics. We first give transformation rules for *subsumption* and *complementation*, and then present transformation rules for the *data fusion* operator.

6.2.1 Transformation Rules for Subsumption and Minimum Union

We consider equivalent transformation rules for moving the *subsumption* (β) and the *minimum union* ($\oplus$) operator around in query plans. The focus in this chapter is on *subsumption* because *minimum union* can be moved around in a tree by splitting the operator into its components *subsumption* and *outer union* and move these two components around separately. We report on transformation rules for *subsumption* only and refer to the literature for the standard *union* transformation rules (e.g., (Ullman et al., 2001)). Table 6.1 at the end of this Section on page 90 summarizes the rewriting rules for *subsumption*.

Combinations with Outer Union *Minimum union* is defined as the combination of *outer union* with the subsequent removal of subsumed tuples (see Section 4). We examine the exchangeability of *outer union* and *subsumption*. As there may be subsuming tuples across sources, simply pushing the *subsumption* operator through *outer union*s is not possible without leaving an additional *subsumption* operation on top of the *outer union*. So the following rules hold:

$$\beta\ (\mathrm{A} \uplus \mathrm{B}) = \beta\ (\mathrm{A}) \uplus \beta\ (\mathrm{B}) \tag{6.3}$$

if there are no subsumed tuples across source relations, and

$$\beta\ (\mathrm{A} \uplus \mathrm{B}) = \beta_1\ (\beta\ (\mathrm{A}) \uplus \beta\ (\mathrm{B})) \tag{6.4}$$

in all other cases.

We use β_1 for the *subsumption* on top of the *outer union* to denote that this does not need to be a full version of the *subsumption* operator as we only need to test for subsumed tuples across sources. To compute β_1, let P_1 be the set of attributes and values private to t_1, i.e., all $(a_j, v_j) \in P_1$ are in t_1 but not in t_2. Let P_2 analogously be the set of attributes and values private to t_2, and let C_1 and C_2 be the sets of attribute/value combinations with attributes common to both tuples. For two tuples t_1, t_2 from two source relations, the following condition needs to hold in order for t_1 to be subsumed by t_2: $P_1 = \varnothing \wedge (C_1 \subset C_2 \vee (C_1 = C_2 \wedge P_2 = \varnothing))$. We use this condition in implementing β_1, a version of the *subsumption* operator that only removes subsumed tuples across sources.

Combinations with Projection. A standard technique in relational optimization is to early project out attributes that are not needed later on. For *subsumption* potentially all attributes are needed to decide if one tuple subsumes another. So, in general, it is not possible to introduce arbitrary *projection*s below a *subsumption* operator without changing the final result. However, a *projection* can be inserted, if it projects out columns that do not affect the removal of subsumed tuples. An example for such a column would be a column that contains only one single value. Another example would be a column that has the same value for all pairs of subsuming tuples. Unfortunately, testing the latter property has the same complexity as removing subsumed tuples itself. This leads us to the following conclusion: In general, we cannot push a *projection* down through a *subsumption* nor introduce a *projection* before a *subsumption*, i.e.,

$$\beta\ (\mathrm{A}) \neq \beta\ (\pi\ (\mathrm{A})) \tag{6.5}$$

$$\pi\ (\beta\ (\mathrm{A})) \neq \beta\ (\pi\ (\mathrm{A})) \tag{6.6}$$

Combinations with Selection. In this paragraph we examine the interchangeability of β and σ with the main goal of pushing *selection*s through *subsumption* to decrease the input cardinality of the *subsumption* operator. As *subsumption* deals with the removal of NULL values, in case the *selection* is applied on a column c without NULL values (e.g., a key, a unique column, or a NOT NULL column), we can indeed push *selection* down through the operator (see Rule 6.8). If column c allows NULL values, pushing *selection* through *subsumption* alters the result only if a tuple subsuming another tuple is removed from the input of *subsumption* by the *selection*. We distinguish the following cases:

Case 1: Selections involving a column and a constant, e.g., $B < 2$, $C = 3$, $A \neq 4$. In this case, *selection* can be pushed through *subsumption*, because either both the subsuming and the subsumed tuples, or only the

subsuming tuple passes the *selection*. Hence, the result of applying *selection* prior to *subsumption* is the same as the result of applying *selection* after *subsumption*.

Because tuple and subsumed tuple coincide in all values except the ones where the subsumed tuple has a NULL value (see definition), the more interesting cases are the following ones:

Case 2: Selections that test for NULL equality, i.e., B `IS NULL`. In this case, only the subsumed tuple passes the *selection*. Therefore we cannot exchange β and σ in this case (see Rule 6.7).

Case 3: Selections that test for NULL inequality, i.e., B `IS NOT NULL`. This is the opposite of Case 2, so the subsuming tuple passes the *selection*. Therefore we can exchange β and σ in this case (see Rule 6.8).

Case 4: Selections involving two columns, e.g., $B < C$, $C = D$, $A \neq D$. Generally, such a *selection* can be pushed through *subsumption* as in case 1 with only one column.

When combining *selection* and *subsumption*, the following rules hold:

$$\beta\ (\sigma_c\ (\mathrm{A})) \neq \sigma_c\ (\beta\ (\mathrm{A})) \tag{6.7}$$

if c is of the following form: x `IS NULL`, with x being an attribute with NULL values, and

$$\beta\ (\sigma_c\ (\mathrm{A})) = \sigma_c\ (\beta\ (\mathrm{A})) \tag{6.8}$$

in all other cases.

If the operator tree includes a *selection* with conjuncts or disjuncts of conditions, we can push it entirely through *subsumption* if there is no condition of the form x `IS NULL`, x being an attribute. In all other cases we first need to split the conditions using the standard transformation rules for *selection* and then push it down according to the rules above.

Combinations with Join. We show how to exchange *subsumption* and *cartesian product* ($\times$) and refer to the transformations with *selection* for handling *join*, as a *join* can be expressed as a combination of *cartesian product* and *selection*. When exchanging β and $\times$ we must ensure that when applying $\times$, (1) no additional subsuming tuples are introduced if there have been none in the base relations, and (2) all subsuming tuples in the base relations still subsume each other after applying $\times$. The former follows from the definition of tuple subsumption: if two tuples do not subsume each other, every extension of the schema of the tuples as done by *cartesian product* does not change that fact. The latter follows from the fact that by *cartesian product*, two subsuming tuples from one base relation are combined with the same tuples from the other base relation and therefore the two resulting tuples being subsumed in the result of the *cartesian product*.

When combining *cartesian product* and *subsumption*, the following holds:

$$\beta\ (\mathrm{A}\times\mathrm{B}) = \beta\ (\mathrm{A}) \times \beta\ (\mathrm{B}) \tag{6.9}$$

From this, it follows that

$$\begin{aligned} \beta\ (\mathrm{A}\bowtie_c\mathrm{B}) &= \beta\ (\sigma_c\ (\mathrm{A}\times\mathrm{B})) \\ &= \sigma_c\ (\beta\ (\mathrm{A}\times\mathrm{B})) \\ &= \sigma_c\ (\beta\ (\mathrm{A}) \times \beta\ (\mathrm{B})) \\ &= \beta\ (\mathrm{A}) \bowtie_c \beta\ (\mathrm{B}) \end{aligned} \tag{6.10}$$

if c allows for application of Rule 6.8. When exchanging β and $\bowtie_c$ in the query plan, we need to apply the rules involving *selection* from the previous paragraph. Although those rules are not applicable for all possible join conditions, the most often occurring *join* types (e.g., *key join* and *foreign key join*) can be transformed.

Combinations with Grouping and Aggregation. In general, *subsumption* and *grouping* and *aggregation* are not exchangeable as *subsumption* may remove tuples that are essential for the computation of the aggregate. However, there are certain cases in which we can remove the *subsumption* operator and leave only the *grouping*/*aggregation*:

Case 1: If *subsumption* is followed by *aggregation* alone, the *subsumption* operator can be removed if NULL values or duplicate values do not change the result of *aggregation*. This is true for null-tolerant and duplicate-insensitive aggregation functions MIN and MAX (see transformation Rule 6.11).
Case 2: If *subsumption* follows an *aggregation*, the *subsumption* operator can be removed for any aggregation function, as the result of an *aggregation* consists only of one single value (see transformation Rule 6.12).
Case 3: Considering *subsumption* followed by a *grouping*/*aggregation*, or *grouping*/*aggregation* followed by *subsumption*, the *subsumption* can only be removed if the functions that are used are null-tolerant and duplicate-insensitive, i.e., MIN and MAX (see Case 1), and additionally if the grouping column does not contain NULL values (see transformation Rules (6.13) and (6.14)). The latter condition is necessary, because NULL values in the grouping column are combined in a separate group (according to SQL semantics). Tuples belonging to this group may be removed by *subsumption* resulting in a different aggregate for this group. Also, the group itself may be removed by *subsumption* after *grouping*/*aggregation*.

For combinations of *subsumption* with *grouping*/*aggregation*, the following transformation rules hold:

$$\gamma_{f(c)}\left(\beta\left(\mathrm{A}\right)\right) = \gamma_{f(c)}\left(\mathrm{A}\right) \tag{6.11}$$

for any column c and aggregation function $f \in \{\max, \min\}$.

$$\beta\left(\gamma_{f(c)}\left(\mathrm{A}\right)\right) = \gamma_{f(c)}\left(\mathrm{A}\right) \tag{6.12}$$

for any column c and any f.

$$\gamma_{c,f(d)}\left(\beta\left(\mathrm{A}\right)\right) = \gamma_{c,f(d)}\left(\mathrm{A}\right) \tag{6.13}$$

for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$.

$$\beta\left(\gamma_{c,f(d)}\left(\mathrm{A}\right)\right) = \gamma_{c,f(d)}\left(\mathrm{A}\right) \tag{6.14}$$

for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$.

If non-standard aggregation functions are allowed to be used in *grouping*/*aggregation*, then the rules above extend in a way that only null-tolerant, duplicate-insensitive functions are allowed to be be used as function f, such as MIN, MAX, SHORTEST, and LONGEST.

Other Combinations. The *subsumption* operator is not order-preserving. Therefore *sorting* (τ) and *subsumption* cannot be exchanged (see transformation Rule 6.15). When dealing with bags of tuples (allowing exact duplicates), *subsumption* and *distinct* (δ) do not interfere with each other, so they can be exchanged (see transformation Rule 6.16). Additionally, two *subsumption* operators can be combined into one (see transformation Rule 6.17). Although they seem to handle two entirely different cases, *subsumption* and *complementation* are not exchangeable. Their order matters, as a tuple that complements another tuple (and therefore adds some additional information to it) may well be subsumed by another tuple, resulting in that additional information not being added (see transformation Rule 6.18).

For combinations with other operators, the following transformations rules hold:

$$\tau\left(\beta\left(\mathrm{A}\right)\right) \neq \beta\left(\tau\left(\mathrm{A}\right)\right) \tag{6.15}$$

$$\delta\left(\beta\left(\mathrm{A}\right)\right) = \beta\left(\delta\left(\mathrm{A}\right)\right) \tag{6.16}$$

$$\beta\left(\beta\left(\mathrm{A}\right)\right) = \beta\left(\mathrm{A}\right) \tag{6.17}$$

$$\kappa\left(\beta\left(\mathrm{A}\right)\right) \neq \beta\left(\kappa\left(\mathrm{A}\right)\right) \tag{6.18}$$

Combinations with *outer union*	
(6.3)	$\beta(A \uplus B) = \beta(A) \uplus \beta(B)$, if there are no subsumed tuples across source relations, and
(6.4)	$\beta(A \uplus B) = \beta_1(\beta(A) \uplus \beta(B))$, in all other cases.
Combinations with *projection*	
(6.5)	$\beta(A) \neq \beta(\pi(A))$, and
(6.6)	$\pi(\beta(A)) \neq \beta(\pi(A))$
Combinations with *selection*	
(6.7)	$\beta(\sigma_c(A)) \neq \sigma_c(\beta(A))$, if c is of the following form: x `IS NULL`, with x being an attribute with NULL values, and
(6.8)	$\beta(\sigma_c(A)) = \sigma_c(\beta(A))$, in all other cases
Combinations with *cartesian product* and *join*	
(6.9)	$\beta(A \times B) = \beta(A) \times \beta(B)$
(6.10)	$\beta(A \bowtie_c B) = \beta(A) \bowtie_c \beta(B)$, whenever *selection* with condition c can be pushed down
Combinations with *grouping* and *aggregation*	
(6.11)	$\gamma_{f(c)}(\beta(A)) = \gamma_{f(c)}(A)$, for any column c and aggregation function $f \in \{\max, \min\}$
(6.12)	$\beta(\gamma_{f(c)}(A)) = \gamma_{f(c)}(A)$, for any column c and any f
(6.13)	$\gamma_{c,f(d)}(\beta(A)) = \gamma_{c,f(d)}(A)$, for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$
(6.14)	$\beta(\gamma_{c,f(d)}(A)) = \gamma_{c,f(d)}(A)$, for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$
Other combinations	
(6.15)	$\tau(\beta(A)) \neq \beta(\tau(A))$
(6.16)	$\delta(\beta(A)) = \beta(\delta(A))$
(6.17)	$\beta(\beta(A)) = \beta(A)$
(6.18)	$\kappa(\beta(A)) \neq \beta(\kappa(A))$

Table 6.1: Transformation rules for *subsumption* in combination with other relational operators.

6.2.2 Transformation Rules for Complementation and Complement Union

We consider equivalent transformation rules for moving the *complementation* (κ) and the *complement union* ($\boxplus$) operator around in query plans. As is the case with *subsumption* and *minimum union*, in this chapter we also focus on *complementation*, because it is a building block of *complement union* and the latter operator can be moved around in query plans by splitting it into its constituents *complementation* and *outer union*. Table 6.2 at the end of this section on page 93 summarizes the rewriting rules for *complementation*.

Combinations with Outer Union. *Complement union* is defined as the combination of *outer union* with the subsequent combination of complementing tuples (see Section 4). We first examine the exchangeability of *outer union* and *complementation*. As there may exist tuples across sources that complement each other, simply pushing the *complementation* operator through *outer union* is not sufficient. An additional *complementation* operator on top of the *outer union* is still necessary. However, if the schemata of the source relations are distinct, pushing *complementation* through *outer union* is possible and sufficient.

Using a similar technique for *complementation* as we used for *subsumption* (namely introducing a special version of the *subsumption* operator that only considers tuples from different sources) is not possible, because *complementation* and *outer union* are not exchangeable in general. This is due to the fact that tuple complementation is not a transitive relationship. The problem here are complementing tuples across sources, where one of them also complements a tuple in the same source. If the *complementation* is based on a NULL value in a common attribute, then constructing the complement in the source may destroy the complementation relationship across the sources. However, if we can guarantee that there are no NULL values in the common attributes (e.g., if the only common attributes are keys), then *complementation* and *outer union* are exchangeable. Thus, considering *complementation* and *union*, the following rules hold:

$$\kappa(A \uplus B) = \kappa(A) \uplus \kappa(B) \tag{6.19}$$

if there are no complementing tuples across source relations, or if the schemata of the source relations are

distinct, and

$$\kappa\ (\mathrm{A} \uplus \mathrm{B}) = \kappa\ (\kappa\ (\mathrm{A}) \uplus \kappa\ (\mathrm{B})) \tag{6.20}$$

in all other cases.

Combinations with Projection. For *complementation* and *projection*, the same considerations apply as for *subsumption* and *projection*. In general, we cannot push a *projection* down through a *complementation*, i.e., the following rules hold:

$$\kappa\ (\mathrm{A}) \neq \kappa\ (\pi\ (\mathrm{A})) \tag{6.21}$$

$$\pi\ (\kappa\ (\mathrm{A})) \neq \kappa\ (\pi\ (\mathrm{A})) \tag{6.22}$$

Combinations with Selection. Analogously to combinations of *subsumption* and *selection*, in this paragraph we consider rules for combinations of *complementation* and *selection*. As *complementation* deals with the removal of NULL values, in case the *selection* is applied on a column without NULL values (e.g., a key, a unique column, or a `NOT NULL` column), we can indeed push *selection* down through the *complementation* operator (see Rule 6.23). In contrast to *subsumption*, where only the subsuming tuple needs to be preserved, computing the complement requires all complementing tuples to be kept in the input of *complementation*. We consider the same four cases as for *subsumption* and discuss how κ and σ can be interchanged.
Case 1: Selections involving a column and a constant, e.g., $B \leq 2$. In this case *complementation* and *selection* can only be exchanged if we assure that either both or none of the complementing tuples pass the *selection*. This property can be achieved when pushing *selection* through *complementation* by adding an additional condition B `IS NULL` to the selection predicate. This assures that a tuple passes the *selection* if it has a NULL value in that attribute. Hence, the input size of the *complementation* can be reduced by pushing a *selection* through, however, as *complementation* combines its input tuples to new tuples, the original *selection* has to be applied again after *complementation*.
Case 2: Selections that test for NULL equality, i.e., B `IS NULL`. In this case, a tuple complementing another tuple may be filtered, because it does not contribute a value to B. Therefore we cannot exchange κ and σ.
Case 3: Selections that test for NULL inequality, i.e., B `IS NOT NULL`. This is the opposite of Case 2 and this time, only the tuple that does contribute a value to B survives the *selection* if it was pushed below the *complementation*. So in this case we cannot exchange κ and σ either.
Case 4: Selections involving two columns, e.g., $B < C$, $C = D$, $A \neq D$. Such a *selection* can be pushed through *complementation*, if the comparison operator is not one of $=, \neq$, which is due to similar effects as in Case 2 and Case 3. Note that adding an additional condition B `IS NULL` has no effect on these two cases. On the other hand, for range predicates, adding this condition allows us to push down *selection* similarly to Case 1.

Thus, considering *complementation* and *selection*, the following transformation rules hold:

$$\kappa\ (\sigma_c\ (\mathrm{A})) = \sigma_c\ (\kappa\ (\mathrm{A})) \tag{6.23}$$

if all columns involved in c do not contain NULL values.

$$\sigma_c\ (\kappa\ (\sigma_{c \vee cnull}\ (\mathrm{A}))) = \sigma_c\ (\kappa\ (\mathrm{A})) \tag{6.24}$$

if columns involved in c may contain NULL values and c is not of the following form: x `IS NULL`, x `IS NOT NULL`, x being an attribute, and $x = y$, $x \neq y$, x, y being attributes. The condition $c \vee cnull$ is the original condition appended by a test for NULL value (x `IS NULL`) for all the columns x involved in c. Disjunctions and conjunctions of conditions can be handled accordingly as in the case of combinations of *subsumption* and *selection*.

Combinations with Join. We show how to exchange *complementation* and *cartesian product* ($\times$) and refer to the transformations with *selection* for handling *join*s, as a *join* can be expressed as a combination of *cartesian product* and *selection*. Considering *complementation*, the same two properties as for *subsumption* have to be ensured to be able to exchange the operators in the query tree. However, already the first property (no

additional complementing tuples are introduced if there have been none in the source relation) is no longer satisfied: applying *cartesian product* may introduce complementing tuples, even if there are no complementing tuples in the base relation. More precisely, it can be shown that this is the case for base relations that contain subsumed tuples. Fortunately, we can fix this problem by introducing an additional κ operator on top that removes the newly introduced complementing tuples.

The following rules for combining complement and cross product hold:

$$\kappa\ (\mathrm{A}\times\mathrm{B}) = \kappa\ (\mathrm{A}) \times \kappa\ (\mathrm{B}) \tag{6.25}$$

if the base relations A and B do not contain subsumed tuples, and

$$\kappa\ (\mathrm{A}\times\mathrm{B}) = \kappa\ (\kappa\ (\mathrm{A}) \times \kappa\ (\mathrm{B})) \tag{6.26}$$

in all other cases.

Based on above observation and the transformation rules for *selection* and *complementation*, a general rule for combining *join* and *complementation* cannot be devised, so *join* and *complementation* are not exchangeable.

Combinations with Grouping and Aggregation. In general, *complementation* and *grouping/aggregation* are not exchangeable as the *complementation* operator may remove tuples that are essential for the computation of the aggregate. However, as in the case of *subsumption*, there are certain cases in which we can remove the *complementation* operator and leave only the *grouping/aggregation*:

Case 1: If *complementation* is followed by *aggregation* alone, the operator can be removed if NULL values or duplicate values do not change the result of the aggregation. This is true for null-tolerant and duplicate-insensitive functions such as MIN and MAX (see transformation Rule 6.27).

Case 2: If *complementation* follows *aggregation*, the operator can be removed for any aggregation function, as the result of *aggregation* consists only of one single value (see transformation Rule 6.28).

Case 3: Considering *complementation* followed by *grouping/aggregation*, or *grouping/aggregation* followed by *complementation*, the *complementation* operator can only be removed if the functions that are used are null-tolerant and duplicate-insensitive, such as MIN and MAX (see Case 1), and additionally if the grouping column does not contain NULL values (see transformation Rules (6.29) and (6.30)). The latter condition is necessary, because NULL values in the grouping column are combined in a separate group (according to SQL semantics). Tuples belonging to this group may be combined with other tuples, resulting in a different aggregate for this group. Also, the group itself may be changed after *grouping/aggregation*. For combinations of *complementation* with *grouping/aggregation*, the following transformation rules hold:

$$\gamma_{f(c)}\ (\kappa\ (\mathrm{A})) = \gamma_{f(c)}\ (\mathrm{A}) \tag{6.27}$$

for any column c and aggregation function $f \in \{\max, \min\}$.

$$\kappa\ (\gamma_{f(c)}\ (\mathrm{A})) = \gamma_{f(c)}\ (\mathrm{A}) \tag{6.28}$$

for any column c and any f.

$$\gamma_{c,f(d)}\ (\kappa\ (\mathrm{A})) = \gamma_{c,f(d)}\ (\mathrm{A}) \tag{6.29}$$

for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$.

$$\kappa\ (\gamma_{c,f(d)}\ (\mathrm{A})) = \gamma_{c,f(d)}\ (\mathrm{A}) \tag{6.30}$$

for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$.

As in the case of combinations of *subsumption* and *grouping/aggregation*, if we allow for other than only the standard aggregation functions, we need to assure that function f is null-tolerant and duplicate-insensitive in order to be used with above transformation rules (e.g., functions MAX, MIN, SHORTEST, and LONGEST).

Other Combinations. As already mentioned, *subsumption* and *complementation* are not exchangeable (see transformation Rule 6.18). Also, the *complementation* operator is not order-preserving. Therefore it does not make sense to exchange *sorting* (τ) and *complementation* operators (see transformation Rule 6.31). When dealing with bags of tuples (allowing exact duplicates), *complementation* and *distinct* (δ) do not interfere with each other, so they can be exchanged (see transformation Rule 6.32). Lastly, two *complementation* operators can be combined into one (see transformation Rule 6.33). Also, according to the definition of *complementation* a relation needs to have at least three attributes for two tuples being able to complement each other. One attribute needs to have the same value in both tuples and the other two are needed for complementing values and NULL values (see Rule 6.34). For combinations with other operators, the following transformations rules hold:

$$\tau\ (\kappa\ (\mathrm{A})) \neq \kappa\ (\tau\ (\mathrm{A})) \tag{6.31}$$

$$\delta\ (\kappa\ (\mathrm{A})) = \kappa\ (\delta\ (\mathrm{A})) \tag{6.32}$$

$$\kappa\ (\kappa\ (\mathrm{A})) = \kappa\ (\mathrm{A}) \tag{6.33}$$

$$\kappa\ (\mathrm{A}) = \mathrm{A} \tag{6.34}$$

if A has only one or two attributes in the case of Rule 6.34.

Combinations with *outer union*	
(6.19)	$\kappa\ (\mathrm{A} \uplus \mathrm{B}) = \kappa\ (\mathrm{A}) \uplus \kappa\ (\mathrm{B})$, if there are no complementing tuples across sources, and
(6.20)	$\kappa\ (\mathrm{A} \uplus \mathrm{B}) \neq \kappa\ (\mathrm{A}) \uplus \kappa\ (\mathrm{B})$, in all other cases
Combinations with *projection*	
(6.21)	$\kappa\ (\mathrm{A}) \neq \kappa\ (\pi\ (\mathrm{A}))$, and
(6.22)	$\pi\ (\kappa\ (\mathrm{A})) \neq \kappa\ (\pi\ (\mathrm{A}))$
Combinations with *selection*	
(6.23)	$\kappa\ (\sigma_c\ (\mathrm{A})) \neq \sigma_c\ (\kappa\ (\mathrm{A}))$, if c is of the following form: x `IS NULL`, x `IS NOT NULL`, or x `<op>` y with x,y being attributes and `<op>` being one of $\{=, \neq\}$, and
(6.24)	$\sigma_c\ (\kappa\ (\sigma_{c \vee cnull}\ (\mathrm{A}))) = \sigma_c\ (\kappa\ (\mathrm{A}))$, in all other cases, with $cnull$ being the additional test for NULL values for all involved columns
Combinations with *cartesian product* and *join*	
(6.25)	$\kappa\ (\mathrm{A} \times \mathrm{B}) = \kappa\ (\mathrm{A}) \times \kappa\ (\mathrm{B})$, if the base relations A and B do not contain subsumed tuples, and
(6.26)	$\kappa\ (\mathrm{A} \times \mathrm{B}) = \kappa\ (\kappa\ (\mathrm{A}) \times \kappa\ (\mathrm{B}))$, in all other cases
Combinations with *grouping* and *aggregation*	
(6.27)	$\gamma_{f(c)}\ (\kappa\ (\mathrm{A})) = \gamma_{f(c)}\ (\mathrm{A})$, for any column c and aggregation function $f \in \{\max, \min\}$
(6.28)	$\kappa\ (\gamma_{f(c)}\ (\mathrm{A})) = \gamma_{f(c)}\ (\mathrm{A})$, for column c and any f
(6.29)	$\gamma_{c,f(d)}\ (\kappa\ (\mathrm{A})) = \gamma_{c,f(d)}\ (\mathrm{A})$, for columns c, d with c as grouping column not containing NULL values and aggregation function $f \in \{\max, \min\}$
(6.30)	$\kappa\ (\gamma_{c,f(d)}\ (\mathrm{A})) = \gamma_{c,f(d)}\ (\mathrm{A})$, for columns c, d with c as grouping column not containing NULL values and aggregation function $f \in \{\max, \min\}$
Other combinations	
(6.31)	$\tau\ (\kappa\ (\mathrm{A})) \neq \kappa\ (\tau\ (\mathrm{A}))$
(6.32)	$\delta\ (\kappa\ (\mathrm{A})) = \kappa\ (\delta\ (\mathrm{A}))$
(6.33)	$\kappa\ (\kappa\ (\mathrm{A})) = \kappa\ (\mathrm{A})$
(6.34)	$\kappa\ (\mathrm{A}) = \mathrm{A}$, if A has only one or two attributes

Table 6.2: Transformation rules for *complementation* in combination with other relational operators.

6.2.3 Transformation Rules for Data Fusion

In this section we consider equivalent transformation rules for moving around the two versions of the *data fusion* operator in query plans. Recall that the normal version of the *data fusion* operator is denoted ϕ and does not remove subsumed tuples before resolving conflicts whereas ϕ^{β} includes the removal of subsumed tuples. Transformation rules for both *data fusion* operators heavily depend on the conflict resolution functions

that are used and their properties. In general, we observe that as conflict resolution functions become more complex fewer transformations are possible. Because the *data fusion* operator is similar in spirit to standard *grouping/aggregation*, the transformation rules that already exist for these operators (e.g., in combination with *join*, see (Yan and Larson, 1993, 1994, 1995; Tsois and Sellis, 2003b,a)) can be adapted to the *data fusion* operator. Table 6.6 at the end of this section on page 103 summarizes the rewrite rules for *data fusion*, which are described in the next paragraphs. If not noted otherwise, we assume that the identifying attributes in F and the attributes covered by the set of conflict resolution functions CR together cover all attributes from the input relation.

Fusion and Fusion. We first examine the relation of the two different variants of the *data fusion* operator, with and without removal of subsumed tuples. Separating the removal of subsumed tuples from the conflict resolution and doing it beforehand is beneficial, because the normal *data fusion* operator allows for a larger number of transformations, as we see later on.

Concerning NULL values in the identifying attributes, the *data fusion* operator behaves in the same ways as the standard *grouping* and *aggregation* operator, as it groups all NULL tuples together into one group. Thus, separating the removal of subsumed tuples can be done if this NULL group (or NULL groups in case of more than one attribute in F) is not affected by the subsumption. In the worst case, the removal of subsumed tuples may eliminate the NULL group. So, only in case that the identifying attributes that are used in the *data fusion* operator (F) do not contain NULL values, can we safely replace the *subsumption* inside the groups by a *subsumption* before the groups and thus rewrite the data fusion operator ϕ^{β} with a combination of ϕ and β. This separation may ease the application of further transformation rules. So, for *data fusion* alone, the following rule holds:

$$\phi^{\beta}_{F,CR,S}\,(\mathrm{R}) \;=\; \phi_{F,CR,S}\,(\beta\,(\mathrm{R})) \tag{6.35}$$

if the set of identifying attributes F does not contain attributes with NULL values present.

An even stronger transformation rule can be identified: For certain conflict resolution functions, the *data fusion* operator with removal of subsumed tuples and the operator without are equal. This follows as a generalization from transformation Rule 6.14 (*subsumption* and *grouping/aggregation*) to the *data fusion* operator and its conflict resolution functions. The two variants of *data fusion* produce the same result if the result of the conflict resolution functions are not affected by the removal of subsumed tuples in the groups. This is for example true for the MAX function and in general for all functions that are duplicate-insensitive (result is not affected by duplicate values) and where the removal of NULL values also does not affect the result (null-tolerant functions). The following rule holds:

$$\phi^{\beta}_{F,CR,S}\,(\mathrm{R}) \;=\; \phi_{F,CR,S}\,(\mathrm{R}) \tag{6.36}$$

if CR only contains duplicate-insensitive and null-tolerant conflict resolution functions, e.g., MAX, MIN, SHORTEST, and LONGEST.

Next, we consider the case that we need to split a *data fusion* vertically into two distinct *data fusion* operations in the sense that we want to separate the fusion on one column from the fusion on another column. This may be beneficial if we use different conflict resolution functions on columns and are afterwards able to more easily push one of the separated parts down, or push a *selection* down below at least one part of the separated operator. The main idea here is to compute the fused value for each of the columns on the original data and then combine them later on. This idea has also been exploited in the SQL rewriting of the *data fusion* operator in Section 5.4.2 on page 78. So, the following rule holds:

$$\phi_{F,CR\cup\{f_j\},S}\,(\mathrm{R}) \;=\; \phi_{F,CR,S}\,(\mathrm{R}) \;\bowtie_F\; \phi_{F,\{f_j\},S}\,(\mathrm{R}) \tag{6.37}$$

with a *full outer join* on F being the combination operator that combines the application of conflict resolution functions CR and conflict resolution function f_j. Applying this transformation rule repeatedly results in gradually splitting off conflict resolution functions and in an *outer join* combination of several *data fusion* operations. Another possibility to combine the separated *data fusion* operators is to use an *outer union* and

subsequently another *data fusion* with COALESCE as conflict resolution combination operator ($\phi_{F,\{\text{COALESCE}\},S}$). This can further be simplified by a simple *grouping*/*aggregation* using MAX or MIN and grouping on F. So the following rules hold as well:

$$\phi_{F,\{f_1,\ldots,f_j\},S}\,(\mathrm{R}) \;=\; \phi_{F,\bigcup_1^j\{\text{COALESCE}\},S}\,(\phi_{F,\{f_1\},S}\,(\mathrm{R}) \uplus \ldots \uplus \phi_{F,\{f_j\},S}\,(\mathrm{R})) \tag{6.38}$$

$$\phi_{F,\{f_1,\ldots,f_j\},S}\,(\mathrm{R}) \;=\; \gamma_{F,\bigcup_1^j\{\text{MAX/MIN}\}}\,(\phi_{F,\{f_1\},S}\,(\mathrm{R}) \uplus \ldots \uplus \phi_{F,\{f_j\},S}\,(\mathrm{R})) \tag{6.39}$$

Another special case is the combination of two *data fusion* operators, one applied after the other. As the result of *data fusion* is a relation where there exists only one single tuple per attribute value combination of the attributes in F, the following rule holds:

$$\phi_{F_1,CR_1,S_1}\,(\phi_{F_2,CR_2,S_2}\,(\mathrm{R})) = \phi_{F_2,CR_2,S_2}\,(\mathrm{R}) \tag{6.40}$$

if $F_1 = F_2$, i.e., if the identifying attributes are the same, and except for functions in CR_1 that are not idempotent, because they will override the conflict resolution. Thus, if one of the functions in CR_1 is CONSTANT(c) or IGNORE, then the corresponding function in CR_2 (the function on the same attribute) is replaced by the function from CR_1. If CR_1 contains COUNT as function, the corresponding function in CR_2 is replaced by CONSTANT(c) with $c = 1$.

Fusion and Outer Union. In the following we examine the exchangeability of *outer union* and the *data fusion* operation. In the general case, *data fusion* does not distribute over *outer union*. A *data fusion* operator ϕ (or ϕ^β) cannot simply be pushed down below *outer union* regardless of the conflict resolution function used. However, there exists the possibility of *early* fusion, which means that part of the *data fusion* can be done below the *outer union* and the remaining part afterwards. Thus, for some special cases, the following rules hold:

$$\phi_{F,CR,S}\,(\mathrm{R} \uplus \mathrm{S}) = \phi_{F,CR_1,S}\,(\phi_{F,CR_{2R},S}\,(\mathrm{R}) \uplus \phi_{F,CR_{2S},S}\,(\mathrm{S})) \tag{6.41}$$

$$\phi^\beta_{F,CR,S}\,(\mathrm{R} \uplus \mathrm{S}) = \phi^\beta_{F,CR_1,S}\,(\phi^\beta_{F,CR_{2R},S}\,(\mathrm{R}) \uplus \phi^\beta_{F,CR_{2S},S}\,(\mathrm{S})) \tag{6.42}$$

with CR_1, CR_{2R}, and CR_{2S} replacing CR according to the following Table 6.3 on page 96. For example CR = MAX(B) (resolving conflicts on column B by using MAX) can be split into CR_1 = MAX(B) and CR_{2S} = CR_{2S} = MAX(B), given that column B appears in both R and S. If it only appears in one relation, then it is only introduced for that relation. The CR_2 sets that are introduced below the *outer union* contain only the corresponding entries from CR of columns from R, or S respectively. The transformation rule extends to more than two relations. In general this transformation can be applied for the functions in Table 6.3, i.e., for all functions that are *decomposable* (e.g., MAX, MIN) or can otherwise be rewritten by a combination of two functions (e.g., COUNT). As can be seen in the second half of Table 6.3 (Case 2), there are fewer transformations possible for the variant with removal of subsumed tuples (ϕ^β). The removal of subsumed tuples is a limiting factor in this case, as it restricts the functions that may be used in the *fusion* operator to be pushed down.

Fusion and Projection. To decrease the size of the input in terms of colums that need to be processed, a *projection* can be inserted below the *data fusion* ϕ. This is not possible for ϕ^β, as in this case all attributes from the input are needed in order for the subsumption part to give correct results (see 6.2.1). When projecting out attributes before *data fusion*, at least all columns that are mentioned in either the set of identifying attributes F, the set of conflict resolution function CR, or the set S of sort attributes must remain. So, in general *projection* cannot be pushed down through *data fusion*, except for the special case mentioned. The following rule concerning *data fusion* and *projection* holds:

$$\phi_{F,CR,S}\,(\mathrm{R}) = \phi_{F,CR,S}\,(\pi_P\,(\mathrm{R})) \tag{6.43}$$

where P contains all attributes referenced in F, CR, and S.

Case 1: Outer Union and Fusion without Subsumption (ϕ)	
Function	**Rewriting**
CR = MIN	CR_1 = MIN, CR_2 = MIN
CR = MAX	CR_1 = MAX, CR_2 = MAX
CR = SUM	CR_1 = SUM, CR_2 = SUM
CR = COUNT	CR_1 = SUM, CR_2 = COUNT
CR = SHORTEST	CR_1 = SHORTEST, CR_2 = SHORTEST
CR = LONGEST	CR_1 = LONGEST, CR_2 = LONGEST
CR = CONCAT	CR_1 = CONCAT, CR_2 = CONCAT
all other functions	no replacement possible
Case 2: Outer Union and Fusion with Subsumption (ϕ^β)	
Function	**Rewriting**
CR = MIN	CR_1 = MIN, CR_2 = MIN
CR = MAX	CR_1 = MAX, CR_2 = MAX
CR = SHORTEST	CR_1 = SHORTEST, CR_2 = SHORTEST
CR = LONGEST	CR_1 = LONGEST, CR_2 = LONGEST
all other functions	no replacement possible

Table 6.3: Early fusion: splitting a *data fusion* operation when distributing it over *outer union*.

Fusion and Selection. In this paragraph we examine the interchangeability of ϕ and σ with the main goal of pushing *selections* down below *data fusion* in order to decrease the input cardinality of the *data fusion* operator. Where possible we also push the *selection* partially below the *data fusion* operator by introducing a new *selection* below and keeping the original *selection* above.

We consider a *data fusion* operator $\phi_{F,CR,S}$ followed by a *selection* σ_c. If the selection condition c only includes columns that are also used as identifying columns in the *data fusion* operator below, then *selection* can be easily pushed down (see Rule 6.44 further down at the end of the section on *data fusion* and *selection*). This can be done because only entire object representations are filtered out. The other case, where c also involves columns that contain data conflicts and are therefore handled by a conflict resolution function is more complex and depends on the conflict resolution function that is used. We distinguish the following cases:

Case 1: Selections involving a column and a constant, e.g., $B = 3$, $C < 2$, $D > z$. To be able to push the *selection* down below the *data fusion* operator and not change the result we need to ensure that no tuple that is needed for a correct conflict resolution is filtered out before the *data fusion* operator. This not only depends on the selection condition but also on the conflict resolution function used, as we see further on.

Consider as an easy example the query $\sigma_{B=3}\ (\phi_{ID,max(B),ID}\ (R))$ on relation R consisting of a schema $S = (R, \{ID, B\}, dom), dom(ID) = dom(B) = integer$, and the tuples $T = \{(1,3), (1,4), (2,2), (2,3)\}$. The query selects all tuples where conflict resolution on R in attribute B using function MAX results exactly in the value 3. The result of the above query would be the tuple set $\{(2,3)\}$. Simply pushing the selection down is not an equivalent transformation, as query $\phi_{ID,max(B),ID}\ (\sigma_{B=3}\ (R))$ would result in the two tuples $\{(1,3), (2,3)\}$. We see that we need to ensure in this case that all tuples with a B value of *at least* 3 need to be retained before conflict resolution using MAX takes place. If we use MIN as conflict resolution instead of MAX we need to ensure that all tuples with a B value of *at most* 3 are retained. Still, in addition we need to keep the original *selection* on top to filter out the cases where B is exactly 3 and thus are only able to partially push the *selection* down. This partial pushdown works because MAX and MIN as conflict resolution functions choose an existing value and do not use all existing conflicting values to compute the resolved values. Please note that this rewriting for MAX and MIN works for both versions of the *data fusion* operator, with and without removal of subsumed tuples (ϕ and ϕ^β). Replacing MIN/MAX with SUM renders the pushdown impossible, as well as the use of other functions such as AVERAGE, MEDIAN, VOTE, etc.

However, besides MAX and MIN, there are several other functions, such as SHORTEST, LONGEST and the taxonomic functions MOST GENERAL CONCEPT and MOST SPECIFIC CONCEPT, that can also be pushed down. This is possible because they are duplicate-insensitive, because they rely on an ordered domain and as long as a suitable expression for the *selection* that has been pushed down can be found. In case of LONGEST this is for

example the expression $length(B) \geq length(3)$. Table 6.4 on page 97 shows the functions and the necessary *selection* introduced below.

Case 1a: $B = x$**, B being an attribute, x a constant of** $dom(B)$	
Function	**Rewriting**
MIN	insert $\sigma_{B \leq x}$ below the data fusion operator
MAX	insert $\sigma_{B \geq x}$ below the data fusion operator
SHORTEST	insert $\sigma_{length(B) \leq length(x)}$ below the data fusion operator
LONGEST	insert $\sigma_{length(B) \geq length(x)}$ below the data fusion operator
MOST GENERAL CONCEPT	no transformation possible with definition as lowest common ancestor
MOST SPECIFIC CONCEPT	insert $\sigma_{taxless(B,x,TAXO)}$ below the data fusion operator with $taxless(B, x, TAXO)$ evaluating to true, if the actual B value is most as specific as x according to taxonomy $TAXO$
CHOOSE(*source*), PREFER(*source*)	insert $\sigma_{fromSource(B,s)}$ below the data fusion operator with $fromSource(B, s)$ evaluating to true, if the actual B value is from source s. Such a selection can be very easy, if source identifiers come with every tuple in a designated attribute
FIRST, LAST,COALESCE	no transformation possible, unless there is a defined order on tuples
RANDOM	no transformation possible as function behaves indeterministic
IGNORE	no transformation necessary, result is alway NULL, selection never selects anything
MOST RECENT	no transformation possible, unless there is a defined and easy accessible recency order on tuples
CHOOSE DEPENDING(*column,value*)	possible, insert $\sigma_{column=value}$ below
all other functions	no transformation possible
Case 1b: $B < x$**, B being an attribute, x a constant of** $dom(B)$	
Function	**Rewriting**
MIN	insert $\sigma_{B<x}$ below the data fusion operator.
MAX	no transformation possible
all other functions	no transformation possible
Case 1c: $B > x$**, B being an attribute, x a constant of** $dom(B)$	
Function	**Rewriting**
MIN	no transformation possible
MAX	insert $\sigma_{B>x}$ below the data fusion operator.
all other functions	no transformation possible

Table 6.4: Transformation rules for combinations of *data fusion* and *selections* for different conflict resolution functions involving a single column compared to a constant, i.e., $B = x$, $C < y$, $D > z$.

A *selection* that is introduced below the *data fusion* operation due to a column B also influences conflict resolution in other columns, as whole tuples are filtered out. Therefore, the above mentioned transformation rules are only applicable if there is only one conflicting column or all *selections* that have been introduced below need to be disjunctively combined. If necessary and beneficial, the rules from the beginning of Section 6.2.3 may be applied in beforehand.

If the *selection* does not contain an equality comparison but a comparison by < or >, the basic principle of introducing an additional *selection* below to early filter out tuples and save runtime stays the same. However, the introduced *selection* depends on the conflict resolution function that is used and is different to the cases with equality comparison. For example, we are able to introduce an additional $\sigma_{B>3}$ if the MAX function is used. On the other hand, introducing a *selection* below is not possible/beneficial for the MIN function. See Table 6.4 on page 97 for an overview.

Case 2: Selections that test for NULL equality, e.g., B `IS NULL`. There exists no transformation rule for this case. For many functions (such as MAX, MIN, etc.) – because they only consider NON-NULL values – the only way that they result in a NULL value is the case where all input values are NULL values. Introducing a *selection* $\sigma_{B \texttt{ IS NULL}}$ or $\sigma_{B \texttt{ IS NOT NULL}}$ would not make a difference. For other functions (such as FIRST, COALESCE, etc.),

Case 2: B `IS NULL`, B **being an attribute**	
Function	**Rewriting**
all functions	no transformation possible

Case 3: B `IS NOT NULL`, B **being an attribute**	
Function	**Rewriting**
MIN, MAX, SHORTEST, LONGEST, VOTE, SUM, COUNT, COALESCE, AVERAGE, MEDIAN, VARIANCE, STANDARD DEVIATION, TOKEN UNION, TOKEN INTERSECTION, COMMON BEGINNING, COMMON ENDING, MOST GENERAL CONCEPT, MOST SPECIFIC CONCEPT, MOST COMPLETE	remove original selection and insert $\sigma_{B \texttt{ IS NOT NULL}}$ below the data fusion operator
all other functions	no transformation possible

Case 4: $B < C$,$B = C$, **etc.,** B, C **being attributes**	
Function	**Rewriting**
all functions	no transformation possible

Table 6.5: Rewriting Selections for different conflict resolution functions involving checks for NULL values and more than one column, i.e., B `IS NULL`, $C < D$, etc.

the order of the tuples is important and a simple *selection* introduced below would also not make a difference. Overall, we cannot devise a transformation rule for these cases.

Case 3: Selections that test for NULL inequality, e.g., B `IS NOT NULL`. If the conflict resolution function used produces a NON-NULL output, if there is at least one NON-NULL input, then such a *selection* can be pushed through a *data fusion* operation, i.e., a *selection* $\sigma_{B \texttt{ IS NOT NULL}}$ is introduced below and the original *selection* is removed. Examples for such conflict resolution functions are MAX, LONGEST, and VOTE among others. Table 6.5 on page 98 gives an overview. If the conflict resolution function may result in a NULL value (e.g., FIRST, RANDOM, or CHOOSE(*source*) among others), such a *selection* can not be pushed through, there does not exist a transformation rule in this case.

Case 4: Selections involving two columns, e.g., $B < C$, $C = D$. The last case is the case where two columns are compared by equality or a less than or greater than comparison. There is no transformation rule for these cases as the dependencies among attribute values and conflict resolution functions are too complex. In general all value combinations in the input are needed to decide about the selection predicate that compares the result of the conflict resolution functions on the input.

In summary, considering queries with combinations of a *selection* and a *data fusion* operator the following rules hold:

$$\sigma_c\left(\phi_{F,CR,S}\left(\mathrm{A}\right)\right) = \sigma_c\left(\phi_{F,CR,S}\left(\sigma_{c'}\left(\mathrm{A}\right)\right)\right) \tag{6.44}$$

where an additional *selection* $\sigma_{c'}$ is introduced according to the rules stated in Tables 6.4 (page 97) and 6.5 (page 98). Concerning *data fusion* with removal of subsumed tuples, we can push a *selection* below ϕ^{β} by applying a combination of the rules from the beginning of this section that transforms a ϕ^{β} into a ϕ, the transformation rules pushing σ below ϕ and furthermore the transformation rules concerning combinations of *selection* and *subsumption*.

The second rule that holds for combinations of selection and data fusion is the special case, in which we can abandon the additionally introduced selection. Considering selection and data fusion the following rule also holds:

$$\sigma_c\left(\phi_{F,CR,S}\left(\mathrm{A}\right)\right) = \phi_{F,CR,S}\left(\sigma_c\left(\mathrm{A}\right)\right) \tag{6.45}$$

if c involves only columns that are identifying columns, i.e., are in F but not involved in any conflict resolution from CR. In all remaining cases *selection* cannot be pushed below data fusion, not even partially. Conjunctions and disjunctions of *selections* are first separated using the standard rules for conjunctions and disjunctions of *selections* and then considered separately.

Combinations with Join. In the following we consider combinations of *join* and *data fusion* and give transformation rules for combinations of both operators. We are particularly interested in the cases where two or more sources are first combined by a *join* and afterwards a *data fusion* operator is applied to the result. We will look at the conditions under which the *data fusion* operator can be pushed down below the *join* to do *early* fusion, either fully or only partially. Early fusion reduces the input cardinality of the *join* and thus saves runtime. There may be just as well cases where the opposite transformation makes sense, namely pulling a *data fusion* operation up above a *join*. These types of transformation rules are an application to the *data fusion* operator of work presented in (Yan and Larson, 1995; Tsois and Sellis, 2003b) where *join* and *grouping/aggregation* are considered.

In particular we want to consider situations similar to the following example scenario: A data fusion operation is executed on CD data, where information on Audio CD's is stored in two tables that are joined. A first table (CD) holds an object identifier (attribute CD_ID in the example) and the CD's title, and a second table (LABEL) holds information on audio labels. The 1:1 relationship (using an *equality join*) between CD and label is modeled by a join attribute in the CD table linking to a label ID in the table LABEL. Please note that we keep the number of attributes to a minimum to keep the example simple, but more attributes (such as price for CD's or address for labels) are possible. The *data fusion* operation uses the object identifier to fuse data and resolves conflicts in all other attributes. The join attributes are only used for the *join* and therefore not included in the output. Also note that we consider only 1:1 relationships on the object level, meaning that one CD object is related to at most one label object. However, as the CD table is dirty, there exists a *n*:1 relationship on the tuple level, where *n* representations (tuples) of CD objects are connected to 1 representation of label objects (*n*:*m* in case of a dirty LABEL table as well).

We assume a dirty CD table (different representations of the same CD) and subsequently focus on three representative case: In a first case, the table LABEL is clean (one representation per label) and the *join* also correlates only one CD object to one label, thus preserving the 1:1 relationship on objects (although not on tuples). We will call this a *clean join*, as the *join* does not mess things up. The second case removes the condition of the *clean join*. Here, it is possible that different object representations of the same CD, are connected to different labels. The last case also removes the condition that the table LABEL is clean, thus also allowing for multiple representations of one label. We also assume that all tables have NON-NULL identifiers and join attributes, i.e., CD_ID and L_ID are NON-NULL. In the following we may also refer to table CD as the main table and table LABEL as the associated table.

Case 1: Table CD is dirty, Table LABEL is clean, and both tables are combined by a clean join. To illustrate, consider the following example scenario:

CD (A)

CD_ID	Title	L_ID
A123	The Seasons	1
A123	Haydn: Seasons	1

LABEL (B)

L_ID	Name
1	Harmonia Mundi
2	Deutsche Grammophon

The CD table contains two object representations for a CD with $CD\_ID = A123$, each with a different title. Table LABEL on the other side is clean, which means that there is only one representation per label. The *join* between the tables is also *clean*, i.e., all representations of $CD\_ID = A123$ are connected to the same label. A related situation involving *grouping/aggregation* and *join* is described in (Yan and Larson, 1994) as *group by pushdown*, or *group by pull up*. The situation above is characterized by two conditions: First, table LABEL is clean, meaning that the L_ID attribute is *unique* and NON-NULL in the table. Second, the *join* is also *clean*, meaning that each CD object id joins with the same label id in the other table, i.e., there exists a functional dependency $FD : CD\_ID \rightarrow L\_ID$. In this situation, the following transformation holds:

$$\phi_{F,CR_A \cup CR_B,S}\ (\mathrm{A} \bowtie_{A.id=B.id} \mathrm{B}) = \pi_P\ (\phi_{F \cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id} \mathrm{B}) \tag{6.46}$$

where P contains all attributes from F and all attributes used in CR_A and CR_B and if there is (1) a functional dependency $FD : F \rightarrow A.id$ between the identifying attributes in table A and the join attribute in A, (2) the label identifying (and join) attribute id is unique in B, and (3) all functions in CR_B are duplicate-insensitive, null-tolerant, and idempotent (e.g., MAX, LONGEST). The transformation Rule 6.46 is depicted in Figure 6.2.

We argue that the two expressions

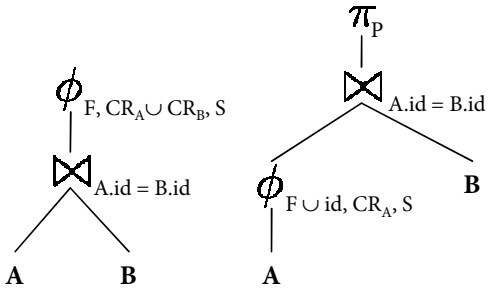


Figure 6.2: Query trees illustrating transformation Rule 6.46, that states the conditions under which both trees are equivalent.

(a) $\phi_{F,CR_A \cup CR_B,S}$ (A⋈$_{A.id=B.id}$B) (lefthand side of equation 6.46) and
(b) π_P ($\phi_{F \cup id,CR_A,S}$ (A) ⋈$_{A.id=B.id}$B) (righthand side of equation 6.46)

have the same result. First, as can easily be seen, they have the same result schema. Producing the same final schema is the sole reason for the *projection* in (b). Second, they both are also producing the same result tuples for the following reasons: Condition (2) ensures that each tuple from A is paired with at most one tuple from B by the *join* on id in (a). In case of (b), a tuple of the result of the *data fusion* operation on A alone is paired with at most one tuple from B. Tuples that do not have a join partner are discarded either way. Condition (2) and (1) together ensure that for each *object* the functions in CR_B are fed with only one single value (single, but different for different objects). In the worst case, the value in (a) is duplicated n times, if there are n different object representations for this object. Then, condition (3) ensures that resolving the conflict resolution via CR_B in (a) has the same result as the value that is joined to this same object in (b). Moreover, this is true as condition (1) ensures that a *data fusion* on identifying attributes F is the same as a *data fusion* on identifying attributes $\{F, id\}$ leaving only one object representation after the early fuse. However, we still need a *data fusion* on $\{F, id\}$ because we need to retain attributes id for joining later on.

Overall, an early fusion does not remove object representations nor does it add additional objects. Furthermore, all objects are joined with the same attribute values from the associated table and the result of conflict resolution is the same. Finally, the specific conditions of this first case allow to fully push ϕ down and do an early fusion.

Concerning the two different variants of the *data fusion* operator we argue that the transformation is also valid for ϕ^{β}. The fact that F is NON-NULL makes it possible to first apply transformation Rule 6.35 and then push the *subsumption* operator down through the join according to transformation Rule 6.10. *Subsumption* can be entirely pushed down to the main table as there are no subsumed tuples present in the associated table and between the two tables. Then, we can apply transformation Rule 6.46 and finally merge the *data fusion* operator with the *subsumption* on the main table (again applying Rule 6.35). So the following transformation rule holds as well:

$$\phi^{\beta}_{F,CR_A \cup CR_B,S} \text{ (A}\bowtie_{A.id=B.id}\text{B)} = \pi_P \text{ (}\phi^{\beta}_{F \cup id,CR_A,S} \text{ (A) } \bowtie_{A.id=B.id}\text{B)} \tag{6.47}$$

where the same conditions as for transformation Rule 6.46 apply.

Case 2: Table CD is dirty, Table LABEL is clean, and both tables are combined by a dirty join. To illustrate, consider the following example scenario:

CD (A)

CD_ID	Title	L_ID
A123	The Seasons	1
A123	Haydn: Seasons	2

LABEL (B)

L_ID	Name
1	Harmonia Mundi
2	Deutsche Grammophon

In contrast to case 1, we consider a *dirty join*, removing the condition that each single CD object joins with the same label object. More formally we omit condition (1) from above, the functional dependency between the identifying and the join attribute in the main table. The other two conditions (2) and (3) stay the same. A

similar situation involving *grouping/aggregation* and *join* is described in (Yan and Larson, 1995) as *eager group by*, or *lazy group by*. In this situation, the following transformation holds:

$$\phi_{F,CR_A \cup CR_B,S}\ (\mathrm{A} \bowtie_{A.id=B.id} \mathrm{B}) = \phi_{F,CR_A \cup CR_B,S}\ (\phi_{F \cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id} \mathrm{B}) \tag{6.48}$$

if (1) the join attribute *id* is unique in B, and (2) all functions in CR_B are duplicate insensitive, null-tolerant, and idempotent (e.g., MAX, LONGEST). The transformation Rule 6.48 is depicted in Figure 6.3.

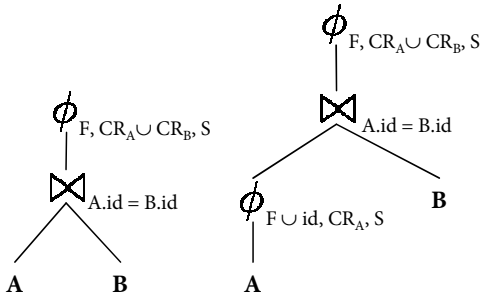


Figure 6.3: Query trees illustrating transformation Rule 6.48, that states the conditions under which both trees are equivalent.

We argue as for case 1 that the two expressions
(a) $\phi_{F,CR_A \cup CR_B,S}\ (\mathrm{A} \bowtie_{A.id=B.id} \mathrm{B})$ (lefthand side of equation 6.48) and
(b) $\phi_{F,CR_A \cup CR_B,S}\ (\phi_{F \cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id} \mathrm{B})$ (righthand side of equation 6.48)
have the same result. The argument follows the one for case 1 with the significant difference that we cannot rely on the fact that *data fusion* on F is the same as *data fusion* on $\{F, id\}$ and that after the early *data fusion* operation on (b) there is at most one representation per object. This is not true, as the functional dependency of condition (1) does not hold. However, the final *data fusion* operator in (b) (on top) will fuse eventually existing multiple representations into at most one representation per object as given by identifying attributes F. With condition (2) and (3) follows that the result of the conflict resolution functions is still the same for each object in (a) and (b).

This transformation is also valid for the *data fusion* variant with removal of subsumed tuples. The removal of subsumed tuples as part of the *data fusion* operations present in the transformation rule does not affect the resolved values for attributes from both relations and from the intermediate relations because of condition (3) on the possible conflict resolution functions. So the following transformation rule holds as well:

$$\phi^{\beta}_{F,CR_A \cup CR_B,S}\ (\mathrm{A} \bowtie_{A.id=B.id} \mathrm{B}) = \phi^{\beta}_{F,CR_A \cup CR_B,S}\ (\phi^{\beta}_{F \cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id} \mathrm{B}) \tag{6.49}$$

where the same conditions as for transformation Rule 6.48 apply.

Case 3: Table CD is dirty, Table LABEL is dirty, and both tables are combined by a dirty join. To illustrate, consider the following example scenario:

CD (A)

CD_ID	Title	L_ID
A123	The Seasons	1
A123	Haydn: Seasons	2

LABEL (B)

L_ID	Name
1	Harmonia Mundi
2	Deutsche Grammophon
2	Dt. Grammophon

In contrast to cases 1 and 2, we consider a dirty table LABEL as well, additionally removing the condition that there is only one representation per label in table LABEL. More formally we omit condition (2) from above, the condition that the identifying and join attribute in the associated table is unique. The remaining condition (3) stays the same. A similar situation involving *grouping/aggregation* and *join* is described in (Yan and Larson, 1995) as *eager split*, or *lazy split*. In this situation, the following transformation holds:

$$\phi_{F,CR_A \cup CR_B,S}\ (\mathrm{A} \bowtie_{A.id=B.id} \mathrm{B}) = \phi_{F,CR_A \cup CR_B,S}\ (\phi_{F \cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id} \phi_{id,CR_B,S}\ (\mathrm{B})) \tag{6.50}$$

if (1) all functions in CR_B are duplicate-insensitive, null-tolerant, and idempotent (e.g., MAX, LONGEST). Transformation Rule 6.50 is depicted in Figure 6.4.

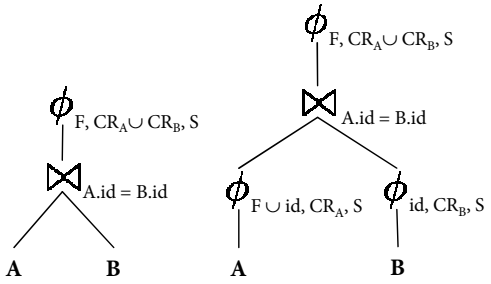


Figure 6.4: Query trees illustrating transformation Rule 6.50, that states the conditions under which both trees are equivalent.

We argue as in the last two cases that the two expressions
(a) $\phi_{F,CR_A \cup CR_B,S}$ (A$\bowtie_{A.id=B.id}$B) (lefthand side of equation 6.50) and
(b) $\phi_{F,CR_A \cup CR_B,S}$ ($\phi_{F \cup id,CR_A,S}$ (A) $\bowtie_{A.id=B.id} \phi_{id,CR_B,S}$ (B)) (righthand side of equation 6.50)
result in the same relation. The argument follows the one for cases 1 and 2 with the difference that we additionally need to take care of different conflicting values for different label objects, in the form of the second early *data fusion* on the LABEL table. In (a) these conflicting values are joined to its associated object representations and fed into the final *data fusion* on top. In (b) conflicts for these value are resolved early. Due to the condition on the conflict resolution functions being duplicate-insensitive, null-tolerant, and idempotent, the final resolved value for each object is the same in both cases (a) and (b). The cardinality of one representation per object is finally secured by the final *data fusion* operator on top.

This transformation is also valid for the *data fusion* variant with removal of subsumed tuples. The removal of subsumed tuples as part of the *data fusion* operations present in the transformation rule does not affect the resolved values for attributes from both relations and from the intermediate relations because of the condition (3) on the possible conflict resolution functions. So the following transformation rule holds as well:

$$\phi^{\beta}_{F,CR_A \cup CR_B,S} (\text{A} \bowtie_{A.id=B.id} \text{B}) = \phi^{\beta}_{F,CR_A \cup CR_B,S} (\phi^{\beta}_{F \cup id,CR_A,S} (\text{A}) \bowtie_{A.id=B.id} \phi^{\beta}_{id,CR_B,S} (\text{B})) \tag{6.51}$$

where the same conditions as for transformation Rule 6.50 apply.

Summarizing, we identified three cases and their conditions in which we can safely exchange a *join* and a *data fusion* operator. We stated the associated transformation rules for three characteristic cases of combinations of the *join* and the *data fusion* operator. By considering only a 1:1 relationship between objects from a main and an associated table we assume that we are fusing representations of objects and are considering conflicts only on the data level. We are specifically not considering representational conflicts that arise when considering 1:n relationships between objects from a main and an associated table, as would be the case of CD representations and their associated tracks.

The main difference to the work in (Yan and Larson, 1995; Tsois and Sellis, 2003b) is – besides the different operators and functions considered – the class of queries, based on a different scenario: we consider a data integration scenario, where there may be different representations of objects, and conflicting attributes generally may be scattered among all input tables but the object identifier usually resides in the main table, whereas (Yan and Larson, 1995; Tsois and Sellis, 2003b) consider an OLAP environment where grouping attributes may be present in all tables (dimensions) and aggregation attributes are usually just from one of the input tables (facts).

6.3 Evaluating Data Fusion Queries

The transformation rules stated in the first part of this chapter can be used by a database optimizer to create a multitude of different – nevertheless equivalent – query plans. Usually, during a second step of optimization,

Combinations with *data fusion*	
(6.35)	$\phi^{\beta}_{F,CR,S}\ (\mathrm{R}) = \phi_{F,CR,S}\ (\beta\ (\mathrm{R}))$, if the set of identifying attributes F does not contain attributes with NULL values present.
(6.36)	$\phi^{\beta}_{F,CR,S}\ (\mathrm{R}) = \phi_{F,CR,S}\ (\mathrm{R})$, if CR only contains duplicate-insensitive and null-tolerant conflict resolution functions, e.g., MAX, MIN, SHORTEST, and LONGEST.
(6.37)	$\phi_{F,CR\cup\{f_j\},S}\ (\mathrm{R}) = \phi_{F,CR,S}\ (\mathrm{R}) \bowtie\!\!\!\!\bowtie_F \phi_{F,\{f_j\},S}\ (\mathrm{R})$, with a *full outer join* on F being the combination operator that combines the application of conflict resolution functions CR and conflict resolution function f_j.
(6.38)	$\phi_{F,\{f_1,\ldots,f_j\},S}\ (\mathrm{R}) = \phi_{F,\bigcup_1^j\{\text{COALESCE}\},S}\ (\phi_{F,\{f_1\},S}\ (\mathrm{R}) \uplus \ldots \uplus \phi_{F,\{f_j\},S}\ (\mathrm{R}))$
(6.39)	$\phi_{F,\{f_1,\ldots,f_j\},S}\ (\mathrm{R}) = \gamma_{F,\bigcup_1^j\{\text{MAX}\}}\ (\phi_{F,\{f_1\},S}\ (\mathrm{R}) \uplus \ldots \uplus \phi_{F,\{f_j\},S}\ (\mathrm{R}))$
(6.40)	$\phi_{F_1,CR_1,S_1}\ (\phi_{F_2,CR_2,S_2}\ (\mathrm{R})) = \phi_{F_2,CR_2,S_2}\ (\mathrm{R})$, if $F_1 = F_2$. If one of the functions in CR_1 is CONSTANT(c) or IGNORE, then the corresponding function in CR_2 (the function on the same attribute) is replaced by the function from CR_1. If CR_1 contains COUNT as function, the corresponding function in CR_2 is replaced by CONSTANT(c) with $c = 1$.
Combinations with *outer union*	
(6.41)	$\phi_{F,CR,S}\ (\mathrm{R}\uplus\mathrm{S}) = \phi_{F,CR_1,S}\ (\phi_{F,CR_{2R},S}\ (\mathrm{R}) \uplus \phi_{F,CR_{2S},S}\ (\mathrm{S}))$, with CR_1, CR_{2R}, and CR_{2S} replacing CR according to Table 6.3.
(6.42)	$\phi^{\beta}_{F,CR,S}\ (\mathrm{R}\uplus\mathrm{S}) = \phi^{\beta}_{F,CR_1,S}\ (\phi^{\beta}_{F,CR_{2R},S}\ (\mathrm{R}) \uplus \phi^{\beta}_{F,CR_{2S},S}\ (\mathrm{S}))$, with CR_1, CR_{2R}, and CR_{2S} replacing CR according to Table 6.3.
Combinations with *projection*	
(6.43)	$\phi_{F,CR,S}\ (\mathrm{R}) = \phi_{F,CR,S}\ (\pi_C\ (\mathrm{R}))$, where C contains all attributes referenced in F, CR, and S.
Combinations with *selection*	
(6.44)	$\sigma_c\ (\phi_{F,CR,S}\ (\mathrm{A})) = \sigma_c\ (\phi_{F,CR,S}\ (\sigma_{c'}\ (\mathrm{A})))$, where an additional selection $\sigma_{c'}$ is introduced according to the rules stated in Table 6.4.
(6.45)	$\sigma_c\ (\phi_{F,CR,S}\ (\mathrm{A})) = \phi_{F,CR,S}\ (\sigma_c\ (\mathrm{A}))$, if c involves only columns that are identifying columns, i.e., are in F but not involved in any conflict resolution from CR.
Combinations with *join*	
(6.46)	$\phi_{F,CR_A\cup CR_B,S}\ (\mathrm{A}\bowtie_{A.id=B.id}\mathrm{B}) = \pi_P\ (\phi_{F\cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id}\mathrm{B})$,where P contains all attributes from F and all attributes used in CR_A and CR_B and if there is (1) a functional dependency $FD : F \rightarrow A.id$ between the identifying attributes in table A and the join attribute in A, (2) the join attribute id is unique in B, and (3) all functions in CR_B are duplicate insensitive and idempotent (e.g., MAX, LONGEST).
(6.47)	$\phi^{\beta}_{F,CR_A\cup CR_B,S}\ (\mathrm{A}\bowtie_{A.id=B.id}\mathrm{B}) = \pi_P\ (\phi^{\beta}_{F\cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id}\mathrm{B})$, with conditions as in (6.46)
(6.48)	$\phi_{F,CR_A\cup CR_B,S}\ (\mathrm{A}\bowtie_{A.id=B.id}\mathrm{B}) = \phi_{F,CR_A\cup CR_B,S}\ (\phi_{F\cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id}\mathrm{B})$, if (1) the join attribute id is unique in B, and (2) all functions in CR_B are duplicate insensitive, null-tolerant, and idempotent (e.g., MAX, LONGEST).
(6.49)	$\phi^{\beta}_{F,CR_A\cup CR_B,S}\ (\mathrm{A}\bowtie_{A.id=B.id}\mathrm{B}) = \phi^{\beta}_{F,CR_A\cup CR_B,S}\ (\phi^{\beta}_{F\cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id}\mathrm{B})$, with conditions as in (6.49)
(6.50)	$\phi_{F,CR_A\cup CR_B,S}\ (\mathrm{A}\bowtie_{A.id=B.id}\mathrm{B}) = \phi_{F,CR_A\cup CR_B,S}\ (\phi_{F\cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id}\phi_{id,CR_B,S}\ (\mathrm{B}))$, if (1) all functions in CR_B are duplicate insensitive and idempotent (e.g., MAX, LONGEST).
(6.51)	$\phi^{\beta}_{F,CR_A\cup CR_B,S}\ (\mathrm{A}\bowtie_{A.id=B.id}\mathrm{B}) = \phi^{\beta}_{F,CR_A\cup CR_B,S}\ (\phi^{\beta}_{F\cup id,CR_A,S}\ (\mathrm{A}) \bowtie_{A.id=B.id}\phi_{id,CR_B,S}\ (\mathrm{B}))$, with conditions as in (6.51)

Table 6.6: Rewrite rules for *data fusion* in combination with other relational operators.

in these different alternative plans each operator (or its particular implementation) is assigned a cost and the plans are evaluated using the combined cost of all operators in the query. To achieve this, a cost model is used to estimate runtimes and a selectivity estimate is used to estimate the output size of an operator as percentage of the input size. The query plan that has the least total costs and thus promises to run fastest gets executed. So, in order to estimate the runtime of a particular query plan basically two things are needed: a) a cost model and formulas to determine the cost of an operator implementation and b) a way of estimating the selectivity of an operator to estimate the output size. We will briefly mention, and overview some basic cost formulas and selectivity estimations for the operators that have been defined earlier.

6.3.1 Estimating Costs

In the literature different cost models are present, among the most popular cost models based on disk i/o and cpu runtime (e.g., as in (Ullman et al., 2001)). When presenting the algorithms earlier on we did not particularly focus on i/o optimized versions (most of the implementations require all tuples of a relation to be held in main memory, resulting in reading and writing the involved relations just once), but merely tried to implement the different data fusion operations in an runtime efficient manner. Therefore, in this section we focus on a cpu runtime based cost model. We give a brief sketch of cost formulas for some of the implementations for computing *subsumption*, *complementation* and *data fusion*. The cost formulas are based on the runtime analysis in Chapter 5 and we count the number of distinct tuple operations (e.g., tuple comparisons, or testing two tuples for *subsumption* or *complementation*). A detailed cost model would require quite specific information about the distribution of NULL and other values among the attributes to determine partition sizes. As such information is difficult to acquire or costly to store and keep up-to-date we restrict the cost model to only include the number of partitions and the size of the NULL partition as both can easily be deduced from the relations at practically no cost.

Subsumption We give cost formulas for the two best implementations for *subsumption* (see the section on experiments, Section 7, for more details) and the naive implementation:
Simple Subsumption: With n being the number of tuples in the relation, the cost formula for the *Simple Subsumption* algorithm is given as

$$C_{SMPS} = \frac{1}{2}n^2 \tag{6.52}$$

Null-Pattern-Based Subsumption: With n being the number of tuples in the relation, the cost formula for the *Null-Pattern-Based Subsumption* algorithm is given as

$$C_{NPBS} = n \log n \tag{6.53}$$

Partitioning(SMPS): We cover the Partitioning variant in more detail. Assuming $k = |P_\perp|$ as the size (in tuples) of the NULL partition, n as the size of the relation and p as the number of partitions P_i, then – assuming an equal distribution of tuples among partitions – the average size of a normal partition P_i is $\frac{n-k}{p}$. Furthermore to keep formulas simple we assume an intermediate selectivity of 100% (no tuple subsumes the other) and assume applying the *Simple Subsumption* algorithm on the partitions. We also assume that creating the partitioning can be done at practically no cost while reading the relation into memory. This leads to the general cost formula for the *Partitioning(SMPS)* algorithm:

$$\begin{aligned} C_{Partitioning(SMPS)} &= \frac{1}{2}k^2 + \frac{1}{2}\left(\frac{n-k}{p}\right)^2 p + \left(\frac{(n-k)}{p}k\right)p \\ &= \frac{1}{2}k^2 + \frac{1}{2}\frac{(n-k)^2}{p} + nk - k^2 \end{aligned} \tag{6.54}$$

If there is no NULL partition, then $k = 0$ and the formula is simplified to

$$\begin{aligned} C_{Partitioning(SMPS)} &= \frac{1}{2}0^2 + \frac{1}{2}\left(\frac{n-0}{p}\right)^2 p + \left(\frac{(n-0)}{p}0\right)p \\ &= 0 + \frac{1}{2}\left(\frac{n}{p}\right)^2 p + 0 \\ &= \frac{1}{p}\frac{1}{2}n^2 \end{aligned} \tag{6.55}$$

thus showing the performance boost by partitioning. This effect has been verified in experiments (see Chapter 7).

Another special case is the case, when we partition by a key, and additionally allowing for a NULL partition. Then, the number of partitions p is $p = n - k$, thus simplifying the formula to

$$\begin{aligned} C_{Partitioning(SMPS)} &= \frac{1}{2}k^2 + \frac{1}{2}\left(\frac{n-k}{n-k}\right)^2 (n-k) + \left(\frac{(n-k)}{n-k}k\right)(n-k) \\ &= \frac{1}{2}k^2 + \frac{1}{2}(n-k) + k(n-k) \\ &= -\frac{1}{2}k^2 - \frac{1}{2}k + \frac{1}{2}n + nk \end{aligned} \tag{6.56}$$

thus resulting in an asymptotic runtime of $\mathcal{O}(n)$ in the number of tuples and $\mathcal{O}(k^2)$ in the size of the NULL partition, as already stated earlier. So with just the information on the number of tuples n of the relation and the number of NULL values k in the partitioning column, we can estimate the runtime of the *Partitioning(SMPS)* algorithm.

A cost formula for *Partitioning(NPBS)* can be devised accordingly:

$$\begin{aligned} C_{Partitioning(NPBS)} &= k\log k + p\left(\frac{n-k}{p}\log\frac{n-k}{p}\right) + \frac{n-k}{p}kp \\ &= k\log k + (n-k)\log\left(\frac{n-k}{p}\right) + (n-k)k \end{aligned} \tag{6.57}$$

Complementation Considering *complementation*, we briefly consider cost functions for the *Simple Complement* algorithm and the *Partitioning Complement* algorithm.

Simple Complement: With m being the size of the largest set of tuples complementing each other (largest *maximal complementing set MCS*) and n being the number of tuples of the relation, the cost results as the sum of the costs for the two steps of the algorithm

$$C_{Simple\ Complement} = \frac{1}{2}n(n-1)2^m + n2^m \tag{6.58}$$

Partitioning Complement: As in the case for partitioning and *subsumption* we assume $k = |P_\perp|$ as the size (in tuples) of the NULL partition, n as the size of the relation and p as the number of partitions P_i, then – assuming an equal distribution of tuples among partitions – the average size of a normal partition P_i is $\frac{n-k}{p}$. In the algorithm, tuples from $|P_\perp|$ are added to each partition before building complements on the partitions, so each P_i' has a size of $\frac{n-k}{p} + k$. Furthermore we assume applying the *Simple Complement* algorithm on the partitions and that creating the partitioning can be done at practically no cost while reading the relation into memory. This leads to the general cost formula for the *Partitioning Complement* algorithm:

$$\begin{aligned} C_{Partitioning\ Complement} &= p\left(\frac{1}{2}\left(\frac{n-k}{p} + k - 1\right)\left(\frac{n-k}{p} + k\right)2^m + \left(\frac{n-k}{p} + k\right)2^m\right) + pk \\ &= 2^m p\left(\frac{1}{2}\left(\frac{n-k}{p} + k - 1\right)\left(\frac{n-k}{p} + k\right) + \left(\frac{n-k}{p} + k\right)\right) + pk \end{aligned} \tag{6.59}$$

If there is no NULL partition, then $k = 0$ and the formula simplifies to

$$\begin{aligned} C_{Partitioning\ Complement} &= p\left(\frac{1}{2}\left(\frac{n-0}{p} + 0 - 1\right)\left(\frac{n-0}{p} + 0\right)2^m + \left(\frac{n-0}{p} + 0\right)2^m\right) + p0 \\ &= p2^m\left(\frac{1}{2}\left(\frac{n}{p} - 1\right)\frac{n}{p} + \frac{n}{p}\right) \\ &= \frac{1}{2}\frac{1}{p}n^2 2^m + \frac{1}{2}n2^m \end{aligned} \tag{6.60}$$

Considering a second special case, where we partition by a key, and additionally allow for a NULL partition, then the number of partitions p is $p = n - k$ and the cost formula results in:

$$\begin{aligned} C_{Partitioning\ Complement} &= (n-k)\left(\frac{1}{2}\left(\frac{n-k}{n-k}+k-1\right)\left(\frac{n-k}{n-k}+k\right)2^m + \left(\frac{n-k}{n-k}+k\right)2^m\right) + (n-k)k \\ &= 2^m(n-k)\left(\frac{1}{2}(1+k)k + (1+k)\right) + nk - k^2 \\ &= 2^m(n-k)(\frac{3}{2}k + \frac{1}{2}k^2 + 1) + nk - k^2 \end{aligned} \tag{6.61}$$

Conflict Resolution and Data Fusion Conflict resolution is used as part of the *data fusion* operator which is used in *data fusion queries*. Therefore we only provide a cost formula for *data fusion* and include the cost for *conflict resolution* in it. However, because of the conflict resolution functions possibly being complex, we include its cost as a function $c(n)$ in the general cost formula for *data fusion*. With g being the number of groups that are built and given as the number of distinct real-world objects as determined by duplicate detection, $\frac{n}{g}$ being the average size of such a group, and n being the number of tuples in the relation, the cost formula for *data fusion* is:

$$\begin{aligned} C_{Data\ Fusion} &= n\log n + g\left(\frac{n}{g}\log\frac{n}{g} + \frac{1}{2}(\frac{n}{g})^2 + c(\frac{n}{g})\right) \\ &= n\log n + n\log\frac{n}{g} + \frac{1}{g}\frac{1}{2}n^2 + gc(\frac{n}{g}) \end{aligned} \tag{6.62}$$

We assume that we employ a grouping algorithm that runs in $\mathcal{O}(n\log n)$ and groups tuples by sorting them. Simple aggregation functions such as MIN or MAX run in linear time. More complex functions such as MOST GENERAL CONCEPT have a more complicated cost formula. If we estimate conflict resolution as being quadratic in the number of tuples – which will be a slight overestimation in the most cases – then $c(n) = n^2$ and the cost of data fusion results in

$$\begin{aligned} C_{Data\ Fusion} &= n\log n + g\left(\frac{n}{g}\log\frac{n}{g} + \frac{1}{2}(\frac{n}{g})^2 + (\frac{n}{g})^2\right) \\ &= n\log n + n\log\frac{n}{g} + \frac{1}{g}\frac{3}{2}n^2 \end{aligned} \tag{6.63}$$

These cost estimates are a first step in order to devise a more sophisticated cost model that uses more metadata on the relations. As we see further on in the experiments, runtime depends on the way how subsuming or complementing tuples are spread across the relation, it is influenced by the total amount of NULL values in the relation and in particular on the skew in the filling of the buckets (in case of the *Null-Pattern* variants) and the filling of the partitions (in case of the *Partitioning* variants). Therefore, the given cost formulas are merely a starting point towards the integration of the operators in physical database optimization.

6.3.2 Estimating Selectivities

The second component that is needed in order to estimate the cost of a data fusion query is an estimation of the selectivity of an operator. Specifically we want to define selectivity estimates for the operators defined earlier on, namely *subsumption*, *complementation* and *data fusion*.

Selectivity is given as the output cardinality divided by the input cardinality of an operator. For example, if a *selection* operator filters 20% of the tuples out, letting 160 tuples out of 200 input tuples pass through, then selectivity $\mathcal{S}$ of the *selection* operator is given by

$$\mathcal{S}_\sigma = \frac{160}{200} = 0.8 = 80\% \tag{6.64}$$

Given such a selectivity factor $\mathcal{S}$, the output cardinality of an operation is computed by multiplying it with the input cardinality.

$$|op(R)| = \mathcal{S}_{op}|R| \tag{6.65}$$

In the case of subsumed and complementing tuples, selectivity estimation can be accomplished by estimating how many tuples will be subsumed or complemented. Subsumed and complementing tuples are both based on the existence of NULL values in the tuples. Thus as a starting point we will estimate the number of tuples that may be subsumed/complemented by computing probabilities of NULL values existing in tuples.

For example consider the case of a simple relation with one attribute. All tuples that are NULL in that single attribute are subsumed by the others. The number of NULL values can easily be deduced from the relations histogram and the probability $\mathcal{P}(a_1 = \perp)$ that a tuple has a NULL value in attribute a_1 can be determined likewise. As this information (probability of an attribute being NULL) can easily be determined by looking at the histogram we briefly present an easy model for estimating selectivity based on this limited information.

However, estimating the exact number of subsumed tuples in general is a difficult task, as our experiences with real-world datasets show that first, this number can vary a lot and second NULL values – which highly influence this number – are not always evenly distributed among tuples and columns. For example, if two tables are combined by *outer union*, missing attribute are padded by NULL values and thus introducing tuples where NULL values in these attributes are not independent. Thus, a more detailed selectivity estimation would require quite specific information about the occurrence of subsumed and complementing tuples, of the dependencies between attributes, of the dependencies between tuples, and in general of the distribution of NULL values among the relation. As such information is difficult to acquire or requires to execute the operation itself, we stick with the easy model.

In the following we briefly sketch a way of estimating the number of subsumed tuples by calculating the likelihood that a tuple in a relation is subsumed, given a) an equal distribution of values in an attribute and b) independence among attributes. We estimate the number of subsumed or complementing tuples by computing the number of tuples that contain NULL values, as this is the precondition for a tuple to be subsumed or complementing. However, this way, we might overestimate the number of actual subsumed or complementing tuples because for a tuple being subsumed there must exist another tuple that coincides in an NON-NULL attributes. Nevertheless, given a large enough number of tuples in a relation, this is likely.

Given a relation with two attributes a_1 and a_2, and the likelihood that $a_1 = \perp$ by $p_1^\perp$ and $a_2 = \perp$ by $p_2^\perp$ respectively. Then the likelihood that a tuple contains NULL values is given as follows:

$$\begin{aligned}\mathcal{P}_2 &= \mathcal{P}(\text{tuple contains at least one NULL value})\\ &= \mathcal{P}(a_1 = \perp \vee a_2 = \perp)\\ &= 1 - \mathcal{P}(a_1 \neq \perp \wedge a_2 \neq \perp)\\ &= 1 - \mathcal{P}(a_1 \neq \perp)\mathcal{P}(a_2 \neq \perp)\\ &= 1 - (1 - p_1^\perp)(1 - p_2^\perp)\end{aligned} \tag{6.66}$$

Expanding this formula to the case of k attributes a_i results in the following formula for the likelihood that a tuple contains at least one NULL value:

$$\begin{aligned}\mathcal{P}_k &= (\text{tuple contains at least one NULL value})\\ &= 1 - \mathcal{P}(\text{all } a_i = \perp)\\ &= 1 - \mathcal{P}(\bigwedge_{i=0}^{k} a_i \neq \perp)\\ &= 1 - \prod_{i=1}^{k}(1 - p_i^\perp)\end{aligned} \tag{6.67}$$

with $p_i^\perp$ being the likelihood of a NULL value in column a_i. This likelihood can be approximated by the count of NULL values in the attributes from the relations histogram. If all the values, including the NULL value, in an attribute are equally distributed then $p_i^\perp = \frac{1}{|a_i|}$ with $|a_i|$ being the number of different attribute values in a_i. This results in

$$\mathcal{P}_k^= = 1 - \prod_{i=1}^{k}(1 - \frac{1}{|a_i|}) \tag{6.68}$$

Selectivity estimation for *subsumption* and *complementation* then is determined using probability P_k:

$$\mathcal{S}_\beta = \mathcal{S}_\kappa = 1 - \mathcal{P}_k = \prod_{i=1}^{k}(1 - p_i^\perp) \tag{6.69}$$

for operators β and κ and $p_i^\perp$ being the likelihood of a NULL value present in column a_i. Additionally, in both cases the lower bound for selectivity is given by the maximum of distinct values over all attributes, as this marks the number of tuples that are left over at least:

$$\begin{aligned} \mathcal{S}_\beta &\geq \frac{\max|a_i|}{|\mathrm{R}|} \\ \mathcal{S}_\kappa &\geq \frac{\max|a_i|}{|\mathrm{R}|} \end{aligned} \tag{6.70}$$

As for the *data fusion* operator, selectivity is determined by the number of real-world objects, as specified by the identifying attributes in parameter F. This results in the following formula:

$$\mathcal{S}_{\phi_{F,CR,S}} = \frac{|\delta\,(\pi_F\,(\mathrm{R}))|}{|\mathrm{R}|} \tag{6.71}$$

In contrast to the estimates for *subsumption* and *complementation*, this is an exact value, as each distinct combination of values from the identifying attributes F finally results in one single result tuple. The selectivity estimates as briefly presented here serve only as a starting point for a more detailed exploration of selectivity estimation of the operators defined earlier on.

There are three principal means of acquiring knowledge. Observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination.

(Denis Diderot)

7 Experimentation

So far we have discussed the implementation (see Chapter 5) and optimization (see Chapter 6) of data fusion techniques. To demonstrate the feasibility and scalability of our methods we evaluate the different methods on artificial as well as on real-world datasets. This chapter shows the results of experiments with the different operator implementations, and compares them to algorithms from the literature. It also evaluates the optimization techniques introduced in Chapter 6. We first introduce the datasets that we used in our experiments and the accompanying computational environment in Section 7.1, and then evaluate the different data fusion techniques. We report on experiments on *subsumption* and *minimum union* in Section 7.2, on *complementation* and *complement union* in Section 7.3, and finally on *conflict resolution* and *data fusion* in Section 7.4. Section 7.5 ends the chapter with experiments on the performance gain by applying transformation rules from Chapter 6.

7.1 Data and Setting

We experimented both with synthetic and real-world data and provide a summary of the datasets subsequently used in Table 7.1. The synthetic datasets have been artificially created especially for the experiments whereas the real-world datasets have been in use in other projects within our research group. Some of the datasets consists of only one table (e.g., ACTORS), while others consist of several tables (e.g., GEN1).

dataset	size	%subsumed	%complement	#columns	%nulls
CDDB	≈ 10k	0.1 – 0.4	0.02 – 0.05	6 – 7	21 – 24
ACTORS	≈ 3.6M	0.5	0.5	12	53
MOVIES	≈ 500k	0.04	0.8	27	81
CRM	≤ 1M	0.03 – 0.8	0	5 – 8	13 – 40

(a) Real-world datasets used in the experiments

dataset	size	%dupl.	%subs.	%compl.	#columns	%nulls
GEN1	10k - 5M	1/5/10	1/5/10	n/a	6	5
GEN2	10k - 5M	1/5/10	1/5/10	1/5/10	6/20/40	40
TCPH-10M	150 – 2k	1/5/10/20/50	n/a	n/a	8 – 9	0
TCPH-1G	150k – 2M	1/5/10/20/50	n/a	n/a	8 – 9	0

(b) Artificial datasets used in the experiments

Table 7.1: Summary of properties of datasets that have been used in the experiments.

7.1.1 Artificial Datasets

To evaluate our algorithms based on data with different characteristics, we implemented a data generator that allows to generate datasets of various sizes with a controllable amount of exact duplicates, subsumed

id	colo	col1	col2	col3	col4	col5	#nulls	duplicate?	subsumed?
8	8	48	43	15	876	⊥	1	1	0
35372	8	48	43	15	876	⊥	1	1	0
9	9	977	71	⊥	⊥	838	2	0	0
10	10	784	96	67	1146	364	0	0	0
82831	10	⊥	96	67	⊥	364	2	0	1
11	11	⊥	65	11	716	428	1	0	0
12	12	369	20	⊥	4375	641	1	0	0
13	13	692	73	⊥	3854	35	1	0	0

Table 7.2: Example data as generated by the data generator. COLO serves as real-world ID.

tuples, and complementing tuples. When not mentioned otherwise, the generated datasets consist of integer data in six columns, with an additional column that serves as tuple identifier. One of the generated columns simulates a real-world ID, i.e., representations of the same real-world object have the same NON-NULL value in this column.

We varied the number of tuples per generated dataset from 10,000 to 5 million (1 million in case of complementation) in order to test for scalability. We further varied the percentage of duplicated, subsumed, and complementing tuples from 1% to 5% to 10% for different amounts of overall NULL values (5% and 40%; only 40% for complementation). We also varied the number of colums (6, 20, or 40 columns; only 6 in case of complementation). For evaluation, we annotated each generated tuple with a status (is subsumed, is duplicate, etc.) and precomputed the number of NULL values in the tuple to test the influence of different sort orders (subsumed tuples first, tuples with less NULL values last, etc.). These artificially generated datasets are summarized in rows labeled GEN1 and GEN2 in Table 7.1. Each dataset consists of multiple tables and they differ in some of their parameters. Table 7.2 shows sample output from one of these tables where we can see an exact duplicate (id 8 and id 35372) and subsumed tuples (id 10 and id 82831).

The TPC-H benchmark[1] is a decision support benchmark that consists of a data generator and queries that are run against the generated data. It models customers and their orders of products. We also employ the TPC-H benchmark data in evaluating our algorithms and techniques. This is done by artificially polluting some of the tables (CUSTOMER and ORDERS) that are generated by the data generator and then running data fusion queries on the data. Tables are polluted in the following way: the customer table and the order table are polluted by introducing exact duplicates, subsumed and complementing tuples. This is done in a way that we can control the percentage of duplicated, subsumed, complemented tuples and by randomly choosing a tuple and adding a duplicated, subsumed, or complementing tuples, introducing NULL values where necessary. We did experiments with two different sizes of the TPC-H data, one of 10MBytes and one of 1GByte. We used the data from the CUSTOMER and ORDERS tables. They contain 150 tuples (150.000 tuples, respectively) in the CUSTOMER table and 1.500 tuples (1.500.000 tuples, respectively) in the ORDERS table. We varied the percentage of NULL values introduced and the number of duplicates and complementing tuples. Table 7.1 summarizes the properties of the benchmark data used in the experiments.

7.1.2 Real-world Datasets

To validate our approaches on real-world data, we used data from three real-world datasets. The first, called CDDB, is a sample of 10,000 CDs from FreeDB[2]. The second dataset is an integrated dataset of actor and movie information integrated from IMDB[3], Filmdienst[4], and three other smaller movie sources found on the web. The third dataset is from a CRM domain with an industry partner, whose identity cannot be disclosed due to privacy issues. These real-world datasets mainly store strings and float. The characteristics of the real-world datasets are also shown in Table 7.1.

[1] TPC-H benchmark: `http://www.tpc.org/tpch/`
[2] FreeDB: `http://www.freedb.org`
[3] IMDB: `http://www.imdb.com`
[4] Data kindly provided by Filmdienst: `http://film-dienst.kim-info.de`

7.1.3 Setting

A 2.3GHz dual processor quad core server with 16GBytes of main memory was used to store the data in a relational database (IBM DB2 v9.5) and to perform the experiments that study the runtime of different algorithms on varying datasets. Unless otherwise noted, reported runtimes are median values over five runs. All algorithms are implemented in Java 1.6 using a variety of open source frameworks. Most prominently we used the parser generator ANTLR[5] to parse queries and version 1.0 of the XXL framework[6] from the University of Marburg (den Bercken et al., 2001).

7.2 Results on Subsumption

To test the algorithms for computing the *subsumption* operator, we implemented all algorithms mentioned in Section 5.1, including two partitioning variants: one that uses the *Null-Pattern-Based* algorithm from related work as subroutine to perform *subsumption* whereas the other variant uses the *Simple* algorithm. In the following figures we denote these variants as *Partitioning(Null-Pattern-Bases)* and *Partitioning(Simple)*, respectively. For comparison, we also evaluated the *Simple* algorithm and the *Null-Pattern-Based* algorithm without partitioning and two variants of subsumption expressed as an SQL statement. One is a straightforward and generally applicable implementation using a `NOT EXISTS` subquery and the other one is only applicable if a favorable ordering exists (Rao et al., 2004). We experimented both with synthetic and real-world data. Table 7.1 – shown earlier – provides a summary of the datasets subsequently used. We now report on the runtime obtained for different algorithms and settings.

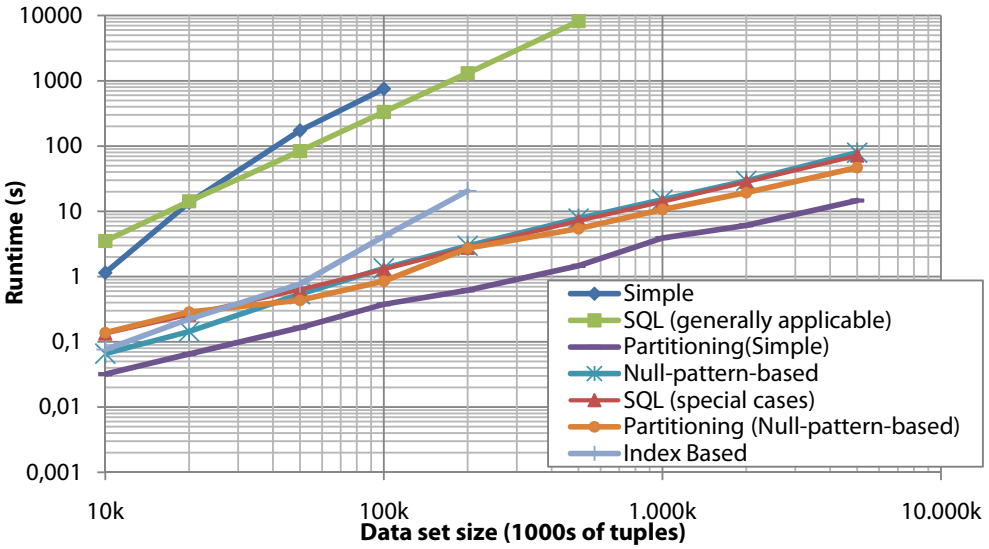


Figure 7.1: Comparing subsumption algorithms on the GEN2 dataset with 5% subsumed tuples

Experiment 1 (ALGORITHM COMPARISON)
The goal of this experiment is to compare different implementations of the subsumption *operator in terms of runtime on artificial data.* □

Methodology. We consider seven different implementations of subsumption: The *Simple* algorithm and the generally applicable SQL statement serve as baseline algorithms. The specifics of the artificial data also allow to use the SQL rewriting for special cases as described in (Rao et al., 2004) because a favorable ordering exists. The remaining four implementations are the *Null-Pattern-Based* algorithm, the *Index-Based* algorithm, and the two partitioning variants *Partitioning(Simple)*, and *Partitioning(Null-Pattern-Bases)* as described earlier on. Figure 7.1 shows the runtime (in seconds) obtained on dataset GEN2 when varying the size between 10,000

[5] AntLR parser generator: `http://www.antlr.org`
[6] XXL library: `http://www.xxl-library.de`

and 5 million tuples and setting the percentage of subsumed tuples to 5%. The heuristic given in Section 5.1.3 is used to choose the partitioning column. In this case, it is column COLO which contains the real-world identifier, resulting in many very small partitions.

Discussion. The partitioning technique presented in Section 5.1.3 used together with the *Simple* and/or *Null-Pattern-Based* algorithm clearly outperforms both the *Simple* algorithm and the generally applicable SQL statement and is able to handle up to five million tuples in less than 100 seconds. On relatively small datasets, e.g., for 100,000 tuples, the difference in runtime between the baseline algorithms (*Simple* and the general applicable SQL statement) and the partitioning variants is already two orders of magnitude. The special case SQL rewriting is applicable for this dataset and performs comparable to the partitioning variants and the *Null-Pattern-Based* algorithm.

We also make the following observations: For all but small dataset sizes, the partitioning technique applied to the *Null-Pattern-Based* algorithm improves runtime compared to the *Null-Pattern-Based* algorithm alone. The same is even more true when comparing the results of *Simple* and *Partitioning(Simple)*. The *Index-Based* algorithm performs comparably well to the partitioning technique for small dataset sizes, but not for larger dataset sizes (see Figure 7.1). Additionally, because of its specific data structure, the *Index-Based* algorithm uses too much memory for large dataset sizes to be competitive in our experiments. However, it outperforms the two baseline algorithms. Further experimentation also showed that total runtime heavily depends on the percentage of NULL values in the dataset and also on the percentage of subsumed tuples. The time used for precomputation (building the index) is stable in these experiments, but the time actually removing subsumed tuples varies. However, even this time alone (without index building) cannot compete against the partitioning technique for larger datasets.

Experiment 2 (INFLUENCE OF PARTITIONING)
In the previous experiment, Simple performs best, but as this experiment demonstrates, its runtime highly depends on the size of the NULL partition and, to a lesser extent, on the number of partitions. □

Methodology. We used two GEN2 datasets with 5% of subsumed tuples and 20,000 tuples. The two datasets differ in the numbers of distinct and NULL values per column and therefore in the size of the NULL partitions. Choosing different columns for partitioning results in different numbers of partitions. Figure 7.2 shows the runtimes.

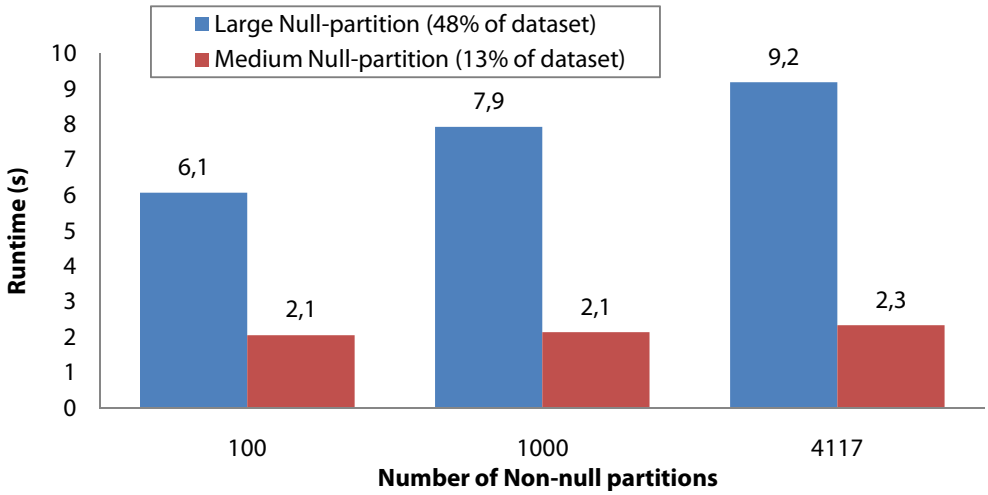


Figure 7.2: Comparing runtime of Partitioning (Simple) when using three different columns for partitioning and using two tables (difference in size of NULL partition) of datasets GEN2 with 5% subsumed tuples and a size of 20,000 tuples.

Discussion. We see in Figure 7.2 that a poor column choice significantly increases the runtime of *Simple* and using the *Null-Pattern-Based* algorithm instead would have been the better choice in this case. Indeed, the first observation is that if the partitioning column has a large percentage of NULL values (large NULL partition

in the figure), the runtime is significantly higher than when choosing a column that yields a smaller NULL partition. The reason for this behavior is that every tuple in the NULL partition is compared to every other tuple in every other NON-NULL partition. This behavior supports our theoretical considerations in Section 5.1. In further experiments, e.g., the one reported in Figure 7.7 we also observe that when the size of the NULL partition is comparable, runtime decreases with an increasing number of partitions, which is also in accord with our theoretical analysis.

Experiment 3 (PRECOMPUTE TIME VS. RUNNING TIME)
In our experiments we generally report on total runtime, including the initialization of data structures, necessary precomputations, and the computation itself. In this experiment, we study how the overall runtime splits up among these three phases of our algorithms. □

Methodology. We perform time measurements on a version of GEN2 with six attributes, 100,000 tuples and 5% of subsumed tuples. Results are reported in Table 7.3.

	Simple	Null-pattern-based	Partition (Simple)	Partition (Null-pattern-based)	Index-based
init	10 ms	10 ms	257 ms	180 ms	10 ms
precompute	0	0 ms	66 ms	44 ms	1386 ms
runtime	754770 ms	1333 ms	53 ms	628 ms	2438 ms
total time	754780 ms	1343 ms	376 ms	852 ms	3834 ms

Table 7.3: Comparison of initialization time, precomputation time, and runtime on a table of dataset GEN2 with 5% of subsumed tuples and a size of 100,000 tuples.

Discussion. We see that the algorithms differ in their distribution of total runtime among the three phases. Whereas the *Null-Pattern-Based* algorithm spends most time actually computing the subsumption, the partitioning variants spend as expected a considerable amount of time in collecting statistics (init) to determine the partitioning attribute and to partition the data itself (precompute). The time spent in these two phases considerably decreases the actual runtime. The differences in init and precompute between the two partitioning variants are negligible, but the runtime comparison shows that partitioning considerably helps the *Null-Pattern-Based* as well as the *Simple* algorithm. The performance gain for the *Simple* algorithm is more significant resulting in the overall fastest runtime. The *Index-Based* algorithm spends approximately two thirds of the total time actually removing subsumed tuples. In further experiments we see that index building times (precompute) range anywhere from 25% to 50% of total time for different table sizes, and different percentages of subsumed tuples. Thus, it is the algorithm with the most equal distribution between the precompute and the runtime part, also showing that particularly runtime is affected by the characteristics of the tables.

Experiment 4 (AMOUNT OF NULL VALUES IN NON-PARTITIONING COLUMNS)
In Experiment 2 we have seen that the amount of NULL values in the partitioning column significantly affects the runtime of the partitioning variants. In this experiment, we study the influence of the amount of NULL values in the remaining columns. □

Methodology. We perform two analyses: (i) We fix the type of dataset to GEN2 and vary the distribution of NULL values such that the percentage of subsumed tuples equals to 1%, 5%, and 10%, and (ii) we fix the percentage of subsumed tuples to 5% and vary the type of dataset from GEN1 to GEN2. The main difference is that GEN1 only contains 5% of NULL values over all its attributes, whereas GEN2 consists of 40% overall NULL values. Runtime results for the former case are shown in Figure 7.3 and results for the latter case in Figure 7.4.
Discussion. In Figure 7.3, we see that all three tested algorithms are relatively robust against changes in the percentage of subsumed tuples. However, common to all algorithms, we observe a slight decrease in runtime the higher the percentage of subsumed tuples, which is best observed on the graphs for *Null-Pattern-Based*. In Figure 7.4, where we vary the total percentage of NULL values while fixing the percentage of subsumed tuples, we see that the runtime again decreases the higher the amount of NULL values, most obvious in the graph for the *Partitioning(Simple)* variant. The *Null-Pattern-Based* algorithm seems most affected by the variation of overall NULL values. We explain this behavior with the nature of the algorithm.

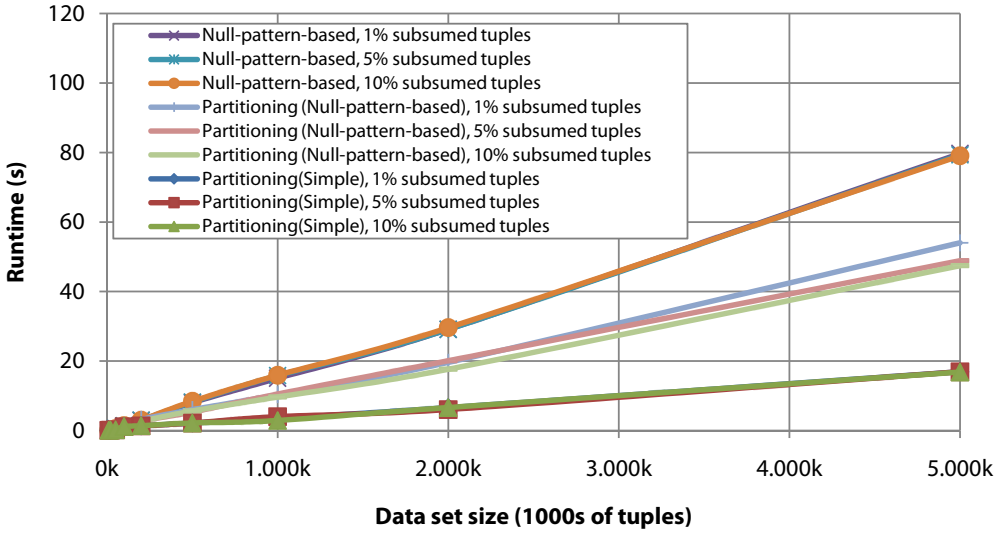


Figure 7.3: Comparing the influence of different percentages of subsumed tuples

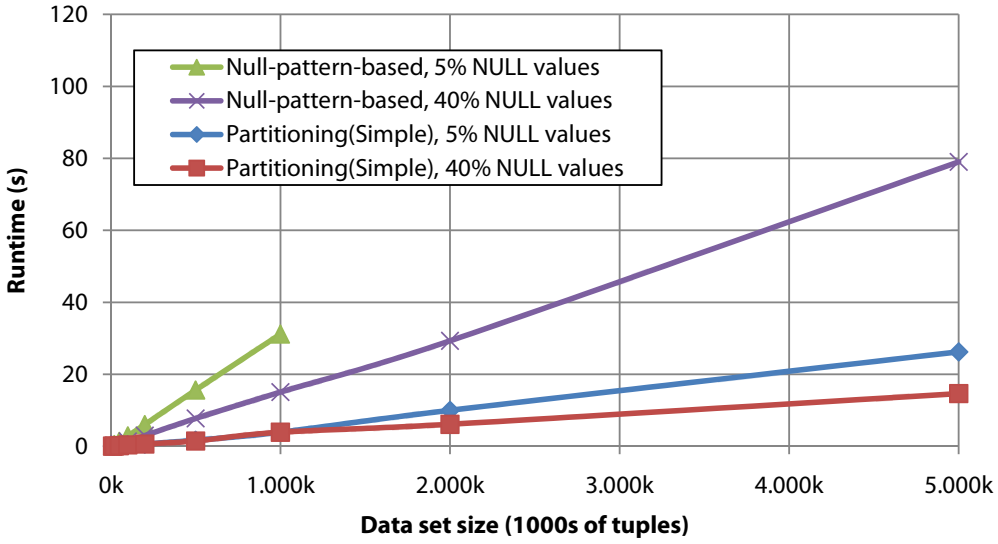


Figure 7.4: Runtime of the *Partitioning(Simple)* and *Null-Pattern-Based* algorithms on GEN1 and GEN2 with 5% subsumed tuples

Experiment 5 (VARYING THE NUMBER OF ATTRIBUTES)
In this experiment, we study how runtime varies depending on the number of attributes of a relation. □

Methodology. In the previous experiments, we observed that given a good partitioning the *Partitioning(Simple)* variant usually performs best, so here we consider only this algorithm. We use tables of dataset GEN2 of different sizes and with 6, 20, or 40 attributes per table. Runtime results are shown in Figure 7.5.

Discussion. Clearly, the more attributes a relation has, the more time is needed to perform *subsumption*. This comes as no surprise, as each pairwise tuple comparison takes longer when more attributes exist in a tuple. Nevertheless, the runtime is still acceptable: Tables with 40 attributes and 500,000 tuples are processed in roughly 16 seconds. For comparison we also plotted runtime for the SQL statement, which shows that for larger tables the performance gain by our algorithms even increases.

Experiment 6 (SUBSUMPTION ON REAL-WORLD DATA)
The previous results for subsumption *were obtained on artificially generated datasets. We now evaluate different subsumption algorithms on the different real-world datasets summarized in Table 7.1.* □

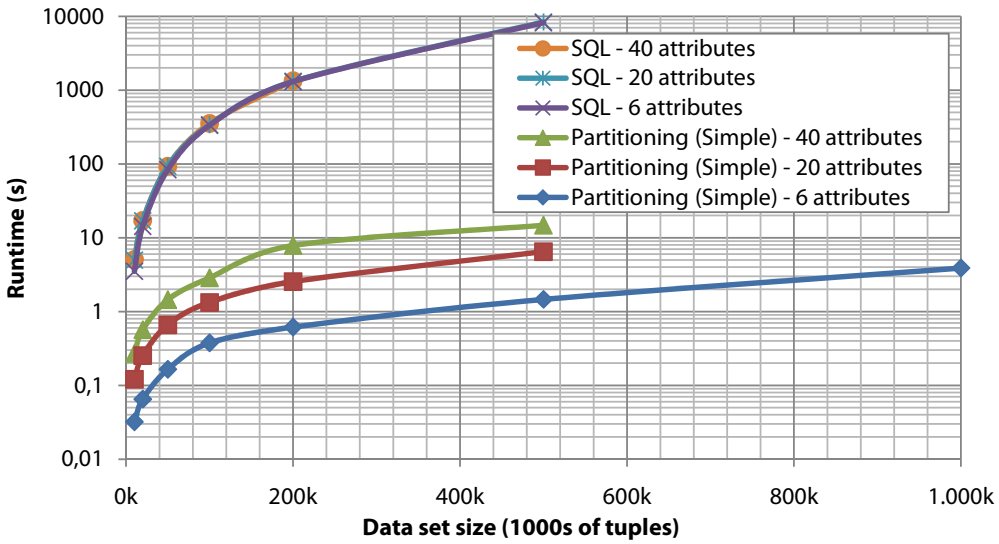


Figure 7.5: Comparing runtimes for different schema sizes for *Partitioning(Simple)* on the Gen2 dataset

Methodology. We consider the real-world datasets Actors, Movies, CDDB, and CRM. Within the CRM database, we analyze three tables (similarly to the Movies dataset from which we use both actors and movies). We denote these tables as CRM1 through CRM3. For each of these datasets, we take samples of different sizes and measure the runtime of the *Null-Pattern-Based* algorithm and the *Partitioning(Simple)* variant. Figure 7.6 reports the results for the Actors and the CRM datasets. The results on the Movies dataset are comparable to the shown results for *Partitioning(Simple)*, whereas the runtime of the *Null-Pattern-Based* algorithm is higher than for the other datasets. On the CDDB dataset, we evaluate the effect of different partitionings on the runtime of *Partitioning(Simple)* and report results in Figure 7.7.

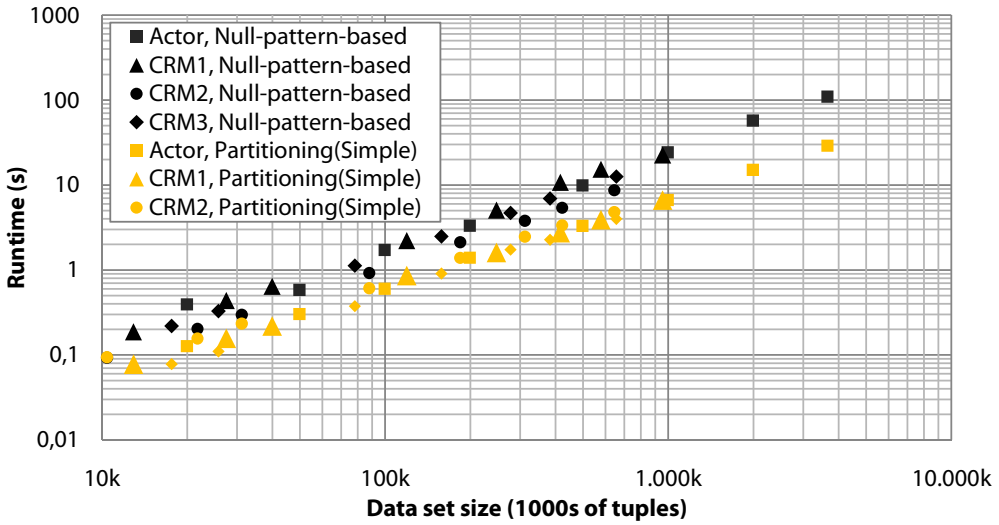
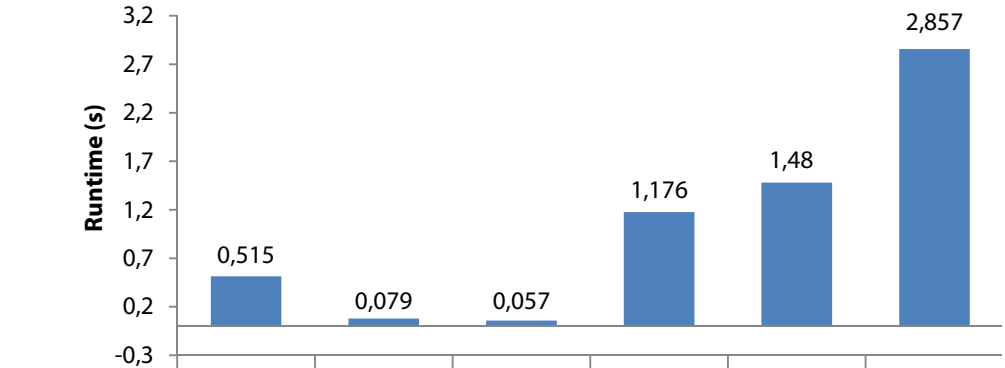


Figure 7.6: Runtime of subsumption algorithms on real-world data

Discussion. In Figure 7.6 as well as on the Movies and the CDDB dataset, we observe that in our real-world scenarios, which all have a low percentage of subsumed tuples, the runtime of *Partitioning(Simple)* is lower than the runtime of the *Null-Pattern-Based* algorithm. Runtime for the *Simple* algorithm and the Sql version are worse than both. Both algorithms exhibit an increase in runtime with respect to the increasing dataset size. The datasets considered in Figure 7.6 all have a small number of attributes (i.e.,$\leq$ 12) and we do not observe significant differences in runtime for these. On the other hand, we observe that the runtime of the *Null-Pattern-Based* algorithm on the Movies dataset with 27 attributes is significantly higher than on the other datasets. This is in accord with our theoretical analysis in Section 5.1.3. As a final observation, we see that



	CATEGORY	ARTIST	TITLE	GENRE	YEAR	CDEXTRA
#partitions	11	7275	9368	521	80	1791
size null partition	0	0	15	3394	4541	6509

Figure 7.7: Runtime for different partitioning columns in dataset CDDB

the runtime of both algorithms is higher on the real-world datasets than on generated data of same size and with comparable characteristics. This difference is mainly due to the longer comparison runtime for the string data present in the real-world datasets compared to the integer comparisons performed on the generated data.

Figure 7.7 confirms the observations of Experiment 2 as it shows that when partitioning the data based on attributes CATEGORY, ARTIST, and TITLE, which yield a small NULL partition, the runtime of *Partitioning(Simple)* is significantly less than the runtime obtained when partitioning based on attributes with large NULL partitions, e.g., GENRE, YEAR, and CDEXTRA. Furthermore, for attributes with comparable NULL partition sizes we observe that, as expected, the runtime decreases with increasing number of partitions. Runtime for *Null-Pattern-Based* is approximately 0,1s and thus slightly worse than the best cases for *Partitioning(Simple)*.

Experiment 7 (Partitioning by more than one column)
We evaluate the choice of more than one column for the partitioning algorithm and evaluate the greedy heuristics to find the best partitioning. □

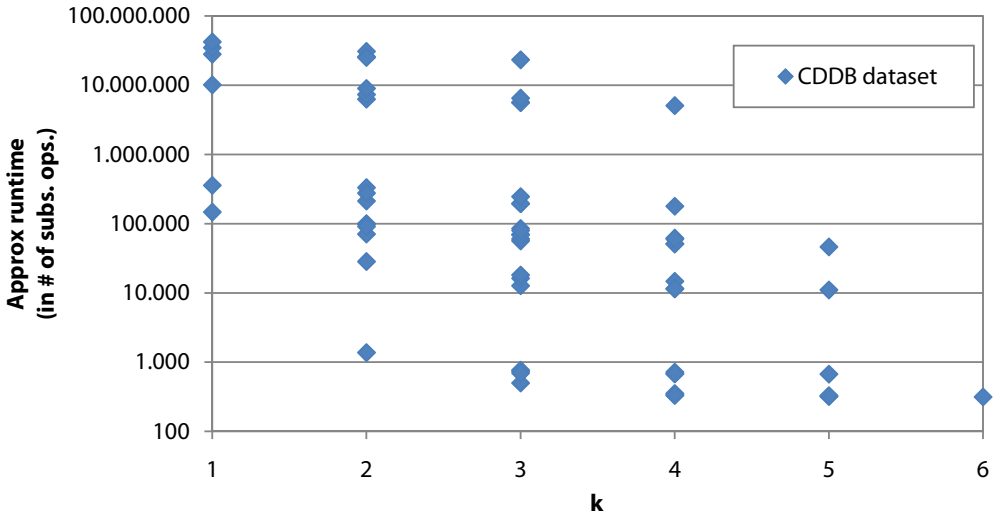
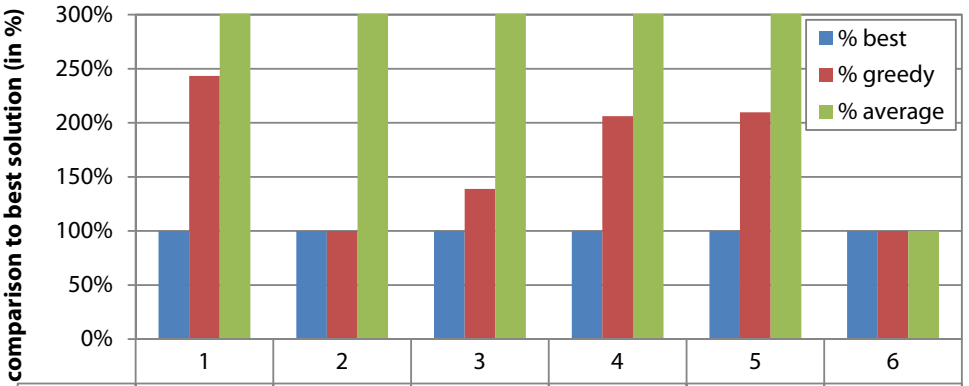


Figure 7.8: Approximated runtime in number of subsumption comparison operations needed for all combinations of k attributes, k varying from 1 to 6 for the CDDB dataset

Methodology. We use the real-world CDDB dataset for our experiments with more than one attribute when partitioning. We approximate the runtime by the formula given in Section 5.1.3 and show it for all combinations of up to $k = 6$ attributes in Figure 7.8. We also find the best combination of k attributes and

compare it to the combination found by the greedy heuristics from Section 5.1.3 and the average runtime over all possible solutions (Figure 7.9). We approximate the runtime by issuing an SQL statement on the CDDB dataset with 6 columns. The statement counts the partition sizes and computes the total numbers of subsumption comparisons needed.



k	1	2	3	4	5	6
% best	100%	100%	100%	100%	100%	100%
% greedy	243%	100%	139%	206%	210%	100%
% average	13143%	513485%	423115%	111697%	3065%	100%

Figure 7.9: Experiment on selecting partitioning attributes for the CDDB dataset (we truncated the y-axis at 300% for better readability)

Discussion. As already expected, we see in Figure 7.8 that approximated runtime differs much when comparing different possible partitions using k attributes. This underlines the importance of a good choice for the partitioning. We also see that the approximated runtimes of both the best and the worst possible solution decrease with increasing k: using more attributes for the partitioning pays off. Figure 7.9 compares the approximate runtimes of the best solution, the average runtime of all possible solutions and the approximate runtime of the solution found by the greedy heuristics relative to the best solution. For example, the solution found by the heuristic for $k = 3$ attributes is (ARTIST, TITLE, GENRE) resulting in approximately 693 comparison operations whereas the best solution is the partitioning by (ARTIST, TITLE, CATEGORY) with estimated 499 comparison operations. This results in the greedy solution being 1.39 times the best solution. Although the greedy algorithm finds the optimal solution only twice (for $k = 1$ and $k = 6$), it always finds a solution within 2.5 times the best solution. Comparing to the other possible values in Figure 7.8 this is always among the best solutions and by magnitudes better than the average solution combination.

7.3 Results on Complementation

We evaluate the algorithms discussed earlier on in Section 5.2, i.e., *Simple Complement*, and *Partitioning Complement* (PC), and compare them to algorithms adapted from the literature, i.e., *Null-Pattern-based Complement* (NPC) and others. Replacing complementing tuples by their complement in a relation is equivalent to finding all maximal cliques in a graph that has been constructed by creating one node per tuple and an edge between nodes if the corresponding tuples complement each other. We compare to a clique finding algorithm (Stix, 2004) (referred to as *Dynamic*). We also evaluate the performance of an algorithm (referred to as *Johnson*) for the dual problem of finding independent sets (Johnson et al., 1988).

Experiment 8 (ALGORITHM COMPARISON)
We compare the different algorithms for computing the complementation *operator on artificial data.* □

Methodology. We generate tables of varying size that contain a varying percentage of complementing tuples (1%, 5%, and 10%). Figure 7.10 shows the runtime for *Simple Complement*, *Partitioning Complement*, and *Dynamic*– the approach from (Stix, 2004) – for 1% and 10% of complementing tuples.

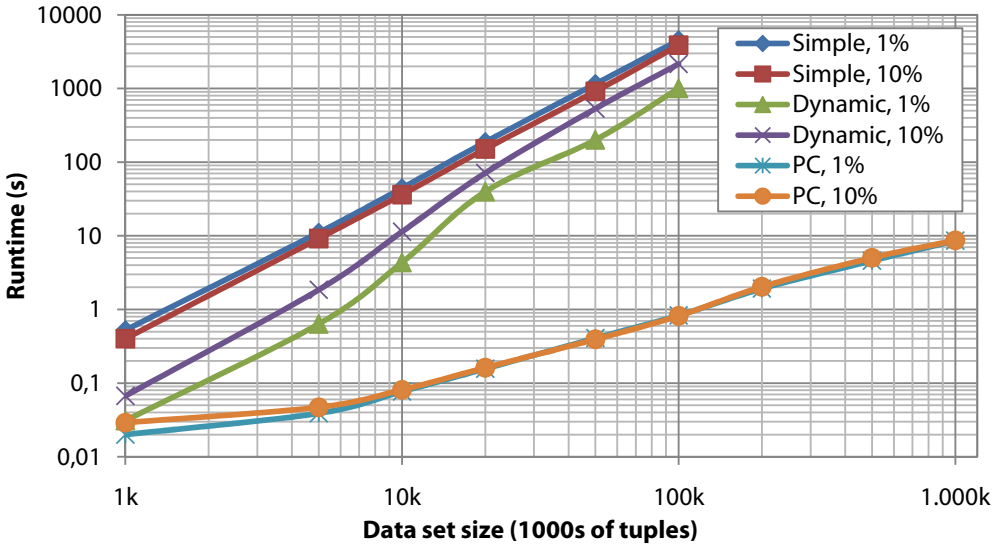


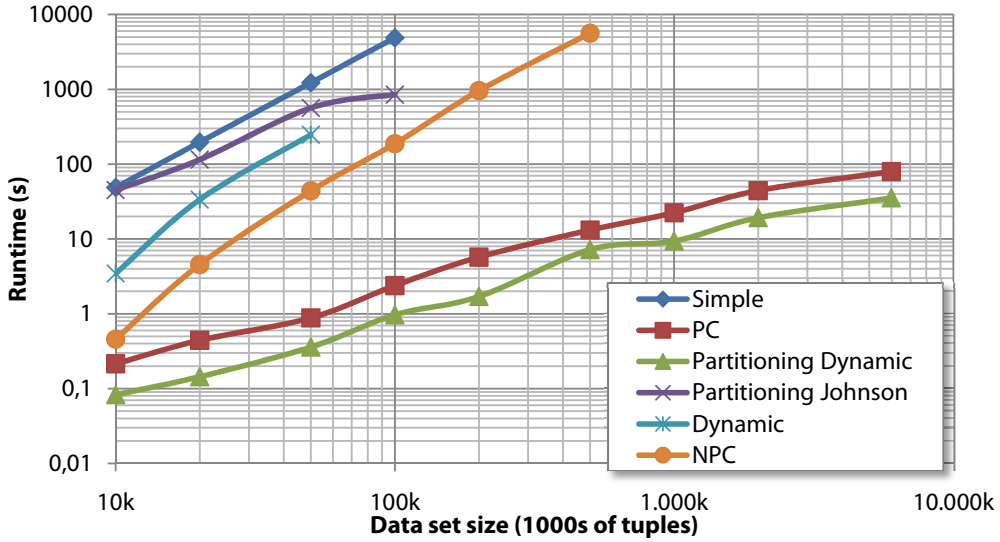
Figure 7.10: Runtimes for different percentages of complementing tuples

Discussion. We observe that *Dynamic* is considerably faster than the *Simple Complement*. However, as the runtime of *Simple Complement* decreases with increasing percentage of complementing tuples, the runtime of *Dynamic* increases. This is mainly due to the data structures used in both algorithms and the runtime of *Simple Complement* being mainly dominated by the size of the largest clique. The algorithm from (Johnson et al., 1988) (not shown) performs very poorly, even for only 1000 tuples. This is because it solves the dual problem and therefore needs to consider the inverse graph of the graph that is used by *Dynamic*. Whereas the graph used by *Dynamic* is sparse in our examples (only few complementation relationships), the graph for *Johnson* is very dense. We also observe that partitioning pays off, as *Partitioning Complement* is the fastest among the shown algorithms. However, applying partitioning to the other approaches results in a comparable runtime, also for *Johnson*. All partitioning algorithms scale well for relations of up to 1 million tuples. Varying the column for partitioning shows significant differences in runtime for all algorithms, so care must be taken in choosing a partitioning column.

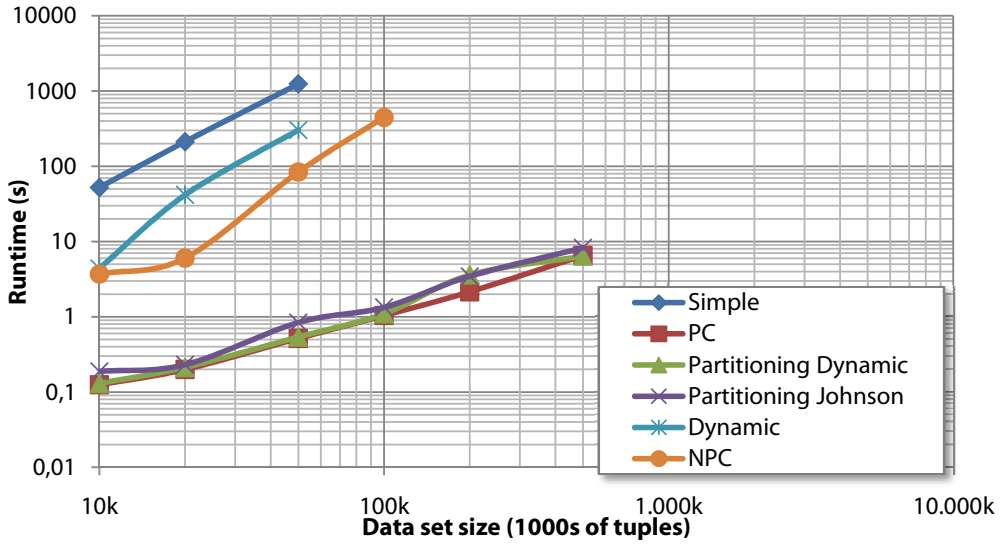
Experiment 9 (Complementation on real-world data)
We now consider the real-world datasets Actors, Movies, and CDDB. □

Methodology. For each of these datasets, we take samples of different sizes and measure the runtime of the same algorithms as in the previous experiment (Figure 7.11).
Discussion. For the CDDB dataset, which consists of only 10,000 tuples, *Null-Pattern-based Complement* takes 0,16 seconds and thereby outperforms *Partitioning Complement* (0,34 seconds with best partitioning). Results on the Actors and the Movies datasets are represented in Figure 7.11. We see that for both Actors and Movies, the *Partitioning Complement* algorithm outperforms *Null-Pattern-based Complement*. This difference is mainly due to the beneficial partitioning, although the number of columns (12 vs. 27 columns) also has some influence: *Partitioning Complement* better copes with large schemata. Interestingly, *Null-Pattern-based Complement* seems better suited for datasets with less null values (Actors) whereas results show that the opposite is the case for *Partitioning Complement*: this algorithms performs slightly better on the dataset with more overall null values (Movies). However, in this case we cannot completely rule out the influence of the larger number of attributes of Movies (27 columns).

We further investigated the runtime of related algorithms, as in the previous experiment (see Figure 7.11). The superiority of the approach from (Stix, 2004) (*Dynamic*, approx. 34 seconds for 20k tuples) over *Simple Complement* (although being worse than *Null-Pattern-based Complement* and *Partitioning Complement*) and the corresponding partitioning version over all other algorithms is mainly due to the small percentage of complementing tuples in the datasets (below 1%), which is beneficial for this algorithm. In contrast to the generated datasets, the partitioning variants show differences in runtime, especially *Johnson* performs worse than for the generated datasets (nearly 40 seconds for 10k tuples).



(a) Runtimes on the real-world dataset Actors



(b) Runtimes on the real-world dataset Movies

Figure 7.11: Runtimes for complement algorithms on two different real-world datasets

7.4 Results on Data Fusion

We evaluate the implementation of the *conflict resolution* and *data fusion* operator and show that it scales up to sizes of several million tuples on artificial as well as real-world datasets. In order to do that we employ the artificially created dataset already used in the prior experiments, the tcph-1g dataset, and the Movies as well as the Actors dataset. We evaluate the algorithms discussed earlier on in Section 5.3, i.e., the algorithms for *conflict resolution* and *data fusion*. Algorithms from the XXL library (den Bercken et al., 2001) are used as basic building blocks in order to implement them.

Experiment 10 (Scalability)
We compare runtimes of the algorithms for conflict resolution and data fusion on artificial as well as on real-world data. □

Methodology. We use the artificially generated tables also used in the previous experiments on subsumption and complementation. They are of varying size and in addition to subsumed and complementing tuples also contain a varying percentage of duplicated and modified tuples (1%, 5%, and 10%), i.e., tuples polluted in

such a way that they contain data conflicts (fuzzy duplicates). We also varied the number of columns (6, 20, 40 columns). Figure 7.12 shows the runtime for the *data fusion* implementation using algorithms from the XXL library as basic building blocks on the artificial data. In addition, Figure 7.13 shows the runtime on the real-world datasets Actors and Movies. The implementation of *conflict resolution* and *data fusion* uses a sort-based grouping algorithm as well as a sort-based algorithm for removing exact duplicates (both taken from the XXL library). Subsumed tuples are removed by the *Simple* algorithm. We also varied the conflict resolution functions used. Runtimes are median values over 10 runs.

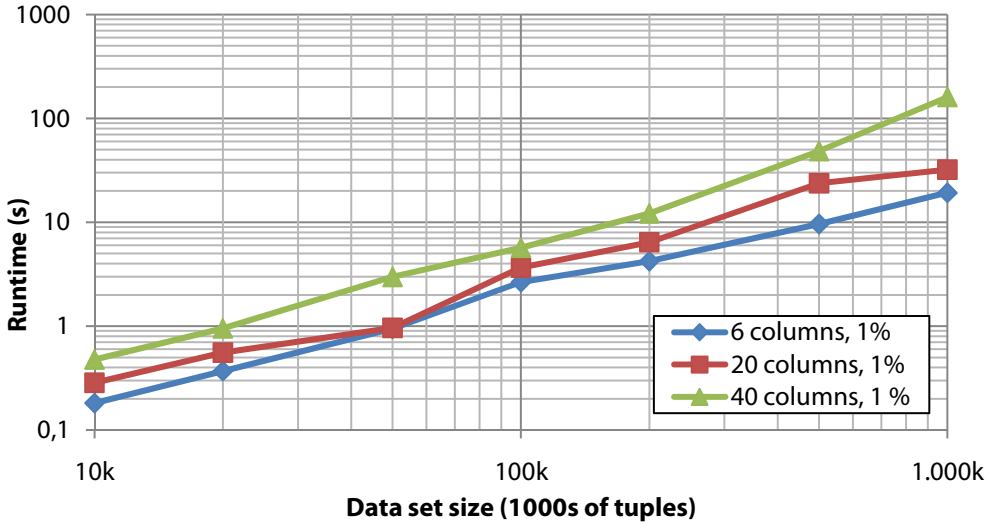


Figure 7.12: Runtimes on artificial data for tables of different numbers of columns and 1% duplicates with data conflicts

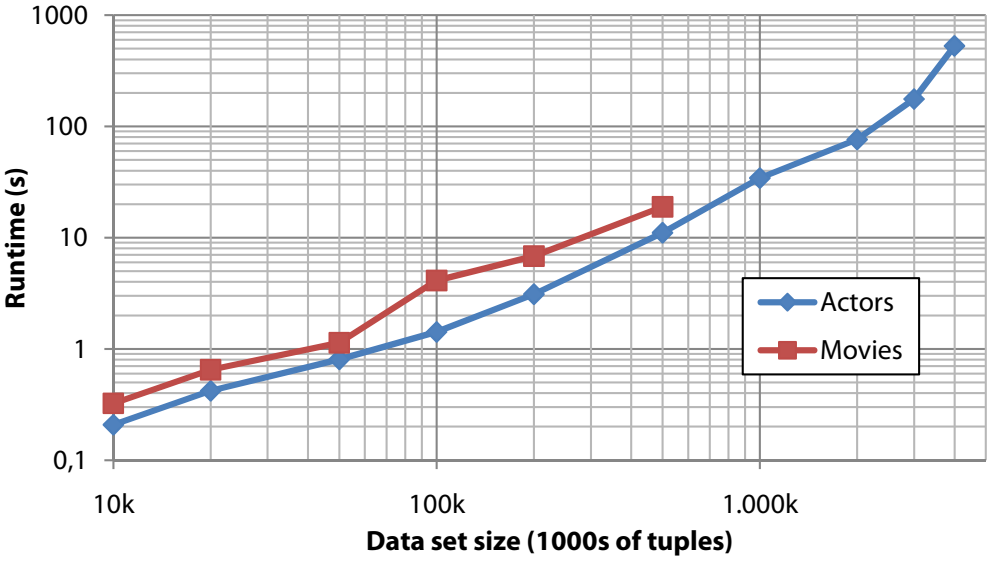


Figure 7.13: Runtimes for the two real-world datasets Actors and Movies

Discussion. As we see in Figures 7.12 and 7.13, the algorithm implementation scales up to more than 3 million tuples (the size of the Actors dataset). However, in comparison to the implementations of the *subsumption* and *complementation* operators it is generally slower. This is mainly due to the fact that using the algorithms from – and thus embedding the data fusion operator into – the XXL framework comes with a runtime penalty, in exchange for a more versatile operator implementation. We did not observe any large differences in runtime when considering different percentages of duplicated tuples or different combinations of conflict resolution functions. However, as expected and as Figure 7.12 shows, there is a slightly faster runtime for the dataset with less columns (Actors dataset).

Experiment 11 (Group sizes)
We compare the data fusion algorithm for different group sizes on artificially polluted data from the TPC-H benchmark using data set TCPH-1G. □

Methodology. We artificially polluted the CUSTOMER table of the TPC-H benchmark, which contains 150.000 tuples in the TCPH-1G dataset. We introduced duplicates with data conflicts (fuzzy duplicates) and varied the group size of the duplicate clusters. A group size of 4 means for example that a duplicated customer has 4 different representations (tuples) with data conflicts present. Runtimes are median values over 10 runs. We measure total runtime of the data fusion operation on the CUSTOMER table.

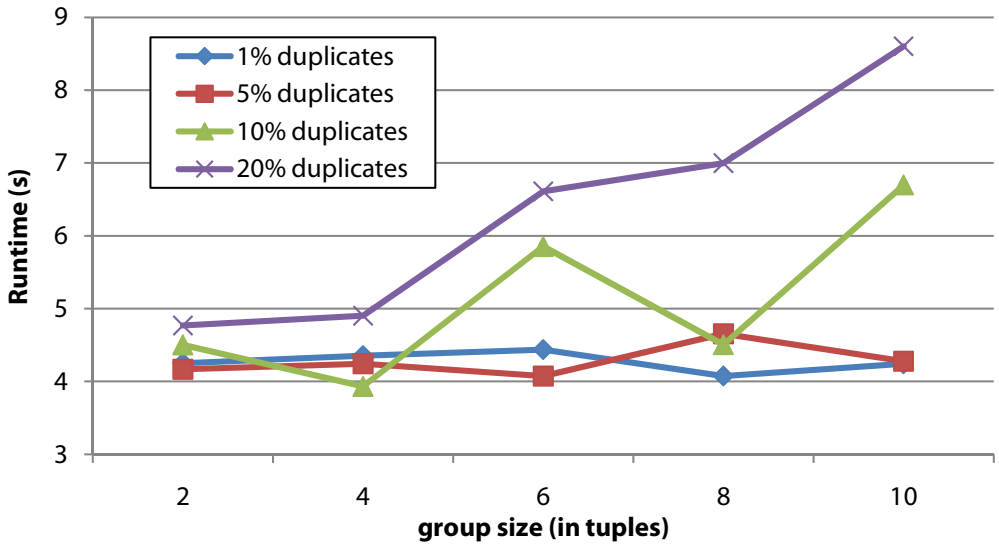


Figure 7.14: Runtimes for different percentages of duplicated tuples and different group sizes

Discussion. We observe (see Figure 7.14) that different group sizes only marginally affect the runtime of the data fusion operator if the percentage of tuples where data conflicts are present is low (see the lines for 1% and 5% duplicates in Figure 7.14). If the number of duplicate tuples rises, an increasing group size also increases total runtime. In our experiments we observe an increase of approximately factor 2 at the worst case, for a percentage of duplicated tuples of 20 and a group size of 10.

7.5 Performance Gain with Transformation Rules

In Chapter 6 we considered transformation rules to rewrite queries involving *subsumption*, *complementation*, and *data fusion* operators in order to speed up computation. We now evaluate some of these transformation rules to show the performance gain by applying them. In these experiments, we use the artificially created datasets GEN1 and GEN2.

Experiment 12 (Selection over Subsumption)
We evaluate transformation Rule 6.8 (see page 88), measuring the effect of pushing selection *down below* subsumption. □

Methodology. We measure the runtime of a set of simple queries on relation R where a *selection* follows a *subsumption* ($Q_1 = \sigma_c\ (\beta\ (\mathrm{R}))$) and compare their runtimes to the transformed queries where *subsumption* follows *selection* ($Q_2 = \beta\ (\sigma_c\ (\mathrm{R}))$). The performance gain measures the usefulness of applying this particular transformation rule. We varied the selectivity of the selection predicate from 1% to 80% as well as the percentage of subsumed tuples (1%, 5%, and 10%). We report the performance gain in Figure 7.15 for tables with 6 columns and in Figure 7.16 for tables with 40 columns. Runtimes are median values over 10 runs and the figures show the values for tables of one million tuples. Performance gain is reported as the runtime of query Q_1 divided by the runtime of query Q_2. The *Partitioning(Simple)* algorithm has been used in removing subsumed tuples.

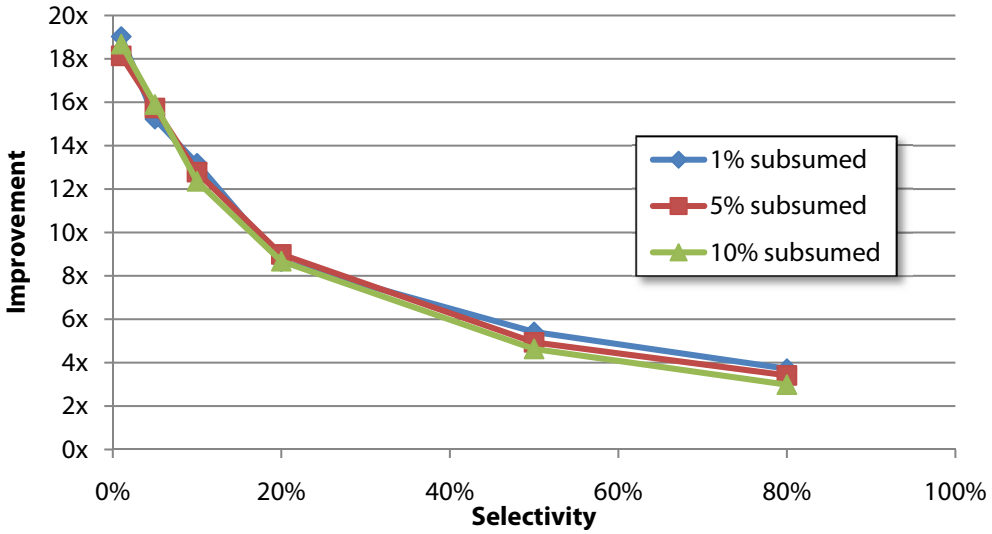


Figure 7.15: Improvement in runtime by pushing *selection* below *subsumption* for tables with 6 columns and one million tuples, regarding different selectivities and different percentages of subsumed tuples

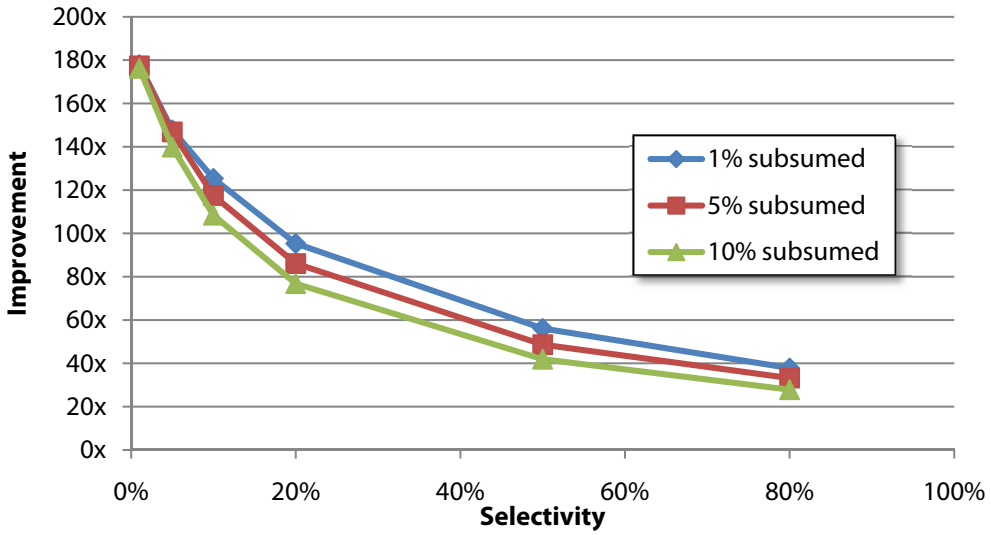


Figure 7.16: Improvement in runtime by pushing *selection* below *subsumption* for tables with 40 columns and one million tuples, regarding different selectivities and different percentages of subsumed tuples

Discussion. Applying the transformation rule and pushing *selection* down below *subsumption* is worthwhile and pays off as can be seen in Figures 7.15 and 7.16. Performance gain for tables with 6 columns (Figure 7.15) ranges from around 2-3x times for a selectivity of 80% (80% of all tuples pass the *selection*, 20% of all tuples are filtered out) up to 20x for a selectivity of 1%, indicating that the transformed query Q_2 (*selection* pushed down) is always faster for the tested combinations of selectivity, size, number of columns and percentage of subsumed tuples. Experiments with other table sizes (from 10.000 up to 1 million tuples) show similar results. Figure 7.15 also shows as expected that the performance gain is higher for lower selectivities of the *selection*, as in these cases fewer tuples pass the *selection* and are input to the *subsumption*. As can be seen in Figure 7.16, performance gain for tables with 40 columns is one order of magnitude larger than for tables with only 6 columns, ranging from around 20x up to around 200x. This is a result of *subsumption* needing to check all columns and a *subsumption* check on 40 columns being more costly than one on only 6 columns. Thus, the higher benefit when saving *subsumption* checks in the former case. The difference in performance gain between tables with different percentages of subsumed tuples can also be explained: even if, e.g., a 50% *selection* filters out half of the tuples, there will still be proportionally more subsumed tuples to be handled if

there exist 10% subsumed tuples in the table than if there exist only 1% subsumed tuples. Thus the smaller performance gain for tables with 10% subsumed tuples.

Experiment 13 (Selection over Complementation)
We evaluate transformation Rule 6.23 (see page 91), measuring the effect of pushing selection *down below* complementation. □

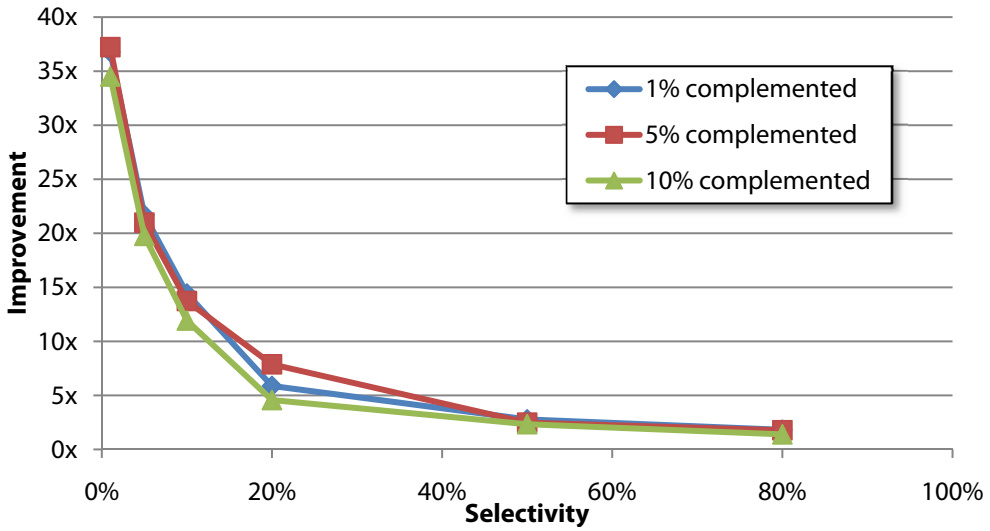


Figure 7.17: Improvement in runtime by pushing *selection* below *complementation* for tables with 6 columns and one million tuples, regarding different selectivities and different percentages of complementing tuples

Methodology. Analogously to Experiment 12, we measure the runtime of a set of simple queries where a *selection* follows a *complementation* ($Q_1 = \sigma_c\ (\kappa\ (\mathrm{A}))$) and compare their runtime to the transformed queries where *complementation* follows *selection* ($Q_2 = \kappa\ (\sigma_c\ (\mathrm{A}))$). We varied the selectivity of the *selection* (1% to 80%), and the percentage of complementing tuples (1%, 5%, and 10%).We report the performance gain in Figure 7.17 for tables with 6 columns. Runtimes are median values over 10 runs and the figure contains the values for tables of one million tuples. Performance gain is reported as the runtime of query Q_1 divided by the runtime of query Q_2. The *Partitioning Complement* algorithm has been used in combining complementing tuples.
Discussion. Pushing *selection* down below *complementation* pays off, as can be seen in Figure 7.17. Experiments with other table sizes show similar results. The performance gain for tables with 6 columns ranges from around 1.4x times for a selectivity of 80% up to around 37x for a selectivity of 1%, indicating that the transformed query Q_2 (*selection* pushed down) is always faster for the tested combinations of selectivity, size, and percentage of subsumed tuples. Figure 7.17 also shows that the performance gain is higher for lower selectivities of the *selection*, as in these cases fewer tuples pass the *selection* and are input to the costly *complementation* operation. The difference in performance gain between tables with different percentages of complemented tuples is small, but consistently existent, except the outlier for 20% selectivity. The explanation for these differences is analogous to the subsumption case.

Experiment 14 (Join over Subsumption)
We evaluate transformation Rule 6.10 (see page 88) that allows the exchange of subsumption *and* join. □

Methodology. We measure the runtime of a set of simple *join* queries (*key/foreign-key join*) where a *subsumption* follows a *join* ($Q_1 = \beta\ (\mathrm{R} \bowtie_c \mathrm{S})$) on key column c and compare their runtimes to the transformed queries where the *join* follows *subsumption* ($Q_2 = \beta\ (\mathrm{R})\ \bowtie_c \beta\ (\mathrm{S})$). To test different scenarios, we varied the selectivity of the *join* from 1% to 80% as well as the percentage of subsumed tuples (1%, 5%, and 10%).We report the performance gain in Figure 7.18 for tables with 6 columns. Runtimes are median values over 10 runs and the figure shows the values for tables of 20.000 tuples. Performance gain is reported as the runtime of query

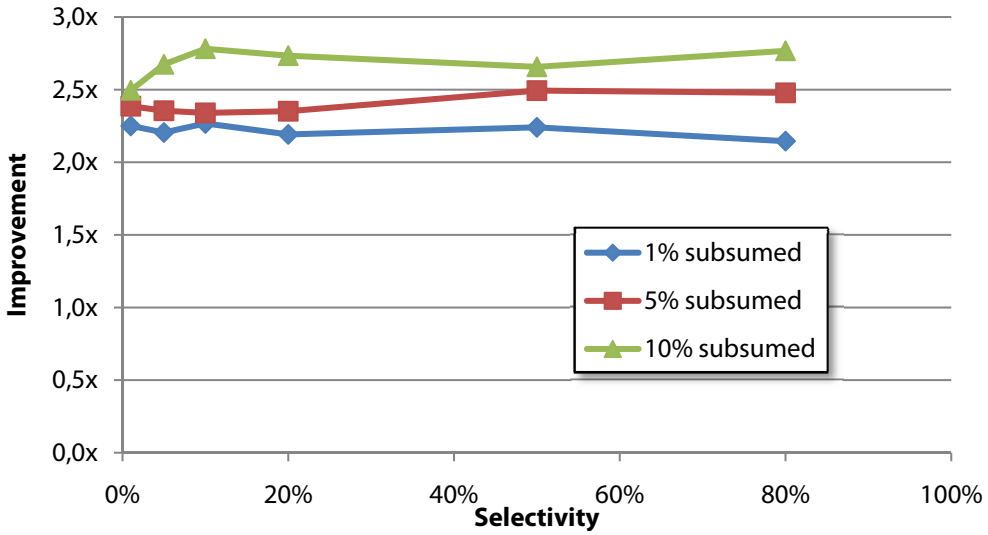


Figure 7.18: Improvement in runtime by pushing *subsumption* below *join* for tables with 6 columns and 20.000 tuples, regarding different join selectivities and different percentages of subsumed tuples.

Q_2 divided by the runtime of query Q_1. The *Simple* algorithm has been used in removing subsumed tuples together with the NL-Join implementation of the XXL framework (den Bercken et al., 2001).
Discussion. Figure 7.18 shows an improvement of around 2.5x for all selectivities and the three percentages of subsumed tuples. Experiments with other table sizes show similar results. Improvement varies a bit but we see a tendency to a general and uniform performance gain. This is mainly due to the difference in total runtime of the two operators: whereas the *subsumption* operator is optimized for performance and needs only a few milliseconds, the main part of the queries total runtime is consumed by the implementation of the *join* of the XXL framework (several seconds). This implementation is constructed for high reusability and to allow a wide variety of join conditions and therefore incurs a certain overhead that could have been avoided with a specialized *join* operator. So, Figure 7.18 mainly shows the improvement that is gained by the reduced input cardinality of the *join*, which is caused by removing subsumed tuples. The improvement is higher for tables with 10% subsumed tuples (than it is for tables with 5% or even 1% subsumed tuples) because more tuples are removed and therefore the input to the *join* is smaller. Although the improvement is not as large as in Experiments 12 and 13, it shows that applying the transformation rule pays off.

Experiment 15 (Join over Data Fusion)
We evaluate transformation Rule 6.46 (see page 99) that allows the exchange of data fusion *and* join. □

Methodology. As an example for the transformation rules considering the exchange of the *data fusion* operator and the *join* operator, we measure the runtime of a set of simple queries where a *data fusion* follows a *join* (query Q_1) and compare their runtime to the transformed queries where the *data fusion* operator has been pushed down below the *join* (query Q_2). We used the CUSTOMER and ORDERS tables from the TCPH-1G data set and varied the selectivity of the *data fusion* operation (50%, 80%, 90%, 95%, and 99%) and the size of the relations (500 up to 5.000 customer tuples and their 5.000 up to 50.000 related orders). We fuse on the ORDERS tables and report the performance gain in Figure 7.19. Runtimes are median values over 10 runs. Performance gain is reported as the runtime of query Q_1 divided by the runtime of query Q_2.
Discussion. Pushing a *data fusion* operator down below a *join* pays off, as can be seen in Figure 7.19. The performance gain for the tables from the TCPH-1G data sets ranges from around 1.02x times for a selectivity of 99% up to around 1.73x for a selectivity of 50%, indicating that the transformed query Q_2 (*data fusion* pushed down) is always faster for the tested combinations of selectivity and size. Figure 7.19 also shows that the performance gain is higher for lower selectivities of the *data fusion* operation, as in these cases fewer tuples are in the output of the *data fusion* and in the input to the *join* operation. Performance gain also decreases with increasing size of the CUSTOMER table. When considering same selectivity of the *data fusion* operation, this is due to the different number of tuples in the input to the *join* in query Q_1 compared to query Q_2.

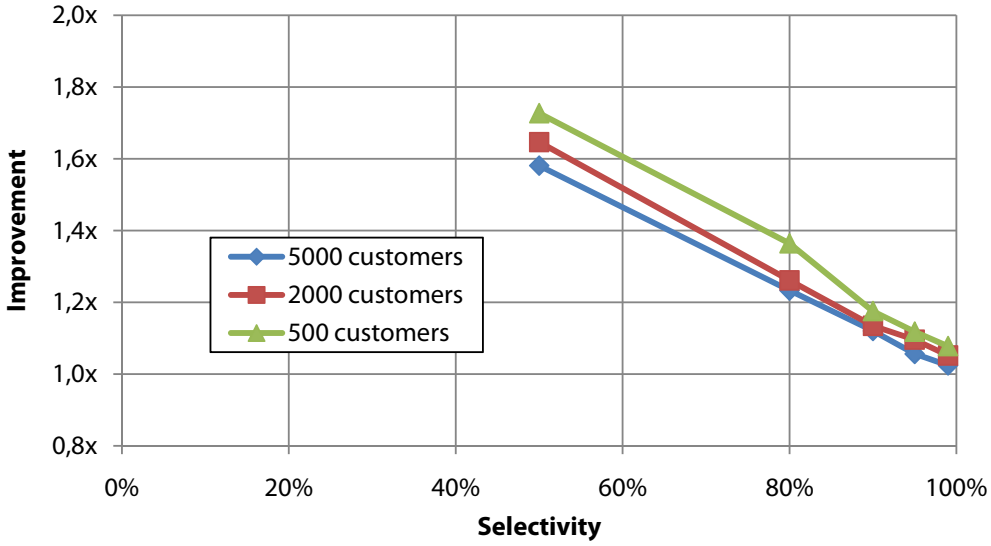


Figure 7.19: Improvement in runtime by pushing *data fusion* below a *join* for examples from the TCPH-1G dataset with 500, 2000 and 5000 customers and their orders, regarding different percentages of duplicated tuples in the table containing the orders (resulting in different selectivities).

Part IV

Surrounding Data Fusion

A ship in harbour is safe, but that is not what ships are built for.

(William Shedd)

8

System Support for Data Fusion

Data fusion techniques have been integrated into two research tools that have been developed within our research group. In the following chapter we take a look at these two systems, describe their data fusion capabilities and how they support the task of data fusion.

We first present the HumMer[1] system (Bilke et al., 2005), which is an information integration system. It implements a three step data integration process (*schema matching*, *duplicate detection*, and *data fusion*) as described in Chapter 2.2 and guides the user – using a wizard-like interface – through the different steps of data integration. A second way of interacting with the HumMer system is via a query language. This query interface has been further enhanced and led to the FuSem[2] system (Bleiholder et al., 2007), a system that offers users a variety of different fusion techniques and lets them explore their differences. The FuSem system serves as testbed and implementation platform for many of the techniques described so far and some of the related work. Compared to standard ETL tools, FuSem does not allow to model and execute a workflow, but is query driven. Thus, it lets the user declaratively define a result and executes the corresponding operators to get the result. However, individual operators that are implemented in the system (such as *subsumption* or *data fusion*) could also be part of an ETL tool. In that case they would have been implemented as step (or transformation) in the ETL tool and thus been used in an ETL process/workflow.

8.1 The HumMer System - Integrating Data

The Humboldt-Merger (HumMer system) is an integrated information system and allows the semi-automatic integration of several remote and heterogeneous data sources (Bilke et al., 2005; Naumann et al., 2006). It has been developed within our research group to combine work from three different fields so that all three steps of data integration (*schema matching*, *duplicate detection*, and *data fusion*) are supported and automated as far as possible. The system is mainly used via a wizard-like interface which requests user input if needed, but uses default settings whenever possible.

It semi-automatically defines a schema mapping between the sources that are to be integrated, which is then used to semi-automatically detect duplicate object representations and finally fuse them into one clean representation. At each step, the user can influence the process. He can alter the proposed mapping, he can additionally classify possible duplicates as real or no duplicates, and he can define how data conflicts are to be resolved when fusing the object representations in the end. The system handles relational data and is able to access a variety of data sources.

The system considers and acknowledges the whole range of conflict types (schematic, identity, and data conflicts). Schematic conflicts are solved using a duplicate-based schema-matching approach (Bilke and Naumann, 2005), whereas identity conflicts are solved using the DogmatiX approach (Weis and Naumann, 2005) applied to relational data. The result of the duplicate detection step consists of duplicate clusters, where all duplicate object representations of a real-world object are assigned to the same cluster. Data fusion

[1] "Humboldt-Merger", see `http://www.hpi.uni-potsdam.de/naumann/hummer/` for a detailed description

[2] "Fusion Semantics" or short for "fuse them", see `http://www.hpi.uni-potsdam.de/naumann/sites/fusem/` for a detailed description and a Java Webstart application

is accomplished by using these duplicate clusters, removing subsumed tuples, and then applying conflict resolution functions to each cluster/group as defined by the user (Bleiholder and Naumann, 2005). The system uses the *data fusion* operator as defined in Section 4.4.

HumMer is the first system to combine independent techniques to handle different kinds of conflicts and thus to support the full data integration process (see Section 2.2 on page 16 for a description of the process) and to combine heterogeneous data from different sources into one clean representation.

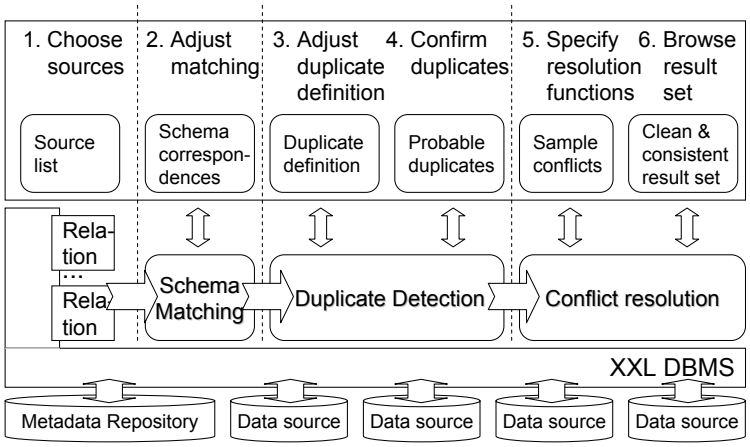


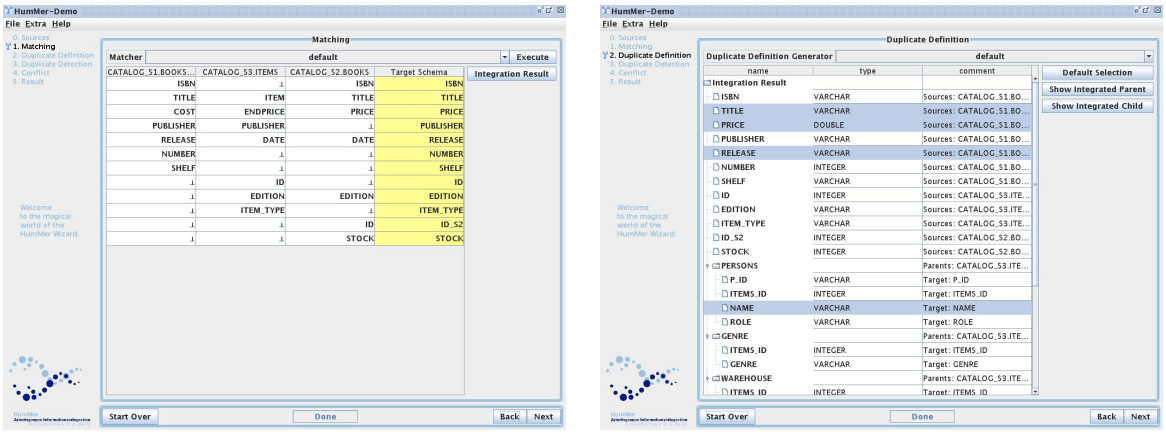
Figure 8.1: The HumMer framework fusing heterogeneous data in a single process.

The Humboldt-Merger is implemented as a stand-alone Java application. The underlying engine of the entire process is the XXL framework, an extensible library for building database management systems (den Bercken et al., 2001). This engine together with some specialized extensions handles tables and performs the necessary table fetches, joins, unions, and groupings. On top of the process lies a graphical user interface that drives the user experience. A metadata repository stores all registered sources of data under an alias. Sources can include tables in a database, flat files, XML files, web services, etc. Because we assume relational data within the system, the metadata repository additionally stores instructions to transform data into its relational form.

HumMer allows integrating data via a wizard, guiding users in a step by step fashion (in Figure 8.1 seen from left to right): Given a set of source aliases as chosen by the user, HumMer first generates the relational form of each and passes them to the schema matching component (Step 1. Choose sources in Figure 8.1). There, columns with same semantics are identified and renamed accordingly, favoring the first source mentioned in the query. The result is visualized by aligning corresponding attributes on the screen (see Figure 8.2(a)). Optionally, users can correct or adjust the matching result (Step 2. Adjust matching). An extra sourceID column is added to each table to store the alias of the data source and then an *outer union* is performed on the set of tables.

The resulting table is the input to duplicate detection. If source tables are part of a larger schema, this component can consult the metadata repository to fetch additional tables and generate child data to support duplicate detection. First, the schema of the merged table, along with other tables that still might reside in the databases is visualized as a tree (see Figure 8.2(b)). Heuristics determine which attributes should be used for duplicate detection. Users can optionally adjust the results of the heuristics by hand within the schema (Step 3. Adjust duplicate definition). The duplicate detection component adds yet another column to the input table – an objectID column designating sets of tuples that represent the same real-world object. The results of duplicate detection are visualized in three segments: Sure duplicates, sure non-duplicates, and unsure cases, all of which users can decide upon individually or in summary (Step 4. Confirm duplicates) (see Figure 8.2(c)).

The final table is then input to the conflict resolution phase, where tuples with same objectID are fused into a single tuple and conflicts among them are resolved. Figures 8.3, 8.4, and 8.5 show three screenshots of the system from this last phase. Figure 8.3 shows the input to data fusion, namely duplicate clusters as produced by duplicate detection. The additionally introduced columns objectID and sourceID can be seen on the left. The conflict resolution functions to be used are given above the column header. Here, e.g., the min function is used to resolve conflicts in the Price column. Figure 8.4 shows available conflict resolution functions that can



(a) Schema Matching

(b) Duplicate Definition

(c) Duplicate Detection

Figure 8.2: Visualization of the first three steps of the HumMer wizard. We see (a) a schema matching that can be altered, (b) chosen attributes for duplicate detection, and (c) decision screen for unsure duplicates.

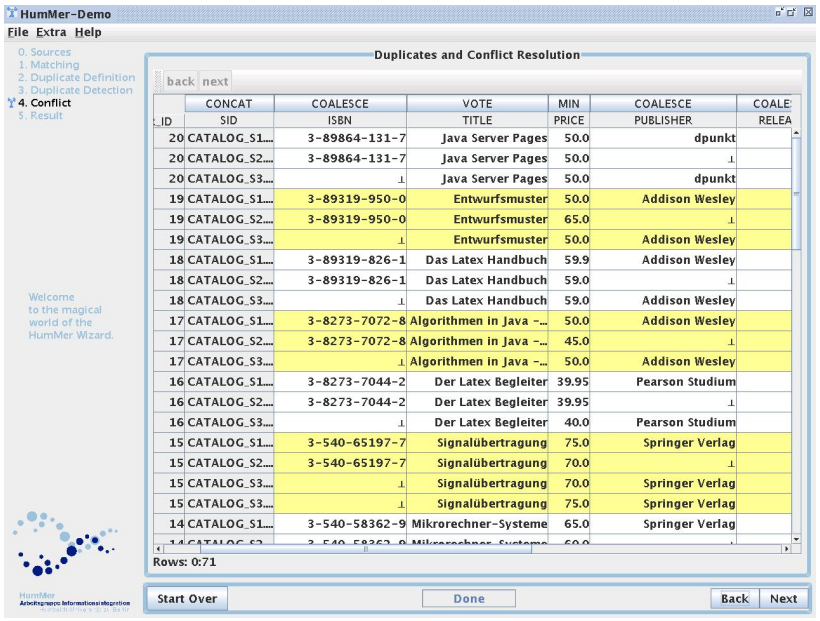


Figure 8.3: Visualization of steps of the HumMer wizard from the data fusion phase: duplicate clusters after schema matching and duplicate detection. The conflict resolution functions is given above the column header.

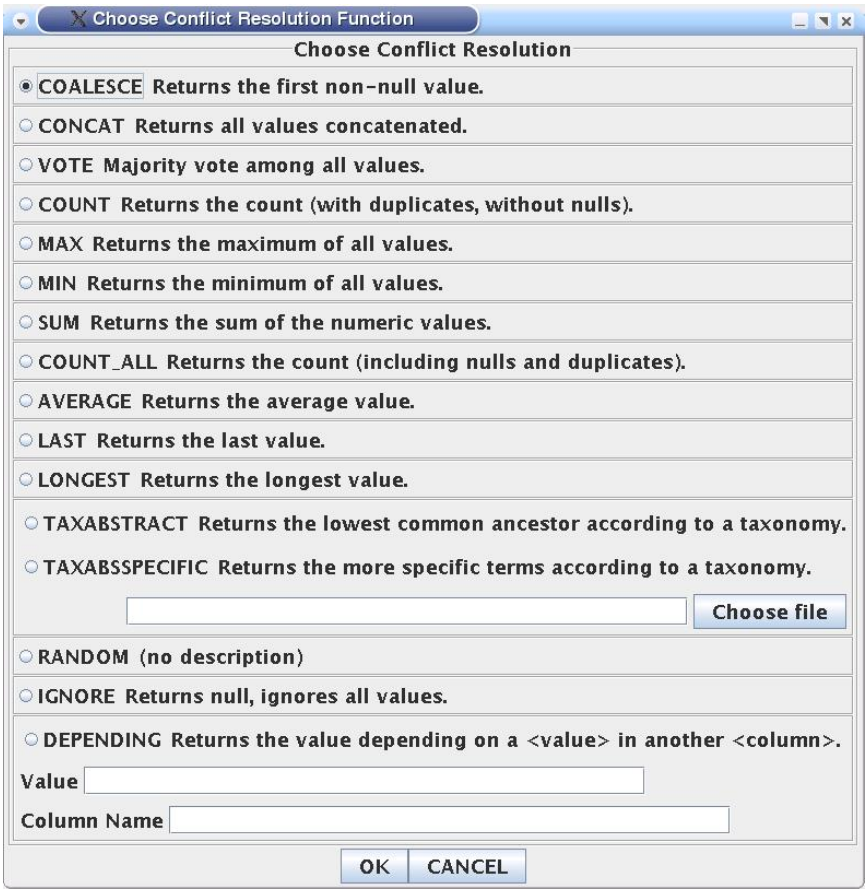


Figure 8.4: Visualization of steps of the HumMer wizard from the data fusion phase: available conflict resolution functions, a subset of possible functions as proposed e.g., in (Naumann and Häussler, 2002; Bleiholder and Naumann, 2005) or in Section 3.2.2.

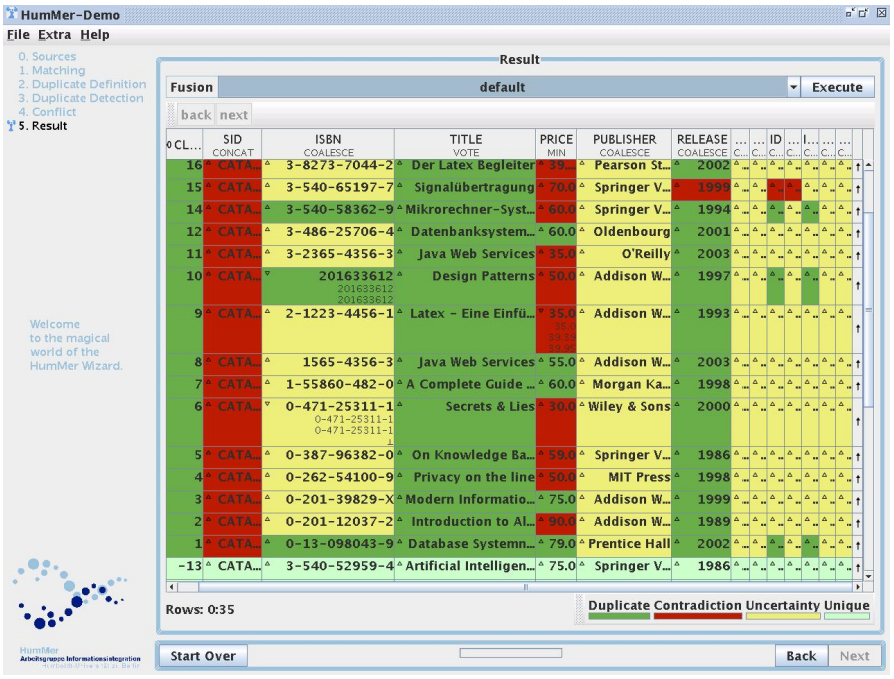


Figure 8.5: Visualization of steps of the HumMer wizard from the data fusion phase: a result with conflicts resolved.

be chosen for each column (Step 5. Specify resolution functions). The list is a subset of the functions given in Section 3.2.2 and documents the current functionality of the HumMer system. Additional functions can easily be added to the system. Finally, Figure 8.5 shows the result of data fusion. Each object is represented only once in the result with the resolved value according to the conflict resolution function used in each column. The result value is color coded to show where uncertainties and conflicts have been resolved and where values have been equal in the sources. The system also accounts for data lineage and shows the original values (for each resolved value) upon request. The final result is passed to the user to browse or use for further processing (Step 6. Browse result set). Also, data values can be color-coded to represent their individual lineage (one color per source relation, mixed colors for merged values, not shown here).

8.2 The FuSem System - Comparing Fusion Results

FuSem – short for *Fusion Semantics* or just *fuse them* – is a Java application based on parts and an enhancement of the HumMer system (Bleiholder et al., 2007). It allows to access different data sources and run Sql-like queries. Complementing the original wizard interface of the HumMer system, users of FuSem can specify results completely via a query language. It assumes that schema matching and duplicate detection has been performed beforehand. Queries to the system are operator-based, result in the execution of an operator tree and thus can be optimized, whenever possible. The system incorporates the semantics of several related data fusion techniques (e.g., *data fusion*, *merge*, *match join*, consistent answers, etc.) and understands the proposed syntax of each. Because these semantics have different properties and outcomes, FuSem also allows the user to visually compare different results of fusion operations. This is schematically shown in Figure 8.6: Different fusion techniques can be applied to possibly conflicting data from different data sources. Then, their results can be compared.

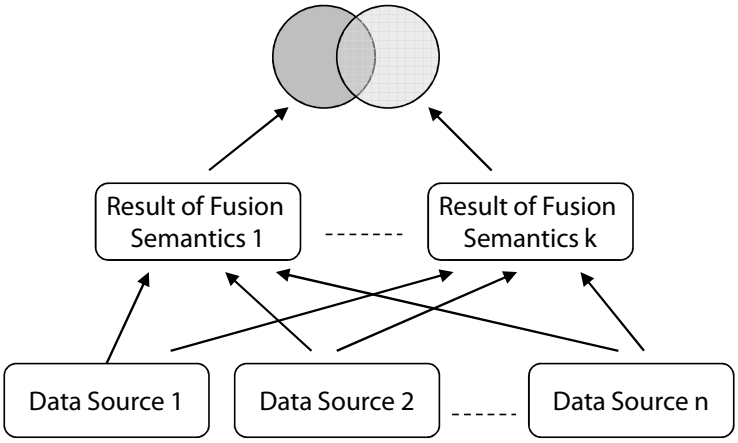


Figure 8.6: Schematic overview of the functionality of the FuSem system: Different fusion semantics and/or fusion techniques are applied on source data and their results are then visually compared.

Together with an interactive query interface, fast query execution, and an intuitive presentation of the results, the system allows easy exploration of fusion results, the underlying data sources, and applying and testing different data fusion operations to finally come up with the most suitable fusion result for the task at hand. Using the XXL framework (den Bercken et al., 2001), the system's architecture allows easy addition of new semantics for data fusion.

Query Processing FuSem is able to handle inconsistent information from multiple heterogeneous sources by using five different approaches:

- The sources can be queried using standard Sql. Thus, it is possible to "fuse" data by applying an (*outer-*) *union* or an (*outer-*) *join*. More complex Sql statements are also possible, offering some, but generally

only limited data fusion capabilities. The *minimum union* and *complement union* are also part of FuSem's functionality, as well as *subsumption* and *complementation*.

- The system can perform the *merge* (⊠) operator (see (Greco et al., 2001) or Section 9.1.2 for more details), combining two data sources by a series of *outer join* and *union* operations, creating a result with uncertainties removed, but that still may contain contradictions.

- FuSem also implements the *match join* operator (see (Yan and Özsu, 1999) or Section 9.1.1 for more details), which uses an *outer-join* to combine all possible attribute values for an object and chooses among them given a confidence parameter, possibly still not resolving all contradictions at hand.

- FuSem also supports the consistent query answering approach in form of the rewritings given by the ConQuer system[3] (see (Fuxman et al., 2005a) or Section 9.2.2 for more details). This approach allows only non-contradictory tuples to enter the final result.

- Finally, FuSem offers to resolve inconsistencies by use of the *data fusion* operator presented in Section 4.4 and specified by means of the Fuse By statement presented as an Sql extension.

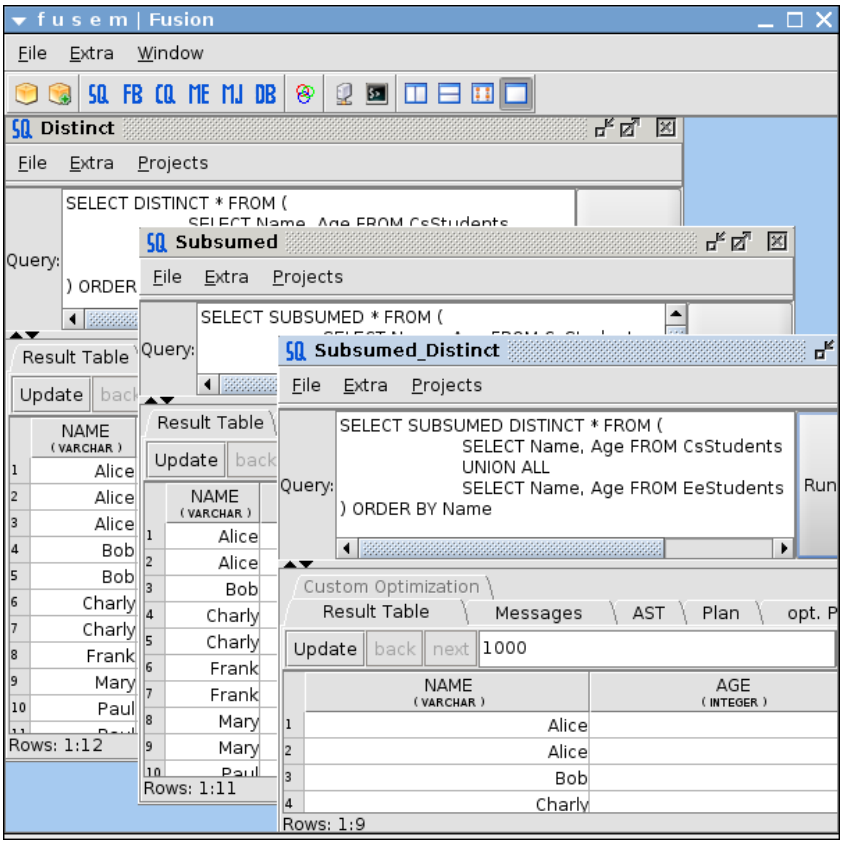


Figure 8.7: Different views of FuSem's GUI: specifying *subsumption* and *minimum union*.

Different data fusion techniques can be combined by using the output table of one as input table for the other technique. Figure 8.7, 8.8, and 8.9 show different views of the user interface of FuSem, where queries are issued to the sources under different fusion semantics – one window for each query, using one of the possible semantics. Thereafter, the interface can graphically display the query execution plan and the query result table itself. Query plans are optimized whenever possible using a mixture of query rewritings and physical optimization. One simple option to compare results of different approaches is to align the corresponding result tables next to each other. More sophisticated comparison methods are discussed next.

[3] We thank Ariel Fuxman for kindly providing the source code.

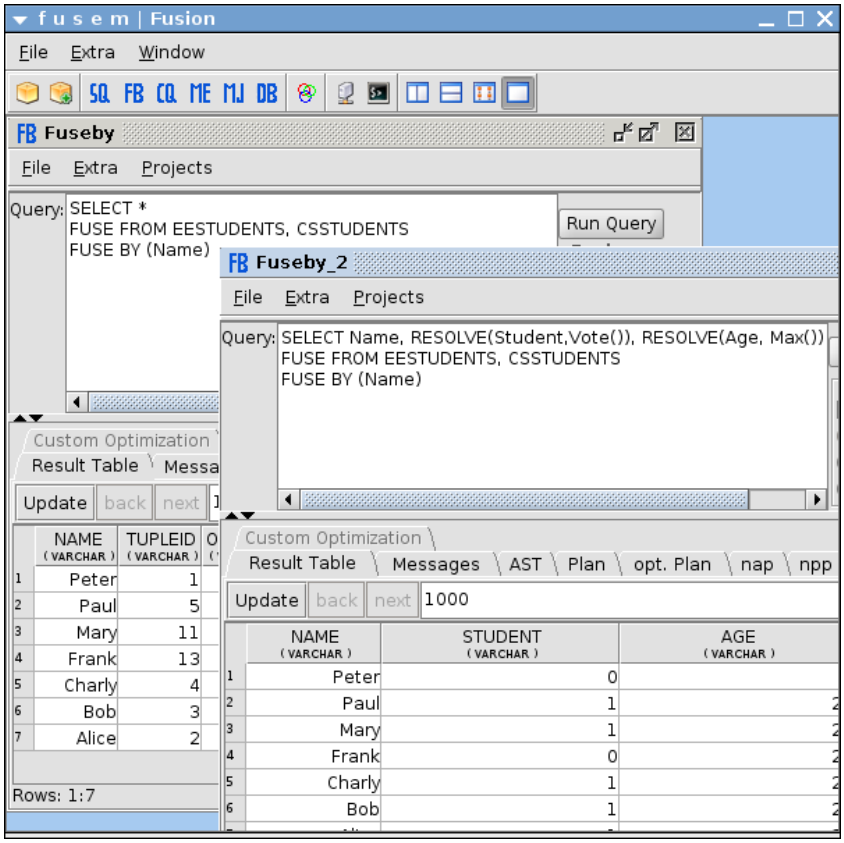


Figure 8.8: Different views of FuSem's GUI: fusing data by Fuse By queries.

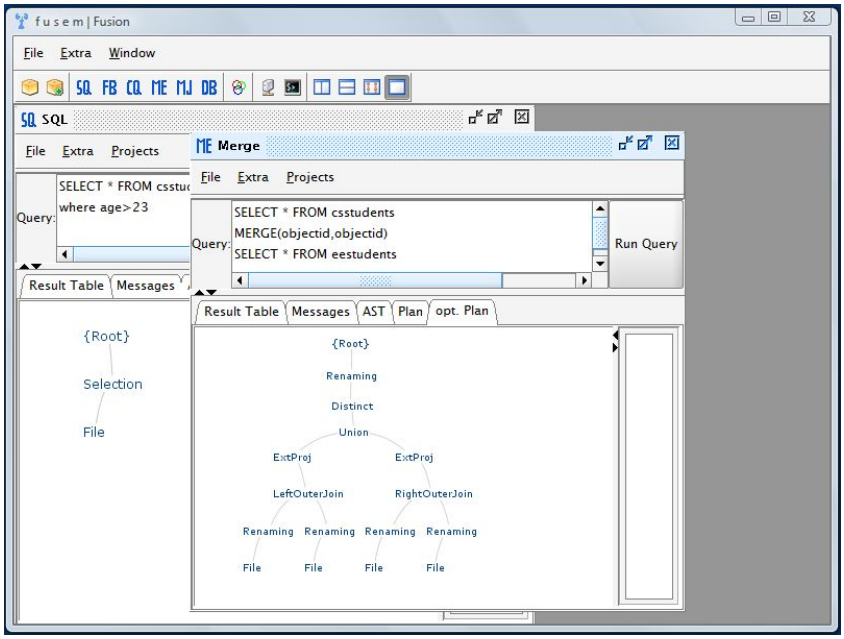


Figure 8.9: Different views of FuSem's GUI: showing operator plans.

Exploring Differences Just as in the HumMer system, in the FuSem system data fusion results are presented as visually enhanced tables. They graphically show contradictions and uncertainties in the original data and visualize how the different semantics deal with those contradictions. Once the user has created several fused results using different techniques, the system is able to compare these. FuSem provides some numerical information when comparing fused results and in particular graphical access to the fused results. To quickly

test several fusion techniques, the system generates a sample of the data for visualization.

We employ Venn-diagrams to show the overlap of object descriptions, similarities, and contradictions in the representations of fused results. To illustrate, regard the example in Table 8.1. For four objects o_1 – o_4 we show the two attributes DESCRIPTION and SIZE as they would appear under the respective semantics. In order to compare object descriptions, a column holding an object identifier is needed.

Object	Semantics	DESCRIPTION	SIZE
o_1	Merge	sugar bowl	large
	Fuse By	Zuckerdose	large
	Match Join	sugrabowl	small
o_2	Merge	teapot	large
	Fuse By	teapot	large
	Match Join	teapot	large
o_3	Merge	teacup	large
	Fuse By	teacup	large
o_4	Merge	saucr	middle
	Fuse By	saucer	middle
	Match Join	saucer	middle
…	…	…	…
	…	…	…

Table 8.1: Example data showing object representations as produced using different fusion semantics.

Figure 8.10, 8.11, and 8.12 show three different visualizations of the data of Table 8.1 as presented by FUSEM. Each visualizes a different aspect of data fusion; each is a more detailed view of the previous.

Visualizing object overlap: In Figure 8.10 we show the Venn-diagram of three regions, each corresponding to one of the three fusion semantics of the example – *merge*, FUSE BY, and *match join*. Each region contains the set of result tuples. The number of objects represented by them is shown in the corresponding region in the diagram. If one set contains object representations that also appear in another set, it is represented by overlapping regions. For instance, objects o_1, o_2, and o_4 are located in the highlighted sub-region towards the center of the diagram. All three objects are in the results of all three semantics. Thus they are placed in the overlap of all three regions.

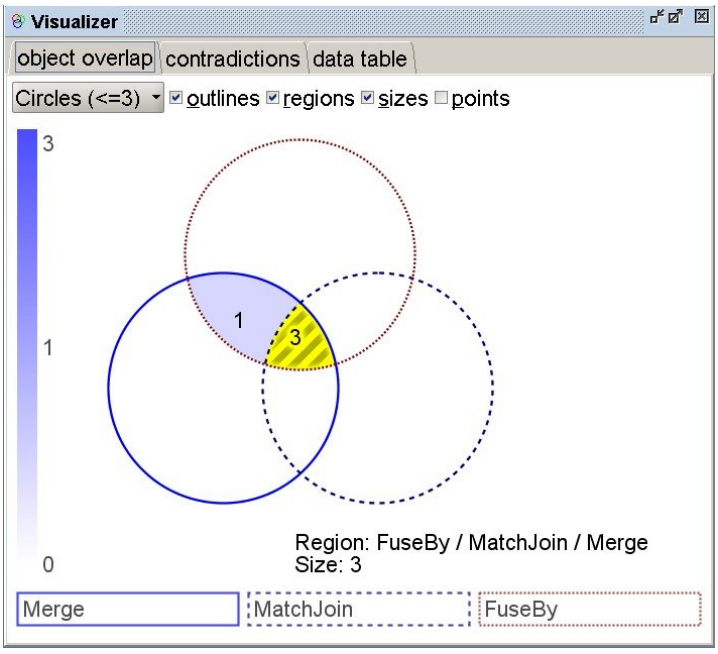


Figure 8.10: Visualization of the example fusion results from Table 8.1: object overlap in fusion results.

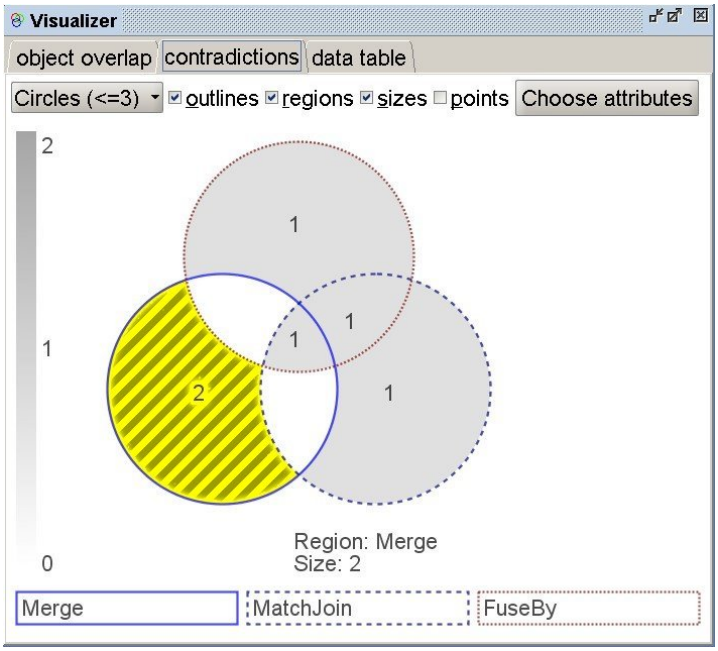
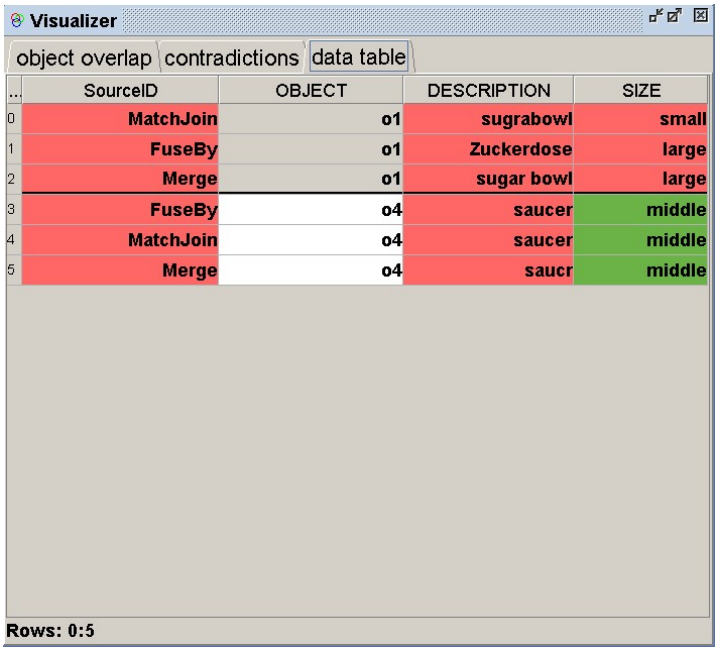


Figure 8.11: Visualization of the example fusion results from Table 8.1: contradictions.



...	SourceID	OBJECT	DESCRIPTION	SIZE
0	MatchJoin	o1	sugrabowl	small
1	FuseBy	o1	Zuckerdose	large
2	Merge	o1	sugar bowl	large
3	FuseBy	o4	saucer	middle
4	MatchJoin	o4	saucer	middle
5	Merge	o4	saucr	middle

Figure 8.12: Visualization of the example fusion results from Table 8.1: contradicting tuples.

By clicking on one of the sub-regions (e.g., the one in the middle, highlighted in Figure 8.10) a second view details the actual objects of that regions, and in particular, their tuple overlap.

Visualizing tuple overlap: This second view (Figure 8.11) visualizes the degree of data conflicts among object representations. Again, each region in the Venn-diagram represents a fusion semantics. If two object representations from different semantics are equal (same Description and Size values) they appear in the corresponding overlap of their regions. For instance, object o_2 has the same Description and Size values in all three fusion semantics, which puts it into the overlapping sub-region of all three regions. Again, we show only the number of objects that are contained in a certain sub-region. Object o_1 on the other hand differs in all three semantics, which is the reason why it is placed into the three outer regions, which have no overlap.

Which attributes are considered in determining data conflicts can be specified by the user. This ability allows

rapid exploration of the degree of conflict in individual or certain combinations of attributes. The specific conflicting tuples with their data values can be explored by again selecting a certain sub-region.

Visualizing data conflicts: In the table view of Figure 8.12 we present all representations of the set of objects in the sub-region chosen in the previous view (e.g. the lower left one, highlighted in Figure 8.11). Thus, users can assess the severity of the actual data conflicts, and if needed change the original queries and finally select the desired fusion semantics. In this view, values are color-coded, indicating conflicts by red color.

Exploring source conflicts: The visualization interface of FuSem can also be used to visualize the overlap and inconsistencies among different data sources: Instead of applying different fusion semantics and comparing them, we assume each data source to represent a "semantics" and compare those. Note that we must still assume an object identifier to identify objects.

Objects and colors: FuSem has the ability to display objects as points in the respective sub-regions. Figure 8.13 shows an example for six different fusion results using square-style Venn-diagrams. That way agreement or disagreement between different fusion semantics can easily be spotted by point clusters in the sub-regions. Points are randomly distributed in the respective sub-region of the digram. As was already shown in previous figures, numbers of objects are not only represented as integers or as sets of points, but are also represented by color-coding sub-regions (see Figure 8.14).

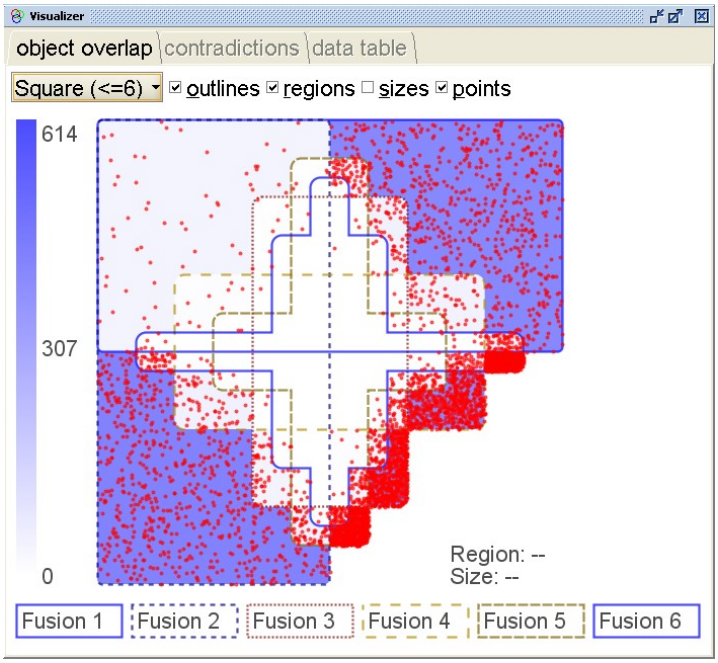


Figure 8.13: Example visualizations: square-style Venn-diagrams.

Venn-diagram types: The displayed technique of showing conflicting data is easily extended to higher numbers of different semantics (or data sources). Unfortunately, Venn-diagrams for more than five or six sets are difficult to understand for the average user. However, in most relevant scenarios, the number of results to compare and therefore the number of sets to visualize will be less than 6. FuSem implements the three ways of drawing Venn-diagrams as presented in (Glassner, 2003). Figure 8.15 shows an Edwards-style Venn-diagram used for a visualization of six different fusion results. The visualization by area-proportional Venn-diagrams (Chow and Ruskey, 2004) is possible only for two result sets.

Statistics: Some basic statistics are presented to the user when comparing fused results:

- Basic statistics such as the number of objects, the number of tuples representing these objects and the number of distinct representations of these objects.
- The degree of completeness and conciseness as defined in Section 2.3 are also computed per sub-region of the diagram and shown to the user.

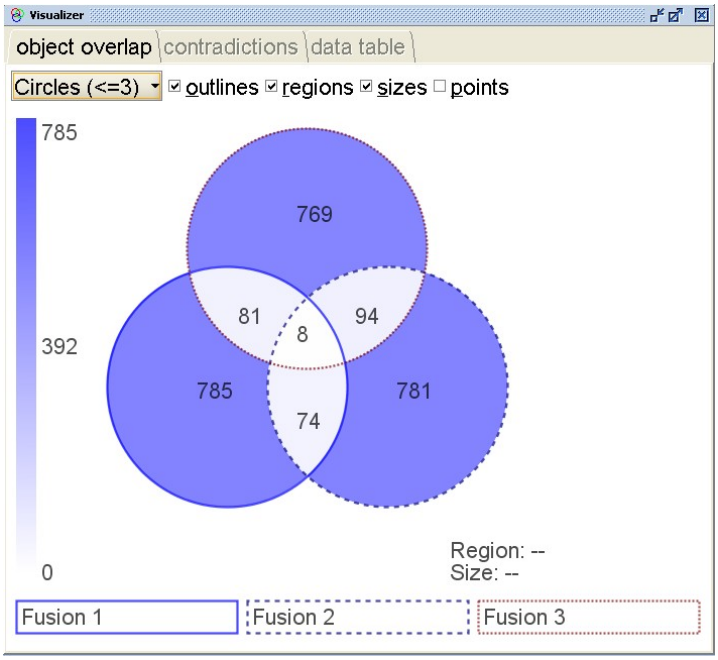


Figure 8.14: Example visualizations: circle-style Venn-diagrams.

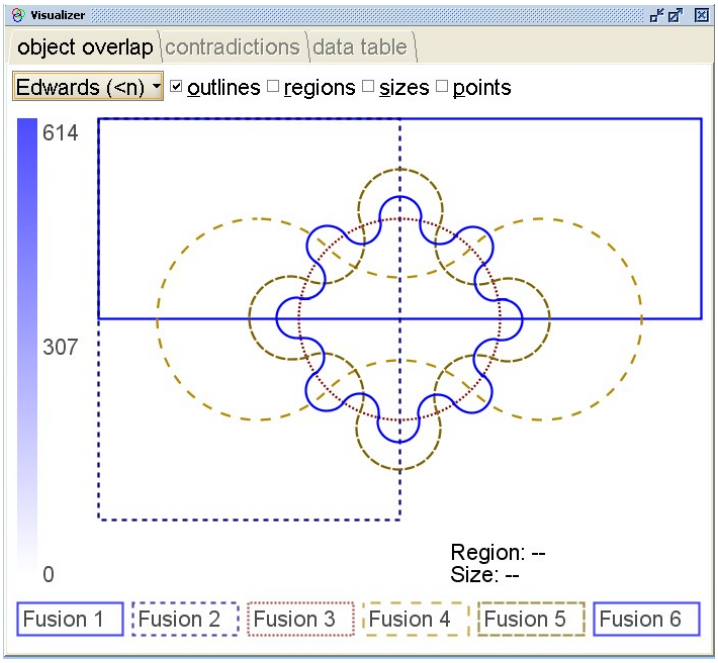


Figure 8.15: Example visualizations: Edwards-style Venn-diagrams.

- Agreement among object representations (coherence). We compute and present per sub-region the percentage of object representations that do not differ. This is also broken down to the column level (column coherence).

Figure 8.16 gives an example of the statistics generated by the FuSem system and presented to the user for an example of two object representations represented by seven tuples. Statistics are presented to the user in form of a statistical summary and can be computed for different attributes or combinations of attributes thus emphasizing in which attributes the semantics differ most, or what percentage of objects are fused the same way using different semantics. The user only sees the parts of the statistics that coincide with the actual view (the highlighted sub-region). For instance, a high coherence among different fusion results points to

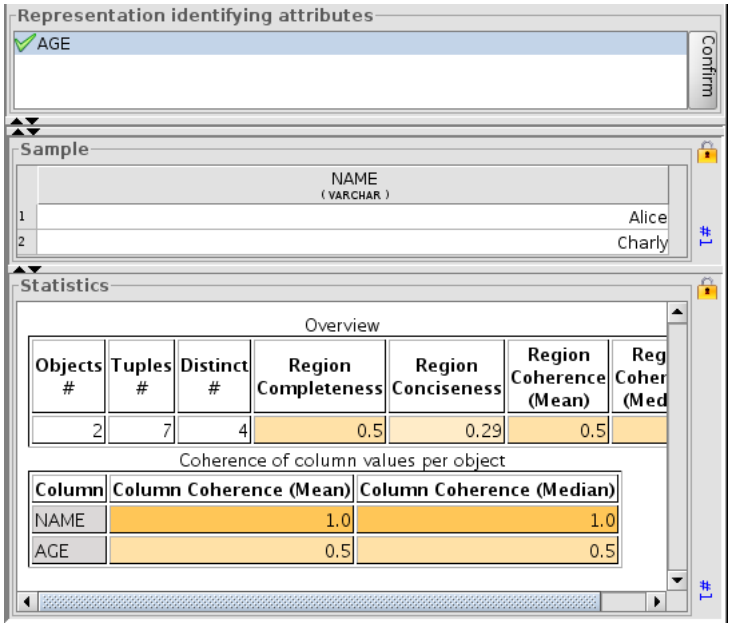


Figure 8.16: Example visualizations: Statistics computed for one particular region.

a high similarity among the different semantics, inviting the user to purposefully explore only the object descriptions where the fusion semantics differ in their results, or finally choose the fusion technique which is computationally cheaper while still creating the same (desired) result.

The FuSem system can also be used to compare different automatic fusion techniques against a result that has been manually produced by a domain expert. That way specific operators or techniques can be tested whether they produce the desired output. FuSem allows for fast and easy parameterization of the operators in order to match the desired result by the expert. If the result is given for only part of the data, the tool can then be used to automatically fuse all of the data.

You must learn from the mistakes of others. You can't possibly live long enough to make them all yourself.

(Sam Levenson)

9 Related Work

The following chapter presents and discusses related work in the field of data fusion in more detail. Although some techniques or systems have already been mentioned and referred to throughout the previous chapters, a description in a broader context has been omitted so far. Related work divides itself into two sections: techniques – mostly of relational nature – and systems. Section 9.1 presents an overview on data fusion techniques that mostly have been considered within the relational model. In Section 9.2 an overview of a series of integration systems is given that employ data fusion strategies and or techniques in some form. This chapter summarizes findings published in (Bleiholder and Naumann, 2008). We refer the reader to the survey on data fusion for yet further details on the techniques and systems.

9.1 Data Fusion Techniques

This section examines different relational operators and other techniques that can be used to combine and fuse data from multiple data sources into one single representation. We describe and compare the techniques in terms of their data fusion capabilities, their ability to create complete and concise results, and their differences in results produced. Example statements, syntax, and – if applicable – SQL rewritings of the respective techniques can be found in (Bleiholder and Naumann, 2008).

9.1.1 Join Approaches

Join approaches in general increase intensional completeness, as attributes from different sources are separately included into the result. However, this comes at the cost of not guaranteeing extensionally completeness except for the case of a *full outer join*. Concerning extensional conciseness, join approaches perform very well, but only if we can rely on globally unique identifiers (e.g., created by duplicate detection techniques), and if no intra-source duplicates are present.

Standard Joins An *equi-join* ($\bowtie_=$) combines tuples from two relations if the join condition consists of equality conditions between columns and if it evaluates to true for two tuples (Ullman et al., 2001). The result also includes all other attributes from the sources in the result. In an integration scenario, where the goal is to relate tuples which describe same objects, using an *equi-join* on some attributes forming a real-world ID, e.g., using R.NAME=S.NAME, seems the natural choice. We refer to this as *key join* ($\bowtie_{id=id}$). However, in general, when combining two tables by a *join*, other conditions are possible, as are the use of attributes that do not form a real-world ID.

The *natural join* ($\bowtie$) combines two tuples from two relations if all common attributes (same name) coincide in their attribute values (Ullman et al., 2001). Attributes that are not present in all relations are not considered in joining the relations but are included in the result.

The *full natural outer join* ($\bowtie\!\!|$) extends the result of a *standard natural join* operation by adding tuples that are only included in one of the two source relations. The attributes present in only one relation are padded by NULL values for the non-joining tuples of the other relation (Ullman et al., 2001). There exists an *outer join*

variant that is based on the *key join* ($|\bowtie|_{id=id}$) instead of the *natural join*, as well as *left* ($|\bowtie_{id=id}$) and *right outer join* ($\bowtie|_{id=id}$) variants.

Joins can be used for data fusion, but they are not perfect. There are two main drawbacks: First, data conflicts, if present in the sources, are generally ignored by *joins*, as objects are either not included in the results (*standard join* and data conflicts in join attributes), or result in two different result tuples (*outer join* and data conflicts in join attributes). Second, using a *join* alone does not allow for inclusion of information from schema matching, semantically equivalent columns may still form two columns in the result (see the next section on how to handle these cases).

Increasing Conciseness in Join Results The schema of regular join results normally does not contain only one attribute for all groups of semantically equivalent attributes. Therefore these operators are complete, but not concise, and all existing, possibly contradictory, data values from the sources still exist in the result. One way of resolving these conflicts and increasing intensional conciseness is by combining these attributes into one with an additional operation.

In combination with *join* operators, the SQL function COALESCE proves to be useful: `coalesce(`v_1`, `v_2`, ..., `v_n`)` gives back the first value v_i, that is not a NULL value. If all v_i are NULL, a NULL is returned. When the function is used in combination with a *join*, the input to COALESCE are usually values from semantically equivalent attributes from different sources, e.g., `coalesce(Police.Birthdate, Hospital.DateOfBirth)`. That way, COALESCE removes uncertainties (if only one date is given and the other is a NULL value) or could be used (see Section 9.1.2) to prioritize input tables (here, if both sources give an age value, prefer e.g., POLICE.BIRTHDATE). COALESCE therefore serves in implementing strategies such as TAKE THE INFORMATION or TRUST YOUR FRIENDS (see Section 3.2.1 for more information on strategies).

In (Lim et al., 1995) a variant of the *full disjunction* operator (see the next section for details on *full disjunction*) is used, which merges the key columns into one column, using COALESCE. This is possible as the key columns either contain the same or a NULL value, resulting from the assumption that no intra-source duplicates exist. They additionally define a *generalized attribute derivation* operation (GAD), which uses functions to combine values from several columns of the output of the *full disjunction* into one single value. Using the GAD operation it is possible to combine semantically equivalent attributes into a single attribute and resolve conflicts. GAD translates to the use of user defined functions (UDF) in SQL, an example statement can be found in (Bleiholder and Naumann, 2008). Thus, conciseness is increased and conflicts are resolved, depending on the functions used. However, intra source duplicates or a poor join order (if using *join* instead of *full disjunction*) leave duplicate tuples in the result.

Full Disjunction As the *outer join* operator in general is not associative, combining more than two tables by an *outer join* may result in different result tables, depending on the order, in which the tables are joined. The *full disjunction* operator is defined by (Galindo-Legaria, 1994) as the combination of two or more tables, where all matching tuples (according to some join attributes) are combined into one single tuple. It could therefore be seen as a generalization of a *full outer join* to more than two relations. *Full disjunction* can generally not be computed by a combination of *outer joins* alone (Galindo-Legaria, 1994). (Rajaraman and Ullman, 1996) cover *full disjunctions* in more detail and recent work on the topic provides fast implementations (Kanza and Sagiv, 2003; Cohen and Sagiv, 2005; Cohen et al., 2006). Generally, *full disjunctions* are beneficial in situations with more than two tables where a *full outer join* would have been used with only two tables. Another characteristic is that the schema of the result of *full disjunction* combines semantically equivalent columns from different source tables as is usually not the case with *joins*.

Match Join and Conflict Tolerant Queries The *match join* has been defined together with a conflict tolerant query model in the context of the AURORA project (Yan and Özsu, 1999) and requires a designated column with a real-world ID. In a first step, corresponding attribute values from all sources are separately projected out and combined by *union*, expanded by their corresponding real-world ID. Then, the real-world IDs are used to rejoin the parts, resulting in one single large table. The operation to create this table is called *match join* and can easily be rewritten in SQL (see (Bleiholder and Naumann, 2008) for an example). In a

second step, to increase conciseness, tuples are selected from this table according to the conflict tolerant query model (Yan and Özsu, 1999). The whole operation can be seen as first creating a set of all possible results (*match join*) and then selecting from it, as specified by the user, creating a conflict free result. This means that in the end there remains only one representation per real-world ID. Conflicts are resolved by applying resolution functions, such as SUM, MAX, MIN, ANY, among others. In contrast to other techniques considered so far, execution complexity becomes important, as the worst case complexity is exponential in the unique values per attribute. In the worst case all possible combinations of attribute values are regarded. Uncertainties could be removed, contradictions are resolved. The selection of tuples in the second step is realized by fine grained operations extending the relational model, which prevents a rewriting of the operation as a whole using standard SQL (only the *match join* part can be rewritten). Please note, that, because the operator looks at all possible combinations of attribute values in the intermediate step, in some cases, the result may contain tuples with attribute value combinations that as a whole have not been present in either of the sources. That way, the operator combines existing facts in a new way to create new facts.

9.1.2 Union Approaches

Union approaches are the natural way of accomplishing extensional completeness, as the result naturally includes all tuples from both relations. In contrast to the join approaches, union approaches do not perform as well concerning conciseness.

Union, Outer Union, Minimum Union, and Complement Union The *union* operator ($\cup$) with set semantics combines the tuples of two union-compatible relations and removes exact duplicates, i.e., tuples that coincide in all attribute values (Ullman et al., 2001). *Outer union* ($\uplus$) overcomes one restriction of *union* and combines two non-union-compatible relations by first adding the attributes that are not in common, padding them with NULL values, and then performing a regular *union* (Galindo-Legaria, 1994). *Outer union* is not a standard operator but can easily be rewritten in SQL (see e.g., (Bleiholder and Naumann, 2008)). The *minimum union* operator ($\oplus$) is defined by (Galindo-Legaria, 1994) as the result of an *outer union*, from which subsumed tuples have been removed (see Section 4).

There is some related work on *minimum union* and *subsumption* as one of its building blocks: *minimum union* is for example exploited in query optimization for *outer join* queries (Galindo-Legaria and Rosenthal, 1997). *Minimum union* is also used in Clio (Hernández et al., 2002) as one possible semantics to use a schema mapping in order to create transformation rules (Popa et al., 2002). *Subsumption* is used in (Rao et al., 2004) to create standardized *outer join* expressions, that way enabling *outer join* query optimization. (Rao et al., 2004) proposes a rewriting for *subsumption* in SQL, using the data warehouse extensions provided by SQL. Another rewriting for *subsumption* is used in (Larson and Zhou, 2005). However, it is also not applicable in general.

The problem of tuple subsumption can be transformed into the problem of minimizing tableau queries (tableau queries as introduced in (Aho et al., 1979)). There exist algorithms for minimizing tableau queries (see e.g., (Sagiv, 1983)), however, they only consider the general case, and are not able to take advantage of the characteristics of the transformed tableau query.

To the best of our knowledge, this work is the first that considers data fusion with the semantics of *complement union* (see Section 4). However, related concepts have been previously explored, i.e., finding and computing the complements of all complementing tuples in a relation is equivalent to finding all maximal cliques in a graph. Standard algorithm for finding all maximal cliques are described in (Bron and Kerbosch, 1973; Stix, 2004; Johnson et al., 1988). See also Section 5.1.6 and 5.2.4 for a more detailed discussion of related work for *subsumption/complementation* and *minimum/complement union*.

Increasing Conciseness in Union Results As semantically equivalent attributes already have been matched, different representations reside in different rows of a union result. Therefore, the obvious way of removing possible data conflicts is by grouping representations of one single real-world object together and aggregating the values of the attributes. In general, the result after grouping is similar to the union result, still extensionally complete, but also does not require any additional schema modification as is the case for join results.

Increasing conciseness in this way requires one or more attributes that serve as identifier to identify same real-world objects. The aggregation part of this operation then takes all tuples of one group and merges them into one resulting tuple. This is done by applying aggregation functions to the attribute values. The standard aggregation functions (MAX, MIN, SUM, COUNT, and AVERAGE) of the SQL standard allow only for limited functionality in resolving conflicts. Therefore, the technique used by FRAQL (Sattler et al., 2000; Schallehn and Sattler, 2003; Schallehn et al., 2004) resolves uncertainties and contradictions by applying non-standard user defined aggregation functions to the attribute values in each group. They define four additional 2-parameter aggregation functions, each of which aggregates one column depending on the values of another column. In general, arbitrarily complex, user-defined function could be used here, in order to combine the values, remove uncertainties and resolve conflicts. Unfortunately, such user-defined aggregation functions are supported by database management systems only in a very limited way, if at all. The AXL framework (Wang and Zaniolo, 2000a,b) supports users with user defined aggregation based on SQL in standard object-relational DBMSs for use in data integration. User defined aggregation is also similar to the approach taken by the data cleansing system AJAX (see Section 9.2.3 and (Galhardas et al., 2000a) for more details on AJAX).

Merge and Prioritized Merge Based on the *match join* operator already discussed earlier, (Greco et al., 2001) define the *merge* operator ($\boxtimes$). This operator can be described as the *union* of two *outer joins* and can be rewritten in SQL. The *merge* operator requires a join condition to relate same real-world objects and uses the COALESCE function to remove uncertainties. Unfortunately, removing uncertainties does not succede if tuples representing the same real-world object and with uncertainties present are in the same relation, although an additional benefit from this behavior is the removal of inter-source subsumed tuples. However, contradictions are not resolved with the *merge* operator. There also exists a variant – *prioritized merge* ($\triangleleft$) – that is used to prioritize among the sources and prefer values from one source. *Merge* is well suited in scenarios where many NULL values could be complemented across sources, e.g., when data sources are sparsely filled or only slightly overlap in their schemata.

9.1.3 Other Techniques

The following techniques are neither join-, nor union-based, many of them incorporate additional information, extend the relational model or existing relational operators, or combine operators in order to fuse data.

Considering all Possibilities A way of dealing with uncertain data is to explicitly represent uncertainty in relations and use it to answer queries. The approaches described in (Lim et al., 1994) and (DeMichiel, 1989) add an additional column to each table, its value helping to decide whether a tuple should be included in the query answer or not. Whereas (DeMichiel, 1989) marks a tuple with a discrete value (the value is one of *yes*, *no*, or *maybe*), the approach in (Lim et al., 1994) uses probabilities. The operators of the relational algebra are extended to cope with this additional data and use it. In the end, all tuples are presented to the user, together with the additional information of it belonging to the result or not. The two main problems with these approaches are that uncertainty is modeled at tuple level only and that it is generally difficult to determine the probabilities and where to take them from.

In (Lim et al., 1997) relations are extended by an additional attribute holding information on the origin (the data source) of the particular tuple. The relational operators are extended to use this information and therefore this approach is an implementation of a TRUST YOUR FRIEND strategy (see Section 3.2.1).

Another way of representing multiple values and their attached probabilities is in allowing multiple-valued attribute types in a relation (Tseng et al., 1993). In (Tsai and Chen, 2000), the authors present the *partial natural outer join*, which extends the *full disjunction* operation by allowing multiple values in an attribute together with their probability of being the correct value. Such a result is as complete as a regular *full disjunction*, but more concise. However, it is not as concise as the result in (Lim et al., 1995), because conflicts are not resolved by a GAD operation. Instead, all values are preserved, with an attached probability of being the correct value. Probabilities are independent and the information of formerly connected or unconnected values (e.g., originating in the same or in different tuples) is not preserved.

In an OLAP scenario (Burdick et al., 2005) use probability distribution functions to model both uncertainty of the value, as well as imprecision according to a dimension hierarchy. However, in contrast to the other approaches including probabilities in the computation of results, in the end, only one value is presented to the user. The value given to the user is the one with the highest probability of all the possible values. However, the approach does not allow any user interaction beyond giving the probabilities, it especially does not allow to influence the conflict resolution.

All these approaches implement the Consider all possibilities strategy, using additional information to allow the user to choose consciously among all possibilities or presenting the most probable value.

Considering only Consistent Possibilities A new line of research was initiated by work started in (Arenas et al., 1999), which introduced the idea of returning only consistent information (not containing conflicts, given some constraints) from a database to a user when issuing a query.

Consistent query answering has developed into a large subfield of its own and is therefore in the following only briefly addressed. The notion of a consistent answer is based on the idea of a repair, a new, consistent, database created by only inserting or deleting tuples into/from an inconsistent database. Inconsistency is given by constraints on the data (e.g., functional or inclusion dependencies) that are not fulfilled. In the relational world, a *consistent* answer to a query is the set of tuples that are contained in the answer to the query in *all* possible repairs. (In contrast to that, a *possible* answer would be a set of tuples contained in *any* possible repair.)

The overall goal in this line of research is to answer a query by giving a *consistent* answer. This generally requires the computation of all possible repairs and therefore has high computational costs. In contrast, most other techniques mentioned so far are fast and scalable, as only one specific solution is computed, or the requirements on constraints that need to be fulfilled are lowered. Research in this area usually limits the class of queries that are considered, to be able to deal with tractable cases. It remains an open issue to find and describe more tractable and scalable classes of queries for consistent query answering.

Work in this line of research include systems for answering queries in the relational world, e.g., ConQuer (see Section 9.2.2) or HIPPO (see Section 9.2.2), the use of preference relationships among tuples, e.g., (Staworko et al., 2006), including aggregate constraints (Flesca et al., 2005; Bertossi et al., 2005), application of consistent query answering to data warehouses (Bertossi et al., 2009), and using update operations instead of insert/delete operations as means to compute repairs (Wijsen, 2003). Recent work also includes modeling the computation of consistent answers as an optimization problem and solving it heuristically (Bohannon et al., 2005), and looking at specific tractable cases (Wijsen, 2009; Staworko and Chomicki, 2010). Finally, (Bertossi and Chomicki, 2003; Bertossi, 2006; Chomicki, 2008) survey the field, and give an overview and additional literature.

9.2 Data Fusion Systems

The remaining related work in this chapter is concerned with data integration systems with data fusion capabilities. We review the most important systems and refer the interested reader to (Bleiholder and Naumann, 2008) for more detailed descriptions. Out of all research-based information integration systems built so far, only some are actually able to handle conflicting data. Most integration systems only cope with schematic conflicts, as this has been the field with the longest research history. Some also consider identity conflicts and acknowledge the existence of duplicate records in different sources. However, only a few of them recognize and deal with all possible varieties of intra- and inter-source data conflicts. These systems take data conflict handling seriously and often return a complete and/or concise answer. They can be divided into two groups: The first group recognizes and handles, but does not resolve conflicting data (conflict avoiding systems). They implement a simple strategy to handle data conflicts, e.g., by preferring data from one specific source. The second group detects data conflicts and uses (nearly all) existing information to come up with a good conflict resolution.

In comparison to research systems, commercially available information systems have only limited data fusion capabilities, if at all. The problem of duplicate records has received relatively large attention in companies and therefore there exist, in addition to standard DBMSs, many specialized software products that are able to detect

duplicates in data. However, how to fuse duplicate records into one representation in the end often is also an open issue there, and most products use a survivor strategy by choosing one of the existing representations and additionally replace NULL values with data values from other representations. In contrast, most ETL tools provide a rich set of standard components that can be used to implement a variety of data fusion strategies.

9.2.1 Conflict Resolving Systems

All systems in this group allow for full resolution of conflicting data, using a variety of strategies and following different implementation paradigms. The result of a query to these systems is in most cases a complete and concise answer.

Multibase MULTIBASE is one of the first multi-database systems that mentions and considers the integration of data from different sources (Dayal, 1983; Dayal and Hwang, 1984; Landers and Rosenberg, 1982). Data in the sources is described using the DAPLEX data model (Shipman, 1981), allowing also for the definition of global views on this local data.

Data describing the same types of entities in different sources is integrated in the global view using the principle of generalization. A global class always generalizes several local classes. (Dayal, 1983) is among the first to identify the problem of having data conflicts in the global view when integrating data from the local sources using generalization. The work also describes how to combine instances from the sources if they are not disjoint and how to cope with conflicting data.

The solution implemented in MULTIBASE is the use of an *outer join* operation, followed by an *aggregation*, where conflicting values are fused by using basic aggregation functions such as MIN, MAX, SUM, COUNT, or AVERAGE. MULTIBASE cannot resolve conflicts between intra-source duplicates because of the use of *outer joins* (see Section 9.1.1).

Hermes In the HERMES[1] system (Subrahmanian et al., 1995; Adali et al., 1996) different sources and reasoning facilities can be combined by a mediator to enable uniform access for an end user. The mediator is specified by an expert in a declarative way and serves to detect and handle several types of schematic conflicts as well as data conflicts.

The mediator is specified using a variant of annotated logic, annotations being the time that the concept, object, or value was entered into the system and the reliability of it. Each content of the database is true with that specified reliability at the given time.

Conflicts on schema and on data level are recognized, but only partly solved, e.g., using reliability and timestamp information. That way it is only possible to implement a TRUST YOUR FRIENDS or a KEEP UP TO DATE strategy within the system. However, the general ability to also choose particular values (e.g., the maximum or minimum value) in a limited way also allows other strategies.

Fusionplex The 'plex' family of systems (see (Motro et al., 2004b) for an overview) consists of the three systems MULTIPLEX (Motro, 1999), FUSIONPLEX (Motro et al., 2004a; Motro and Anokhin, 2006), and AUTOPLEX (Berlin and Motro, 2006). MULTIPLEX is a virtual multidatabase system, that is capable of integrating information from different heterogeneous sources and serves as basis for the other two systems. FUSIONPLEX adds inconsistency recognition and reconciliation facilities based on provided quality information of the data source contents as additional *feature attributes*. AUTOPLEX adds support for automatically adding new data sources.

In all systems, the schemata of the integrated data sources are considered to be different and overlapping, but they do not contain errors, which means that a mapping from the integrated sources to the global schema can be found (GLaV approach). Also, the content of the data sources is considered to contain errors, which means that duplicate objects, different and contradicting object representations may exist.

FUSIONPLEX allows the user to resolve extensional inconsistencies at tuple level. The system groups tuples representing the same object according to a global key. It then uses quality metadata (e.g., timestamp, cost,

[1] Heterogeneous Reasoning and Mediator System

accuracy, availability, clearance, ...), represented in additional feature columns, to choose only high-quality tuples to be used in the fusion process. User preferences in the importance of the features, are taken into account. Next, fusion functions (MIN, MAX, AVERAGE, ANY, ...) are applied attribute-wise per object to come up with one value per attribute and object, the final object representation. Last, new feature values are computed for the final representation. The main drawbacks of FUSIONPLEX are its focus on quality features (which are distinct from the actual data) without convincingly explaining how to get these and the resulting need to redefine relational operators to include these features (similar to some of the techniques describes in Section 9.1.3).

9.2.2 Conflict Avoiding Systems

The systems in this group allow for handling conflicting data by conflict avoidance. They do not resolve conflicts, but nevertheless handle inconsistent data. They do not regard the conflicting values before deciding on how to handle inconsistencies, but take a quick decision on whether to handle inconsistencies at all and if yes, which data value to use. The result of these systems is at least a concise answer.

TSIMMIS TSIMMIS (The Stanford IBM Manager of Multiple Information Sources) uses wrappers to extract data from sources and convert it into its own data model OEM (Object Exchange Model) (Hammer et al., 1997; Garcia-Molina et al., 1997). OEM is a semi-structured data model, where data is represented as a quadruple (Object-ID, label, type, value) and there is no defined schema as in the relational model.

To identify objects, the system uses a skolem function to assign a unique global identifier to real-world objects. The existence of representations of same objects in different sources (inter-source duplicates) is recognized, as well as possible data conflicts between them. However, in the presence of conflicting data, the system does not allow for resolution of the conflicts in the mediator, but avoids them using a simple TRUST YOUR FRIENDS strategy by specifying a preferred source during mediator specification (Papakonstantinou et al., 1996). The value from that preferred source is then used in the final representation.

SIMS and Ariadne The SIMS system (Services and Information Management for decision Systems) is a mediator-wrapper system and serves as implementation platform for integration projects from different domains (Ambite et al., 2001; Knoblock, 1995; Arens et al., 1996).

The main focus of this project is query planning in an AI setting. However, incompleteness and inconsistency in and between sources is recognized as a problem in such a data integration setting. Handling these problems is however deferred to future work.

ARIADNE extends the SIMS system to a WWW environment (Knoblock et al., 1998; Ambite et al., 1998, 2001). It consists of wrappers for web sources and a central mediator which is responsible for query processing. The main concerns are how to apply AI fashion query planning to a large scale environment such as the web using adequate source descriptions.

ARIADNE recognizes the problem of same real-world entities having multiple representations in different sources, although the problem of intra-source duplicates is ignored. It uses manually constructed mapping tables to connect object identifiers in different sources. How conflicts in other attributes are handled is not described, although for object identifiers a TRUST YOUR FRIENDS strategy is used, in that values for object identifiers are taken from a designated source. As future work, developing semi-automatic learning techniques for creating the mapping tables, is mentioned.

Infomix A mediator-wrapper system following the consistent query answering approach is the INFOMIX system (Eiter et al., 2003; Lembo et al., 2002; Leone et al., 2005) (see Section 9.1.3 for more information on consistent query answering). In contrast to all other systems following this approach, very general queries could be answered because of the approach taken here, namely evaluating logic programs. However, the inherent complexity in evaluating logic programs constrains the use to only small databases. The problem of duplicate detection is not mentioned explicitly, but as data fusion is handled by only including consistent answers in the result, INFOMIX is aware of both problems (duplicate detection and data fusion) and even offers a solution for the latter.

HIPPO The HIPPO system also follows the consistent query answering paradigm (Chomicki et al., 2004a,b). Efficiently returning consistent answers heavily depends on the possibility to efficiently compute repairs, respectively to avoid having to compute all repairs. HIPPO solves this problem by using a special data structure, the conflict hypergraph. HIPPO can be used with select-join-union-difference queries and a special class of dependencies. The system is implemented on top of a relational DBMS, holding the conflict hypergraphs in main memory. Candidates are computed by the database system whereas the final consistent answer is computed with the help of the data structure. However, main memory restricts the size of the conflict hypergraph and therefore the size of the databases that could be queried efficiently. Integration and duplicate detection capabilities depend on the underlying DBMS, although the system as a whole offers a solution to the data fusion problem, by returning only consistent answers, thus implementing a No Gossiping strategy.

ConQuer The ConQuer system in contrast, uses a rewriting technique to compute consistent answers (Fuxman et al., 2005a,b). Given a set of key constraints, the system rewrites an SQL query into an equivalent one that only returns consistent answers. In contrast to other consistent query answering systems, the rewriting works with any underlying relational DBMS and does not need any additional processing. As in all other consistent query answering systems, duplicate detection is reduced to assuming a global key and data conflicts are avoided by only returning consistent answers (No Gossiping).

Rainbow Another line of research in consistent query answering is followed by the Rainbow system (Caroprese and Zumpano, 2006; Caroprese et al., 2006), which is a framework for implementing different ways of answering queries to inconsistent integrated databases in the presence of integrity constraints. A relational integration operation, such as *prioritized merge* (see Section 9.1.2) or the *match join* (see Section 9.1.1) are combined with a second step of returning consistent answers: The approach taken here is to introduce preference relations among different repairs and choose answers in preferred repairs.

9.2.3 Other Systems

Finally, we briefly cover some research-based data cleansing systems, that are neither plain conflict resolving nor conflict avoiding systems but nevertheless consider different aspects of handling conflicts.

Ajax The data cleansing process in the Ajax system (Galhardas et al., 2000a,b, 2001) is defined as a workflow on a logical layer using a declarative language. Besides relational operators (*select*, *project*, *join*) it uses specialized operators for data transformation (e.g., *format* or *unit conversion*) and two more advanced operators for *matching* and *merging*. Duplicate detection (*matching*) is done by grouping object representations assigning keys, whereas the *merging* operator resolves existing data conflicts by aggregation.

The system accounts for data conflicts and provides a basic mechanism for their resolution. However, conflicts are not classified and the *merging* operation is only briefly covered. The main focus of the Ajax system is on the matching step and an efficient and effective way of detecting duplicates while at the same time being able to specify, translate and execute a complex data cleansing workflow.

Potter's Wheel Potter's Wheel is a data cleansing system that operates on one source only (Raman et al., 1999; Raman and Hellerstein, 2001). The main focus is on the interactive definition of cleansing operations by a user. The system is freely available and consists of a GUI to the relational data the user wants to clean. He only sees a sample of the entire data, as the system aims at providing real-time cleansing facilities.

A discrepancy detection process runs in the background and shows to the user data values, or whole columns that supposedly are dirty, e.g., that contain outliers. The system does not only detect numerical outliers, but also string values that are formatted in a different way than the other values in that column. Using the discrepancies detected by the system as hints, a user can define transformations on the data.

The idea of duplicate representations of same real-world objects does not exist as concept in the system. Therefore there is no explicit possibility to detect such duplicate representations and combine them into only one representation. In our three step integration process, a system like Potter's Wheel could be used

before the *duplicate detection* step, in order to increase the quality of the data to be able to more easily detect duplicates.

XClean XClean (Weis and Manolescu, 2007b,a) is an XML data cleansing system in the spirit of Ajax and ETL tools. It allows to declaratively specify a data cleansing workflow on XML data in a special language – XClean/PL – and is of modular nature as it allows to build a workflow out of a library of small cleansing components (operators). Statements in XClean/PL are then compiled into XQuery statements and executed by an XQuery engine, thereby taking advantage of its optimizer and storage engine.

Operators that come with the system focus on the *duplicate detection* step in an information integration process and include among others operators such as *scrubbing* to remove errors in text (typos, format, etc.), *pairwise duplicate classification* to find fuzzy duplicates, or *duplicate filtering* to filter out non-duplicates. The problem of data conflicts is specifically acknowledged and a special operator exists – *fusion* – to include data fusion into data cleansing workflows. However, as XClean focuses on duplicate detection, only a few examples are given for the internals of the *fusion* operator, which essentially works by choosing values among the input (a cluster of representations) to form the output (a single representation).

Science is always wrong. It never solves a problem without creating ten more.

(George Bernard Shaw)

10

Conclusion and Outlook

Data fusion is one step in a general data integration process and handles data conflicts among multiple digital representations of single real-world objects. In this thesis we considered this particular step of data fusion, the step of actually integrating data from different data sources in the context of information integration systems. Unlike the problems of naming and identity conflicts that are solved by schema mappings and duplicate detection techniques, the problem of data fusion has not been received as much attention in the past.

This thesis first outlined the general problem of combining different object representations in form of a data integration process. The main goal of concise and complete integration is also defined, as well as the concept of data conflict. When considering data conflicts, we distinguished between missing and contradicting data and defined a catalog of conflict handling strategies. The concept of conflict resolution function is also defined, and examples are given how theses functions can be used to implement conflict handling strategies and finally to resolve conflicts. Basic notation is formally introduced and subsequently used to define three data fusion operators: To handle missing information the thesis defined the *subsumption* and the *complementation* operators and subsequently their combination with *outer union* resulting in the *minimum union* and *complement union* operators. Whereas *subsumption* and *minimum union* have been introduced before, *complementation* and *complement union* are newly defined in this thesis. At last, the *conflict resolution* and *data fusion* operators are defined that are able to handle contradicting information by applying conflict resolution functions on the data. An extension to SQL allows to express queries using the operators.

The more practical aspects of data fusion are also covered in this thesis and different ways of implementing and using the operators are given. This thesis presented different implementation variants for all operators and introduced efficient techniques to compute *subsumption*, *complementation* and the *data fusion* operator. We also introduced the three operators into logical relational query optimization by giving transformation rules to move the operators around in data fusion query trees. Extensive experimentation shows the feasibility and scalability of the proposed techniques but also the differences between the variants. The performance gain by some of the transformation rules is also shown by experiments.

Last, this thesis covered also the surrounding of data fusion: the two research prototypes that have been developed within our research group and an extensive survey of related work. In general, there is no common way of fusing data and resolving data conflicts. Data fusion and the application of data fusion strategies is domain dependent. Strategies that work in one domain do not necessarily work in another domain. A better understanding of how people actually solve data conflicts under different circumstances would be beneficial and could guide future work in the field. A first step has been done with the MANDUP experiments; a next step could be an extensive multi-domain survey that studies the underlying reasons why people prefer one operator over another or choose specific conflict resolution functions. Beneficial input for such a user study can also come from other fields, such as the field of decision making in psychology that studies how people come to a decision given complete or incomplete information. Summarizing, the main contributions of this work are:

Classification of data fusion strategies and survey of existing techniques and systems

With this thesis we gave the first extensive survey of the field of data fusion techniques and information integration systems developed so far and described and compared their abilities and limitations. In addition we presented a classification of conflict handling strategies (Bleiholder and Naumann, 2006a,b)

that can be implemented by a data fusion technique or within an information integration system. The results of this survey have been published in (Bleiholder and Naumann, 2008) and have been summarized in Chapter 9, the conflict handling strategies have been presented as part of Chapter 3. We furthermore showed how the different strategies can be implemented as part of an information integration system. The classification is open to more strategies or even classes of strategies. Input from other fields, such as philosophy, decision theory, psychology, etc. is expected to extend the existing classification in the future.

Formalization of data fusion

In this thesis we embedded *data fusion* as a third main step in the more general information integration process after *schema matching* and *duplicate detection*. We defined a versatile *data fusion* operator that can handle conflicting information and can be used to implement most of the data fusion strategies given. We showed how such a *data fusion* operator can be embedded into the relational world by giving an extension to SQL to express data fusion queries (Bleiholder and Naumann, 2005, 2004; Bleiholder, 2005) and transformation rules that can be used by a relational query optimizer to move the operator around in query trees. In addition, we described two special case operators for handling missing information: *subsumption* has previously been defined and *complementation* has been introduced in this thesis. *Subsumption* and *complementation* have been defined as distinct operators. An alternative definition is the combination of *subsumption* and *complementation* in one operator or the inclusion of the removal of subsumed tuples in the *complementation* operator. Exploring this line of future work has implications both in the design of algorithms and in the definition of transformation rules. However, it is unclear but unlikely that implementations of a combined algorithm would be significantly faster than either one alone. On the other hand, applying a combined operator instead of both operators one after the other could be beneficial in case both subsumed and complementing tuples need to be handled. Future work in this direction also needs to consider if the possible performance gain is worth the reduced flexibility.

Implementation and query optimization for subsumption and minimum union

Minimum union is a relational operator where subsumed tuples are removed from the result of an *outer union*. The *subsumption*/*minimum union* operation is one possible way of implementing a specific data fusion strategy. However, an efficient implementation of the *subsumption* operation has been missing so far. This thesis filled this specific gap by presenting and comparing different implementation alternatives (Bleiholder et al., 2010a). It additionally introduced *subsumption* into relational query optimization by giving transformation rules that can be used to move the *subsumption* operator around in query plans. Possible extensions of this line of work are the algorithmic improvement of the index based approach and the introduction of a subsumption index and the corresponding index management. A subsumption index would only hold the tuples of a relation that are not subsumed. An interesting topic would also be the combination of indexes: is it possible to compute the *minimum union* by simply combining the subsumption indexes of the base tables? Another line of future work that has only briefly been discussed is the inclusion of techniques that are used in set containment joins. As a self-set-containment join removes many, but not all, subsumed tuples, adopting the techniques used there, especially the hashing techniques, could be worthwhile.

Implementation and query optimization for complementation and complement union

Another way of dealing with missing data and implementing a specific data fusion strategy are the *complementation* operator and the *complement union* operator. Similarly to *subsumption* and *minimum union*, *complement union* is the combination of *complementation* and *outer union*. *Complementation* has been firstly defined as part of this work (Bleiholder et al., 2010a) and combines information from tuples with complementing attribute values. This thesis presented different implementation variants of the *complementation* operator (Bleiholder et al., 2010b) and introduced the operator into logical query optimization. The transformation rules given can be used to move the operator around in query trees. In future work, the presented algorithms for *complementation* can be improved, especially the line of work inspired by the *subsumption* operator, the null-pattern approach. Another possible extension

is the inclusion of other operators from related work into the optimization framework. Some of the operators are easily introduced, e.g., the *merge* operator, because it can be expressed using other relational operators. Transformation rules for other operators, such as *match join* or *full disjunction*, are more difficult. However, introducing other data fusion operators into relational query optimization as well as a better and more detailed cost model would result in a complete data fusion optimization framework.

Research prototypes HumMer and FuSem

Lastly, this thesis contributed a large part to two research systems: the extension of Sql as presented in Chapter 4 is used in the HumMer system (Bilke et al., 2005; Naumann et al., 2006), as well as the *data fusion* operator, also defined in Chapter 4. Another research system that has been developed within our research group is the FuSem system (Bleiholder et al., 2007). It implements different data fusion semantics and allows the user to compare them. Simple visualization of differences of fusion results as done in the FuSem system is already beneficial for the user. However, in future work, this could be improved to include hints as to what result is best in terms of completeness and conciseness. Combining the visualization of the FuSem system with features from the ManDup system is also possible: watching the user solve a few sample conflicts and inferring the conflict resolution functions that best describe the way the user solved the conflicts is the first step in including (semi-)automatic conflict resolution (by example) into data integration systems.

As conflict resolution is difficult to be done completely automatically and will almost always require a human domain expert to ensure a high reliability of conflict resolution, people will always play a major role. Thus, this thesis covered the process of data fusion as part of a data integration process in its entirety and shows where human involvement can be minimized and where not: When fusing data, an expert user can first choose among several strategies to fuse data, and expresses the strategies using an integrated information system. Then, the system executes the operation in an efficient way and finally lets the user look at the result and explore differences between different fusion techniques/strategies.

Glossary

Notation	Description
$\bowtie_{\vert}$	*right outer join* operator.
γ	*aggregation* operator.
λ	*conflict resolution* operator.
$\times$	*cartesian product* operator.
κ	*complementation* operator.
$\boxplus$	*complement union* operator.
δ	*distinct* operator.
ϕ	*data fusion* operator.
ϕ^{β}	*data fusion* operator with implicit subsumption.
$\bowtie_{c}$	*join* operator, using condition c.
$\boxtimes$	*merge* operator.
$\oplus$	*minimum union* operator.
$\bowtie$	*natural join* operator.
$\uplus$	*outer union* operator.
$\triangleleft$	*prioritized* merge operator.
π	*projection* operator.
σ	*selection* operator.
$\tilde{\bowtie}$	*similarity join* operator.
τ	*sort* operator.
β	*subsumption* operator.
$\cup$	*union* operator.
$\gtrless$	complements relationship.
$\sqsubset$	subsumed relationship, inverse of $\sqsupset$.
$\sqsupset$	subsumes relationship.
$_{\vert}\bowtie$	*left outer join* operator.
$_{\vert}\bowtie_{\vert}$	*full outer join* operator.

List of Figures

List of Tables

Bibliography

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley, Reading, MA, 1st edition.

Adali, S., Candan, K. S., Papakonstantinou, Y., and Subrahmanian, V. S. (1996). Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 137–146. ACM Press.

Aho, A. V., Sagiv, Y., and Ullman, J. D. (1979). Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246.

Ambite, J. L., Ashish, N., Barish, G., Knoblock, C. A., Minton, S., Modi, P. J., Muslea, I., Philpot, A., and Tejada, S. (1998). Ariadne: a system for constructing mediators for internet sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 561–563. ACM Press.

Ambite, J. L., Knoblock, C. A., Muslea, I., and Philpot, A. G. (2001). Compiling source descriptions for efficient and flexible information integration. *Journal of Intelligent Information Systems*, 16(2):149–187.

Arenas, M., Bertossi, L. E., and Chomicki, J. (1999). Consistent query answers in inconsistent databases. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 68–79. ACM Press.

Arens, Y., Knoblock, C. A., and Shen, W.-M. (1996). Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 6(2/3):99–130.

Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley, Essex, UK.

Batini, C., Lenzerin, M., and Navathe, S. B. (1986). A comparative analysis of methodologies for database schema integration. *ACM Computing Survey*, 18(4):323–364.

Berlin, J. and Motro, A. (2006). Tuplerank: Ranking discovered content in virtual databases. In *Proceedings of the International Workshop on Next Generation Information Technologies and Systems (NGITS)*, pages 13–25.

Bertossi, L. E. (2006). Consistent query answering in databases. *SIGMOD Record*, 35(2):68–76.

Bertossi, L. E., Bravo, L., Franconi, E., and Lopatenko, A. (2005). Complexity and approximation of fixing numerical attributes in databases under integrity constraints. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, pages 262–278.

Bertossi, L. E., Bravo, L., and Marileo, M. C. (2009). Consistent query answering in data warehouses. In *Proceedings of the Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*.

Bertossi, L. E. and Chomicki, J. (2003). Query answering in inconsistent databases. In *Logics for Emerging Applications of Databases*, pages 43–83.

Bilke, A., Bleiholder, J., Böhm, C., Draba, K., Naumann, F., and Weis, M. (2005). Automatic data fusion with HumMer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1251–1254.

Bilke, A. and Naumann, F. (2005). Schema matching using duplicates. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 69–80.

Bleiholder, J. (2005). A relational operator approach to data fusion. In *Proceedings of VLDB 2005 PhD Workshop*, Trondheim, Norway.

Bleiholder, J., Draba, K., and Naumann, F. (2007). Fusem - exploring different semantics of data fusion. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1350–1353.

Bleiholder, J. and Naumann, F. (2004). FUSE BY: Syntax und semantik zur informationsfusion in SQL. In *INFORMATIK 2004, Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, pages 331–335, Ulm.

Bleiholder, J. and Naumann, F. (2005). Declarative data fusion - syntax, semantics, and implementation. In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 58–73.

Bleiholder, J. and Naumann, F. (2006a). Conflict handling strategies in an integrated information system. Technical report HUB-IB-197, Humboldt-Universität zu Berlin.

Bleiholder, J. and Naumann, F. (2006b). Conflict handling strategies in an integrated information system. In *Proceedings of the Workshop on Information Integration on the Web (IIWeb)*.

Bleiholder, J. and Naumann, F. (2008). Data fusion. *ACM Computing Survey*, 41(1):1–41.

Bleiholder, J., Szott, S., Herschel, M., Kaufer, F., and Naumann, F. (2010a). Subsumption and complementation as data fusion operators. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 513–524, Lausanne, Switzerland.

Bleiholder, J., Szott, S., Herschel, M., and Naumann, F. (2010b). Complement union for data integration. In *Proceedings of the International Workshop on New Trends in Information Integration (NTII)*.

Bohannon, P., Flaster, M., Fan, W., and Rastogi, R. (2005). A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 143–154.

Bron, C. and Kerbosch, J. (1973). Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577.

Buneman, P., Khanna, S., and Tan, W. C. (2001). Why and where: A characterization of data provenance. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 316–330. Springer-Verlag.

Burdick, D., Deshpande, P., Jayram, T. S., Ramakrishnan, R., and Vaithyanathan, S. (2005). OLAP over uncertain and imprecise data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 970–981.

Calvo, T., Mayor, G., and Mesiar, R., editors (2002). *Aggregation Operators - New Trends and Applications*. Physica-Verlag, Heidelberg.

Careem, M., De Silva, C., De Silva, R., Raschid, L., and Weerawarana, S. (2006). Sahana: Overview of a disaster management system. In *Proceedings of the International Conference on Information and Automation*.

Caroprese, L., Greco, S., Trubitsyna, I., and Zumpano, E. (2006). Preferred generalized answers for inconsistent databases. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 344–349.

Caroprese, L. and Zumpano, E. (2006). A framework for merging, repairing and querying inconsistent databases. In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 383–398.

Cholewa, W. (1985). Aggregation of fuzzy opinions – an axiomatic approach. *Fuzzy Sets and Systems*, 17(3):249–258.

Cholvy, L. and Garion, C. (2002). Answering queries addressed to several databases: A query evaluator which implements a majority merging approach. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 131–139.

Cholvy, L. and Garion, C. (2004). Querying several conflicting databases. *Journal of Applied Non-Classical Logics*, 14(3):295–327.

Cholvy, L. and Moral, S. (2001). Merging databases: Problems and examples. *International Journal of Intelligent Systems*, 16(10):1193–1221.

Chomicki, J. (2008). Consistent query answering: The first ten years. In *Proceedings of the International Conference on Scalable Uncertainty Management (SUM)*, pages 1–3.

Chomicki, J., Marcinkowski, J., and Staworko, S. (2004a). Computing consistent query answers using conflict hypergraphs. In *Proceedings of the ACM conference on Information and Knowledge Management (CIKM)*, pages 417–426. ACM Press.

Chomicki, J., Marcinkowski, J., and Staworko, S. (2004b). Hippo: A system for computing consistent answers to a class of SQL queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 841–844.

Chow, S. and Ruskey, F. (2004). Drawing area-proportional venn and euler diagrams. In Liotta, G., editor, *Graph Drawing, Perugia, 2003*, pages pp. 466–477. Springer.

Codd, E. F. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434.

Cohen, S., Fadida, I., Kanza, Y., Kimelfeld, B., and Sagiv, Y. (2006). Full disjunctions: polynomial-delay iterators in action. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 739–750. VLDB Endowment.

Cohen, S. and Sagiv, Y. (2005). An incremental algorithm for computing ranked full disjunctions. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 98–107. ACM Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, second edition.

Crestani, F., Lalmas, M., Rijsbergen, C. J. V., and Campbell, I. (1998). "is this document relevant?. . .probably": a survey of probabilistic models in information retrieval. *ACM Computing Survey*, 30(4):528–552.

Darwen, H. and Date, C. J. (1995). The third manifesto. *SIGMOD Record*, 24(1):39–49.

Dayal, U. (1983). Processing queries over generalization hierarchies in a multidatabase system. In Schkolnick, M. and Thanos, C., editors, *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 342–353, Florence, Italy. Morgan Kaufmann.

Dayal, U. and Hwang, H.-Y. (1984). View definition and generalization for database system integration in a multidatabase system. *IEEE Transactions on Software Engineering (TSE)*, 10(6):628–645.

DeMichiel, L. G. (1989). Resolving database incompatibility: An approach to performing relational operations over mismatched domains. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1(4):485–493.

den Bercken, J. V., Blohsfeld, B., Dittrich, J.-P., Krämer, J., Schäfer, T., Schneider, M., and Seeger, B. (2001). XXL - a library approach to supporting efficient implementations of advanced database queries. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 39–48. Morgan Kaufmann Publishers Inc.

Detyniecki, M. (2001). Numerical aggregation operators: State of the art. In *International Summer School on Aggregation Operators and their Applications*, Asturias, Spain.

Dubois, D. and Prade, H. (1988). On the combination of uncertain or imprecise pieces of information in rule-based systems - a discussion in the framework of possibility theory. *International Journal of Approximate Reasoning*, 2(1):65–87.

Dubois, D. and Prade, H. (2004). On the use of aggregation operations in information fusion processes. *Fuzzy Sets and Systems*, 142(1):143–161.

Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.-E., and Zemke, F. (2004). SQL:2003 has been published. *SIGMOD Record*, 33(1):119–126.

Eiter, T., Fink, M., Greco, G., and Lembo, D. (2003). Efficient evaluation of logic programs for querying data integration systems. In *Proceedings of the International Conference on Logic Programming (ICLP)*, pages 163–177.

Euzenat, J. and Shvaiko, P. (2007). *Ontology Matching*. Springer.

Fagin, R., Kolaitis, P. G., and Popa, L. (2005). Data exchange: getting to the core. *ACM Transactions on Database Systems (TODS)*, 30(1):174–210.

Flesca, S., Furfaro, F., and Parisi, F. (2005). Consistent query answers on numerical databases under aggregate constraints. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, pages 279–294.

Fuhr, N. (2001). Models in information retrieval. In Agosti, M., Crestani, F., and Pasi, G., editors, *Lectures on Information Retrieval*, volume 1980 of *Lecture Notes in Computer Science*, pages 21–50. Springer Berlin / Heidelberg.

Fuxman, A., Fazli, E., and Miller, R. J. (2005a). ConQuer: efficient management of inconsistent databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 155–166. ACM Press.

Fuxman, A., Fuxman, D., and Miller, R. J. (2005b). ConQuer: A system for efficient querying over inconsistent databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1354–1357.

Galhardas, H., Florescu, D., Shasha, D., and Simon, E. (2000a). AJAX: An extensible data cleaning tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 590. ACM Press.

Galhardas, H., Florescu, D., Shasha, D., and Simon, E. (2000b). An extensible framework for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 312.

Galhardas, H., Florescu, D., Shasha, D., Simon, E., and Saita, C.-A. (2001). Declarative data cleaning: Language, model, and algorithms. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 371–380.

Galindo-Legaria, C. and Rosenthal, A. (1997). Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)*, 22(1):43–74.

Galindo-Legaria, C. A. (1994). Outerjoins as disjunctions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 348–358. ACM Press.

Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J. (1997). The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132.

Gigerenzer, G. and Todd, P. M., editors (1999). *Simple Heuristics That Make Us Smart*. Oxford University Press.

Glassner, A. (2003). Venn and now. *IEEE Computer Graphics and Applications*, 23(4):82–95.

Greco, S., Pontieri, L., and Zumpano, E. (2001). Integrating and managing conflicting data. In *Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 349–362. Springer-Verlag.

Hammer, J., McHugh, J., and Garcia-Molina, H. (1997). Semistructured data: The TSIMMIS experience. In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 1–8.

Hernández, M. A., Popa, L., Velegrakis, Y., Miller, R. J., Naumann, F., and Ho, C.-T. (2002). Mapping XML and relational schemas with Clio. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 498–499.

Hernández, M. A. and Stolfo, S. J. (1995). The merge/purge problem for large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 127–138.

Hernández, M. A. and Stolfo, S. J. (1998). Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37.

ISO/IEC 9075-*:2003 (2003). *Database Languages - SQL*. ISO/IEC, Geneva, Switzerland.

Johnson, D. S., Papadimitriou, C. H., and Yannakakis, M. (1988). On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123.

Kanza, Y. and Sagiv, Y. (2003). Computing full disjunctions. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 78–89. ACM Press.

Kim, W., Choi, B.-J., Hong, E.-K., Kim, S.-K., and Lee, D. (2003). A taxonomy of dirty data. *Data Mining and Knowledge Discovery*, 7(1):81–99.

Knoblock, C. A. (1995). Planning, executing, sensing, and replanning for information gathering. In Mellish, C., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1686–1693. Morgan Kaufmann.

Knoblock, C. A., Minton, S., Ambite, J. L., Ashish, N., Modi, P. J., Muslea, I., Philpot, A. G., and Tejada, S. (1998). Modeling web sources for information integration. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 211–218, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Kobayashi, M. and Takeda, K. (2000). Information retrieval on the web. *ACM Computing Survey*, 32(2):144–173.

Kruppa, J. (2005). Minimum union als DB2 table function. Studienarbeit, Humboldt-Universität zu Berlin.

Lam, K. W. and Leung, C. H. (2004). Rank aggregation for meta-search engines. In *Proceedings of the International World Wide Web Conferences (WWW)*, pages 384–385. ACM Press.

Landers, T. and Rosenberg, R. L. (1982). An overview of MULTIBASE. In Schneider, H. J., editor, *Proceedings of the Second International Symposium on Distributed Data Bases*. North Holland Publishing Company.

Larson, P.-Å. and Zhou, J. (2005). View matching for outer-join views. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 445–456.

Lembo, D., Lenzerini, M., and Rosati, R. (2002). Source inconsistency and incompleteness in data integration. In *Proceedings of the Workshop on Knowledge Representation Meets Databases (KRDB)*.

Leone, N., Greco, G., Ianni, G., Lio, V., Terracina, G., Eiter, T., Faber, W., Fink, M., Gottlob, G., Rosati, R., Lembo, D., Lenzerini, M., Ruzzi, M., Kalka, E., Nowicki, B., and Staniszkis, W. (2005). The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 915–917.

Levenshtein, V. (1965). Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17.

Li, A.-P. and Wu, Q. (2004). On aggregation operators for fuzzy information sources. In *Proceedings of ER (Workshops)*, pages 223–233.

Lim, E.-P., Cao, Y., and Chiang, R. H. L. (1997). Source-aware multidatabase query processing. In *Proceedings of the International Workshop on Engineering Federated Database Systems (EFDBS)*, pages 69–80.

Lim, E.-P., Srivastava, J., and Hwang, S.-Y. (1995). An algebraic transformation framework for multidatabase queries. *Distributed and Parallel Databases*, 3(3):273–307.

Lim, E.-P., Srivastava, J., and Shekhar, S. (1994). Resolving attribute incompatibility in database integration: An evidential reasoning approach. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 154–163. IEEE Computer Society.

Madhavan, J., Bernstein, P. A., and Rahm, E. (2001). Generic schema matching with Cupid. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 49–58.

Makino, K. and Uno, T. (2004). New algorithms for enumerating all maximal cliques. In *Proceedings of Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 260–272.

Melnik, S., Bernstein, P. A., Halevy, A., and Rahm, E. (2005). Supporting executable mappings in model management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 167–178.

Melnik, S. and Garcia-Molina, H. (2003). Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems (TODS)*, 28(1):56–99.

Melnik, S., Garcia-Molina, H., and Rahm, E. (2002). 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.

Mitra, M. and Chaudhuri, B. B. (2000). Information retrieval from documents: A survey. *Information Retrieval*, 2(2-3):141–163.

Modani, N. and Dey, K. (2008). Large maximal cliques enumeration in sparse graphs. In *Proceedings of the ACM conference on Information and Knowledge Management (CIKM)*, pages 1377–1378.

Motro, A. (1986). Completeness information and its application to query processing. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 170–178.

Motro, A. (1999). Multiplex: A formal model for multidatabases and its implementation. In *Proceedings of the International Workshop on Next Generation Information Technologies and Systems (NGITS)*, pages 138–158. Springer-Verlag.

Motro, A. and Anokhin, P. (2006). Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Information Fusion*, 7(2):176–196.

Motro, A., Anokhin, P., and Acar, A. C. (2004a). Utility-based resolution of data inconsistencies. In *Proceedings of the Workshop on Information Quality in Information Systems (IQIS)*, pages 35–43. ACM Press.

Motro, A., Berlin, J., and Anokhin, P. (2004b). Multiplex, Fusionplex, and Autoplex - three generations of information integration. *SIGMOD Record*, 33(4):51–57.

Naumann, F., Bilke, A., Bleiholder, J., and Weis, M. (2006). Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Engineering Bulletin*, 29(2):21–31.

Naumann, F., Freytag, J.-C., and Leser, U. (2004). Completeness of integrated information sources. *Information Systems*, 29(7):583–615.

Naumann, F. and Häussler, M. (2002). Declarative data merging with conflict resolution. In *Proceedings of the International Conference on Information Quality (IQ)*, Cambridge, MA.

Oracle Forums (2010). Complementation — Oracle forums. `http://forums.oracle.com/forums/thread.jspa?messageID=4029852` [Online; accessed January, 25th 2010].

Oxley, M. E. and Thorsen, S. N. (2004). Fusion or integration: What's the difference? In *Proceedings of the Seventh International Conference on Information Fusion*, pages 429–434, Stockholm.

Papakonstantinou, Y., Abiteboul, S., and Garcia-Molina, H. (1996). Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 413–424. Morgan Kaufmann Publishers Inc.

Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., and Fagin, R. (2002). Translating web data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Hong Kong.

Rahm, E. and Bernstein, P. A. (2001). On matching schemas automatically. Technical Report MSR-TR-2001-17, Microsoft Research, Redmond, WA.

Rahm, E. and Do, H. H. (2000). Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13.

Rajaraman, A. and Ullman, J. D. (1996). Integrating information by outerjoins and full disjunctions (extended abstract). In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 238–248. ACM Press.

Raman, V., Chou, A., and Hellerstein, J. M. (1999). Scalable spreadsheets for interactive data analysis. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*.

Raman, V. and Hellerstein, J. M. (2001). Potter's Wheel: An interactive data cleaning system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 381–390. Morgan Kaufmann.

Rao, J., Pirahesh, H., and Zuzarte, C. (2004). Canonical abstraction for outerjoin optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 671–682. ACM Press.

Rijsbergen, C. J. V. (1979). *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA.

Sagiv, Y. (1983). Quadratic algorithms for minimizing joins in restricted relational expressions. *SIAM Journal on Computing*, 12(2):316–328.

Salton, G. and McGill, M. J. (1986). *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA.

Sattler, K., Conrad, S., and Saake, G. (2000). Adding conflict resolution features to a query language for database federations. In Roantree, M., Hasselbring, W., and Conrad, S., editors, *Proceedings of the International Workshop on Engineering Federated Information Systems (EFIS)*, pages 41–52.

Scannapieco, M., Virgillito, A., Marchetti, C., Mecella, M., and Baldoni, R. (2004). The DaQuinCIS architecture: A platform for exchanging and improving data quality in cooperative information systems. *Information Systems*, 29(7):551–582.

Schallehn, E. and Sattler, K.-U. (2003). Using similarity-based operations for resolving data-level conflicts. In *Proceedings of the British National Conference on Databases (BNCOD)*, pages 172–189.

Schallehn, E., Sattler, K.-U., and Saake, G. (2004). Efficient similarity-based operations for data integration. *Data & Knowledge Engineering*, 48(3):361–387.

Shipman, D. W. (1981). The functional data model and the data languages DAPLEX. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173.

Staworko, S. and Chomicki, J. (2010). Consistent query answers in the presence of universal constraints. *Information Systems*, 35(1):1–22.

Staworko, S., Chomicki, J., and Marcinkowski, J. (2006). Preference-driven querying of inconsistent relational databases. In *Proceedings of the International Workshop in Inconsistency and Incompleteness in Databases (IIDB)*, Munich, Germany.

Stix, V. (2004). Finding all maximal cliques in dynamic graphs. *Computational Optimization and Applications*, 27(2):173–186.

Subrahmanian, V. S., Adali, S., Brink, A., Emery, R., Lu, J., Rajput, A., Rogers, T., Ross, R., and Ward, C. (1995). Hermes: A heterogeneous reasoning and mediator system. Technical report, University of Maryland.

Tsai, P. S. M. and Chen, A. L. P. (2000). Partial natural outerjoin - an operation for interoperability in a multidatabase environment. *Journal of Information Science and Engineering*, 16(4):593–617.

Tseng, F. S.-C., Chen, A. L. P., and Yang, W.-P. (1993). Answering heterogeneous database queries with degrees of uncertainty. *Distributed and Parallel Databases*, 1(3):281–302.

Tsikrika, T. and Lalmas, M. (2001). Merging techniques for performing data fusion on the web. In *Proceedings of the ACM conference on Information and Knowledge Management (CIKM)*, pages 127–134. ACM Press.

Tsois, A. and Sellis, T. (2003a). The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies (long version). Technical report TR-2003-4.

Tsois, A. and Sellis, T. K. (2003b). The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 644–655.

Ullman, J. D., Garcia-Molina, H., and Widom, J. (2001). *Database Systems: The Complete Book*. Prentice Hall PTR.

van der Putten, P., Kok, J. N., and Gupta, A. (2002). Why the information explosion can be bad for data mining, and how data fusion provides a way out. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*.

van Rijsbergen, C. J. (2001). Getting into information retrieval. pages 1–20.

Varzi, A. (2009). Mereology. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Summer 2009 edition. `http://plato.stanford.edu/archives/sum2009/entries/mereology/` [Online; accessed August, 31st 2009].

Wang, H. and Zaniolo, C. (2000a). User-defined aggregates in database languages. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, pages 43–60. Springer-Verlag.

Wang, H. and Zaniolo, C. (2000b). Using SQL to build new aggregates and extenders for object- relational systems. In Abbadi, A. E., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 166–175. Morgan Kaufmann.

Weis, M. and Manolescu, I. (2007a). Declarative XML data cleaning with XClean. In *Proceedings of the International Conference and Advanced Information Systems Engineering (CAISE)*, pages 96–110.

Weis, M. and Manolescu, I. (2007b). XClean in action (demo). In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 259–262.

Weis, M. and Naumann, F. (2004). Detecting duplicate objects in XML documents. In *Proceedings of the Workshop on Information Quality in Information Systems (IQIS)*, pages 10–19, Paris, France.

Weis, M. and Naumann, F. (2005). DogmatiX tracks down duplicates in XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442. ACM Press.

Wijsen, J. (2003). Condensed representation of database repairs for consistent query answering. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 378–393.

Wijsen, J. (2009). Consistent query answering under primary keys: a characterization of tractable queries. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 42–52.

Wikipedia (2009). Mereology — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Mereology` [Online; accessed August, 31st 2009].

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Yan, L. L. and Özsu, M. T. (1999). Conflict tolerant queries in AURORA. In *Proceedings of the IFCIS International Conference on Cooperative Information Systems (CoopIS)*, page 279. IEEE Computer Society.

Yan, W. and Larson, P. A. (1993). Performing group-by before join. Technical Report CS 93-46, University of Waterloo, Waterloo, Ontario.

Yan, W. P. and Larson, P.-Å. (1994). Performing group-by before join. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 89–100. IEEE Computer Society.

Yan, W. P. and Larson, P.-Å. (1995). Eager aggregation and lazy aggregation. In Dayal, U., Gray, P. M. D., and Nishio, S., editors, *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 345–357. Morgan Kaufmann.

Yerneni, R., Papakonstantinou, Y., Abiteboul, S., and Garcia-Molina, H. (1998). Fusion queries over internet databases. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 57–71.

Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Dissertation eigenständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel, angefertigt zu haben. Alle öffentlichen Quellen sind als solche kenntlich gemacht. Die vorliegende Arbeit ist in dieser oder anderer Form zuvor nicht als Prüfungsarbeit zur Begutachtung vorgelegt worden.

Potsdam, den 9.11.2010