Subsumption and Complementation as Data Fusion Operators

Jens Bleiholder
Hasso-Plattner-Institut
Potsdam, Germany
jens.bleiholder@hpi.uni-potsdam.de

Sascha Szott*
Konrad-Zuse-Zentrum
für Informationstechnik Berlin
Berlin, Germany
szott@zib.de

Melanie Herschel†
Universität Tübingen
Tübingen, Germany
melanie.herschel@uni-tuebingen.de

Frank Kaufer
Hasso-Plattner-Institut
Potsdam, Germany
frank.kaufer@hpi.uni-potsdam.de

Felix Naumann
Hasso-Plattner-Institut
Potsdam, Germany
felix.naumann@hpi.uni-potsdam.de

ABSTRACT

The goal of data fusion is to combine several representations of one real world object into a single, consistent representation, e.g., in data integration. A very popular operator to perform data fusion is the *minimum union* operator. It is defined as the *outer union* and the subsequent removal of subsumed tuples. Minimum union is used in other applications as well, for instance in database query optimization to rewrite outer join queries, in the semantic web community in implementing SPARQL's OPTIONAL operator, etc. Despite its wide applicability, there are only few efficient implementations, and until now, minimum union is not a relational database primitive.

This paper fills this gap as we present implementations of subsumption that serve as a building block for minimum union. Furthermore, we consider this operator as database primitive and show how to perform optimization of query plans in presence of subsumption and minimum union through rule-based plan transformations. Experiments on both artificial and real world data show that our algorithms outperform existing algorithms used for subsumption in terms of runtime and they scale to large volumes of data.

In the context of data integration, we observe that performing data fusion calls for more than subsumption and minimum union. Therefore, another contribution of this paper is the definition of the *complementation* and *complement union* operators. Intuitively, these allow to merge tuples that have complementing values and thus eliminate unnecessary null-values.

*Research was partially performed while at Hasso-Plattner-Institut.

†Research was partially performed while at Hasso-Plattner-Institut and IBM Almaden.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

Categories and Subject Descriptors

H.2.5 [**Database Management**]: Heterogeneous Databases

General Terms

Algorithms

Keywords

minimum union, complement union, data integration, data quality

1. INTRODUCTION

Data integration is the process of providing users of an integrated information system with a unified view of several data sources. Three main challenges in data integration are (1) to transform the data stored in the sources to the target schema (*schema matching* [22] and *mapping* [14]), (2) to match object representations from different sources representing the same real-world object (*duplicate detection* [8]), and (3) to combine several existing representations of one real world object into a single consistent representation (*data fusion* [3]). In this last step, which is the focus of this paper, special care must be taken when handling data conflicts between the different object representations.

We focus on two alternative operators, namely *minimum union* and *complement union*. Both aim at resolving a special type of data conflict, called uncertainty. Intuitively, the operators consider cases where a concrete value in one tuple conflicts with a NULL value of another tuple, and replaces the NULL value with the NON-NULL value when merging the tuples.

Minimum union combines an outer union with the removal of tuples that are contained in other tuples, so called *subsumed* tuples [12]. For instance, tuple $t_1 = (a, b, c)$ subsumes tuple $t_2 = (a, b, \perp)$, where $\perp$ denotes a NULL value. When applying subsumption, t_2 is removed. This operator has been widely used in the literature [11, 12, 14], however it lacks efficient and generally applicable implementations of the subsumption part, a gap this paper aims to fill.

Complement union combines object representations by first computing the outer union and then combining tuples that *complement* each other. Consider above t_2 and a tuple $t_3 = (a, \perp, c)$: t_2 and t_3 complement each other and combine to a single tuple (a, b, c)

when complementation is applied. To the best of our knowledge, this is the first proposal of an operator with these semantics.

Police

tid	Name	DOB	Sex	Address
1	Miller	7/7/59	m	12 Main
2	Miller	⊥	⊥	12 Main
3	Peters	1/1/53	m	34 First

Hospital

tid	Name	DOB	Sex	Blood
4	Peters	1/1/53	⊥	AB
5	Peters	1/1/53	m	⊥
6	Miller	⊥	f	B
7	Miller	7/7/59	m	O

(1) Schema matching + outer union + duplicate detection

oid	tid	Name	DOB	Sex	Address	Blood
1	1	Miller	7/7/59	m	12 Main	⊥
1	2	Miller	⊥	⊥	12 Main	⊥
2	3	Peters	1/1/53	m	34 First	⊥
2	4	Peters	1/1/53	⊥	⊥	AB
2	5	Peters	1/1/53	m	⊥	⊥
1	6	Miller	⊥	f	⊥	B
1	7	Miller	7/7/59	m	⊥	O

(2.1) Fusion using subsumption

oid	tid	Name	DOB	Sex	Address	Blood
1	1+2	Miller	7/7/59	m	12 Main	⊥
2	3+5	Peters	1/1/53	m	34 First	⊥
2	4	Peters	1/1/53	⊥	⊥	AB
1	6	Miller	⊥	f	⊥	B
1	7	Miller	7/7/59	m	⊥	O

(2.2) Fusion using complementation

oid	tid	Name	DOB	Sex	Address	Blood
1	1+7	Miller	7/7/59	m	12 Main	O
1	2+6	Miller	⊥	f	12 Main	B
2	3+4	Peters	1/1/53	m	34 First	AB
2	4+5	Peters	1/1/53	⊥	⊥	AB
1	2+7	Miller	7/7/59	m	12 Main	O

Figure 1: Integration result after schema matching, outer union, duplicate detection, subsumption (bottom left), and complementation (bottom right)

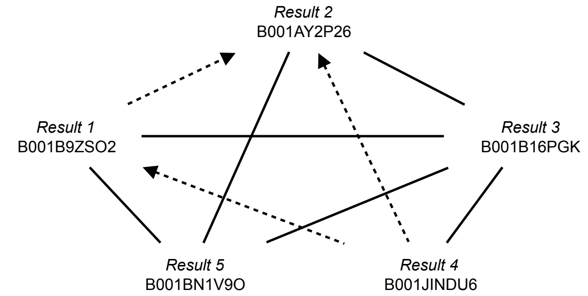
Example. Consider a disaster management scenario where two databases *Police* and *Hospital* are integrated. Excerpts of the two sources are presented in Fig. 1 (top). The attribute *tid* is not part of the relational attributes; its values merely serve as reference to tuples throughout this paper. The goal is to detect missing people that have been admitted to a hospital and contact their relatives when an address is available.

After performing schema matching, and mapping the data into one relation using outer union, duplicate detection is performed. In the result of this first step as depicted in Fig. 1 (center), each tuple corresponds to a person, either reported as missing in the *Police* database or admitted to a hospital according to the *Hospital* database. The attribute *oid* refers to an object ID and is the output of duplicate detection. If two tuples have the same *oid*, they are considered to represent the same person. For instance, tuples with *tid*s $1, 2, 6, 7$ represent the same individual, which we label with *oid* 1, and *tid*s $3, 4, 5$ represent another single individual, identified by *oid* 2.[1]

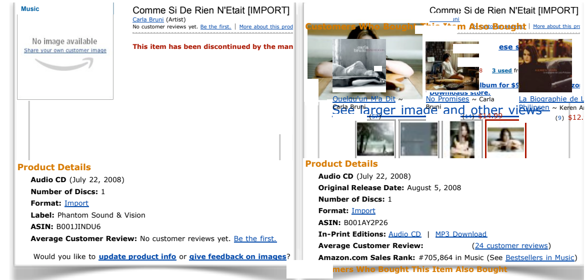
The next and final step in the integration workflow is data fusion that combines tuples that represent the same real-world object. Depending on the chosen strategy and types of conflicts handled by these strategies, the results of data fusion differ. In Fig. 1, we show the result of applying subsumption (bottom left) and complementation (bottom right) as fusion operator on the result of outer union depicted in Fig. 1. Note that in this particular example, these results correspond to the results of minimum union and complement union, respectively, which we could have computed directly. However, having subsumption and complementation as separate operators that are combined with outer union gives us more flexibility, e.g., they can be used to clean the source databases before the integration process and it allows us to potentially use more sophisticated schema mapping algorithms.

When applying subsumption, we see that tuple 2 is combined with tuple 1. This is due to the fact that every NON-NULL value of tuple 2 is also contained in tuple 1, and thus tuple 1 subsumes tuple 2. So we keep only the more informative one. Analogously, tuple 5 is subsumed by tuple 3. We further observe that not all representations of a same real-world object are combined. For instance, tuples 1 and 7 are not combined by minimum union, because one stores the address whereas the other tuple stores the blood type. However, all attributes with a NON-NULL value in both tuples have the same value. Hence, these tuples complement each other. Considering tuples 1 and 6, they both store a value for sex, however their values differ. This type of conflict cannot be solved by minimum union or complement union and requires more sophisticated methods.

[1] Duplicate detection may not achieve a perfectly correct result. E.g., tuple 6 may not in fact be a duplicate to 1 or 7, because it describes a female person. Hence, the *oid*s only approximate real-world keys, such as ISBN or SSN.



(a) Relationship between five representations of the 2008 Import CD “Comme si de Rien n’Etait” by Carla Bruni. Dashed arrows signify that the source is subsumed by the target, whereas solid edges represent complementation.



(b) Amazon representations for Result 4 and Result 2

Figure 2: Real world examples of subsuming and complementing representations of audio CD’s at amazon.com

The result of step 2.2 (Fig. 1, bottom right) shows that complementation allows to combine tuples that complement each other, for instance, tuple 1 and 7. Note that subsumed tuples are not removed by complement union, e.g., the complementation of tuple 4 and 5 appears in the result although it is subsumed by the complement of tuples 3 and 4. In our operator based approach, we can apply a subsumption operator on this result to remove subsumed tuples.

Real-world applications for minimum & complement union. Throughout our experience with real-world data sets, including those for which we report experiments in Sec. 6, we have seen the need for the subsumption and complementation operators. Both operations are fairly “risk-free”: only NULL values are eliminated. Also, we can find numerous examples on the web. For instance, as of 9/1/2009, when searching for “Carla Bruni rien” on Amazon.com, we obtain five results for the 2008 Import CD “Comme si de Rien n’Etait” by Carla Bruni. Fig. 2(a) shows a graph that relates the different representations to each other. There, subsumption is represented by dashed arrows (source is subsumed by target) and complementation is represented by solid lines. Note that we labeled the nodes with the ASIN number, an identifier Amazon assigns to its products. Two sample representations, e.g., Result 4 and Result 2 are represented in Fig. 2(b).

Subsumption and complementation both help to obtain a more concise result. However, their results are inherently different, and which operator to use is application dependent. On the one hand, subsumption removes subsumed tuples but does not combine any information that was not already combined in one of the sources. Hence, subsumption can be used in applications where having correct (combined) information is crucial. 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. On the other hand, complementation combines tuples to form new, more complete tuples than the individual input tuples. This behavior is adequate in scenarios where a more complete description is desired and where occasional incorrect combinations have little negative impact on the application. This is for instance true in our disaster management scenario.

Contributions. We regard subsumption and complementation as database primitives. The contributions of this paper are (1) the definition of the novel *complementation* operator; (2) the definition of *complement union*; (3) efficient algorithms for subsumption; and (4) a study of how subsumption can be moved in an operator tree to optimize queries using minimum union. Due to the lack of space, we limit the discussion of algorithms and experiments to subsumption. Algorithms on complementation and transformation rules are described in [4].

Structure. After covering related work in Sec. 2, we formally define the operators in Sec. 3. Subsequently, we present algorithms to implement subsumption in Sec. 4. How subsumption interacts with outer union (for minimum union) and other relational operators in logical query plans is covered in Sec. 5. Sec. 6 evaluates our algorithms and we conclude in Sec. 7.

2. RELATED WORK

In the literature, many different approaches to integrate two or more sources with the help of individual operators already exist, e.g., see [10, 12, 13, 19, 20, 23, 26, 28, 30] to name just a few. We briefly summarize some of them here and refer readers to [3] for a recent survey. Throughout our discussion, we distinguish possible operators for fusion in general, work related to subsumption and work related to complementation.

Operators for Data Fusion. Relational algebra offers some operators that can be used to combine the information of different object representations for data fusion, e.g., *union* and *join*. However, they do not handle uncertainties and conflicts well. Consequently, specially designed operators to accomplish data fusion have been proposed. We discuss the two most relevant.

Match join [30] implements a possible world semantics and, given the conflicting source tables, creates all possible attribute value combinations for a real world object. A designated identifying attribute, e.g., a real world identifier, needs to be declared. Assuming that Name is chosen as id, we obtain the result shown in Fig. 3(a) when applying match join on *Police* and *Hospital*.

Interestingly, we obtain four variants of Miller that essentially correspond to all possible combinations of sex and blood type, regardless of a combination's existence in any of the sources. The other extreme is the single tuple for Peters, where no attribute value is ambiguous.

Full disjunction [6, 7, 12, 23] is an associative extension of the outerjoin operator to an arbitrary number of relations as it returns all combinations of (partially) satisfied join paths and pads the remaining attributes with NULL values. To produce concise results, subsumption is considered when computing the full disjunction. However, the notion of subsumption refers to tuple sets not being subsumed, as opposed to attribute sets not being subsumed (the semantics we consider, as defined in Sec. 3). In the special case where no NULL values exist in the sources [12, 23] both semantics are equivalent. For instance, performing the full disjunction using (Name, DoB, Sex) as join attributes yields to the relation of Fig. 3(b). Here, we observe that the first tuple subsumes the third tuple when defining subsumption based on the containment of attribute value sets, e.g., {Miller, 12 Main} ⊂ {Miller, 7/7/59, m, 12 Main, O}. On the other hand, referring to the definition based on the containment of tuple sets that is in general used for full disjunction [6, 7], we see that the third tuple results from padding tuple 2 (i.e., the tuple with *tid* 2) with NULL values and we can verify that there is no tuple set in the result of full disjunction that combines tuple 2 with any other tuple set. Hence, the tuple set {tuple 2} is not subsumed by any other tuple set.

Name	DOB	Sex	Address	Blood
Miller	7/7/59	m	12 Main	B
Miller	7/7/59	m	12 Main	O
Miller	7/7/59	f	12 Main	B
Miller	7/7/59	f	12 Main	O
Peters	1/1/53	m	34 First	AB

(a) Match join over Name

Name	DOB	Sex	Address	Blood
Miller	7/7/59	m	12 Main	O
Peters	1/1/53	m	34 First	⊥
Miller	⊥	⊥	12 Main	⊥
Peters	1/1/53	⊥	⊥	AB
Miller	⊥	f	⊥	B

(b) Full disjunction

Figure 3: Results of two operators for data fusion on *Police* and *Hospital*

In summary, full disjunction improves on minimum union as it is able to combine complementing tuples from two sources, e.g., the tuples with *tid* 1 and 7 combine to the first tuple of Fig. 3(b). However, it cannot combine complementing tuples from the same source, nor can it combine tuples where overlapping/join attributes contain NULL values. We observe that, as for complementation, the removal of subsumed tuples is not an integral part of the full disjunction operator.

Work related to Subsumption. The *minimum union* operator is defined in [12] and is used in many applications. For instance, minimum union is exploited in query optimization for outer join queries [11]. 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. 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 [14] as one possible semantics to use a schema mapping for the creation of transformation rules [21].

Subsumption is used in [24] 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 (see Exp. 5, Sec. 6).

Another very efficient rewriting for subsumption is proposed in [15]. However, it is also not applicable in general, as special properties of the problem considered in [15] 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 a special case of the problem of minimizing tableau queries [1] by transforming each tuple into a literal $R(v_1, ..., v_m)$ where m is the schema size and the v_j are the attribute values of the tuple. Then, finding a minimal tableau for the tableau created that way is equivalent to removing subsumed tuples. There are algorithms for min-

imizing tableaus, e.g., [25]. However, they do not consider our special case (resulting from our data fusion task), but the general case, resulting in exponential time complexity.

Computing set containment joins, e.g., [17], 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 that are subsumed but also subsume another tuple still remain in the result.

Work related to Complementation. To the best of our knowledge, this work is the first that considers data fusion with the semantics of complement union. 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 [5, 16], although [27] improves on that on dynamic graphs by introducing edge weights. Work described in [18] enumerates all cliques of a minimum size.

3. DEFINITIONS AND NOTATION

We now formally define the operators discussed throughout this paper, i.e., subsumption, complementation, minimum union, and complement union.

We assume that we know which attributes are semantically equivalent. For simplicity, these attributes have the same name in our discussion. Further, both operators assume an "unknown" semantics of NULL values, as they both aim at filling up NULL values with existing data. Finally, we assume set semantics where exact duplicates are removed; extending the operators to bag semantics and other semantics of NULL (such as labeled NULLs as used in data exchange [9]) is deferred to future work.

Let $A = \{a_1, a_2, \dots, a_m\}$ be a set of m attributes that, together with the relation name, defines the *schema* of a relation. A *relation* $R = (A, T)$ of size n is defined by a schema and a set $T = \{t_1, t_2, \dots, t_n\}$ of n tuples. When the schema and the set of tuples are clear from the context, we denote relations with R. A database consists of k relations $R_1, \dots, R_k$. A *relational tuple* t_i is a set of up to m attribute/value combinations, $t_i = \{(a_1, v_1), (a_2, v_2), \dots, (a_m, v_m)\}$. *Values* v_j are of a *domain* $\mathrm{dom}(j)$, such as integers, strings, etc. Missing information is usually modeled by introducing a special value, the NULL value ($\perp$), which is not part of any domain. We model NULL values 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_j. Based on these definitions, we formally define the relevant operators. Note that examples on how these operators behave are given in Fig. 1.

DEFINITION 1 (TUPLE SUBSUMPTION [12]). *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.* □

When modeled as above, tuple subsumption is equal to set containment. A tuple t_1 *subsumes* another tuple 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 2 (SUBSUMPTION OPERATOR). *We use the unary subsumption operator β to denote the removal of subsumed tuples from a relation R: $\beta\ (R) = \{t \in R : \neg\exists t' \in R : t' \sqsupset t\}$* □

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. Under bag semantics, the distinct operator (δ) can additionally be used to remove exact duplicates.

DEFINITION 3 (OUTER UNION). *The* outer union *operator, $\uplus$, combines two (or more) relations $R_1 \uplus R_2$ by first constructing the schema of the outer union as the set union of the schemata of the source relations ($S = S_1 \cup S_2$) and secondly by combining the two tuple sets to form one single set ($T = T_1 \cup T_2$). Missing tuple values are padded with $\perp$.* □

DEFINITION 4 (MINIMUM UNION). *The* 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, i.e., $A \oplus B = \beta(A \uplus B)$.* □

Note that the *minimum union* operator is commutative and associative. Also note that the definition removes subsumed tuples both from the same source (e.g., tuples with *tid* 1 and 2 from our example in Fig. 1) and from different sources (e.g., tuples with *tid* 3 and 5).

DEFINITION 5 (TUPLE COMPLEMENTATION). *A tuple t_1 complements a tuple t_2 ($t_1 \gtrless t_2$) if (1) t_1 and t_2 have the same schema, (2) values of corresponding attribute in t_1 and t_2 are either 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.* □

When modeled as above, two tuples t_1 and t_2 complement each other, if for the set $C = t_1 \cup t_2$ of all (a_j, v_j) pairs from both tuples it holds that $\forall (a_l, v_l), (a_m, v_m) \in C : a_l = a_m \rightarrow l = m \wedge C \supset t_1 \wedge C \supset t_2$ (assuring at most one value per attribute, or a NULL value). Set C is called the complement. 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. 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 6 (MAXIMAL COMPLEMENTING SET). *A maximal complementing set MCS of a relation R is a set of tuples from T where for each pair t_i, t_j of tuples from MCS it holds that $t_i \gtrless t_j$ and for every $t_k \notin MCS$ there is at least one $t_i \in MCS$ that does not complement t_k.* □

A single tuple $t_i \in T$ that does not complement any other tuple $t_j \in T$ forms a maximal complementing set of size one. A t_i can be part of more than one MCS and there may be multiple MCS for a relation R. All tuples contained in a maximal complementing set can be combined into one tuple, the complement.

DEFINITION 7 (COMPLEMENTATION OPERATOR). *We define the unary complementation operator κ to denote the replacement of all existing maximal complementing sets in a relation R by the complement of the contained tuples.* □

DEFINITION 8 (COMPLEMENT UNION). *The* complement union *operator, $\boxplus$, is the combination of* outer union *and* complementation*, where complementary tuples in the result of the* outer union *of the two input relations are replaced by the complement of the two tuples, i.e., $A \boxplus B = \kappa(A \uplus B)$.* □

Complement union is commutative, but not associative. Similar to subsumption, the definition forms complements of complementary tuples from the same source (e.g., tuples with *tid* 4 and 5) and from different sources (e.g., tuples with *tid* 1 and 7).

4. IMPLEMENTING SUBSUMPTION

After having defined the necessary concepts, we now present different algorithms for computing subsumption. To implement minimum union, we simply combine subsumption with outer union, which we further discuss in Sec. 5. We also devised algorithms for computing complementation [4], which are similar in spirit to those presented for subsumption. Due to the lack of space, we do not discuss these in detail here.

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 possible directions of speeding it up: First, we can stop pairwise comparisons of a given tuple to other tuples as soon as we have found a tuple that subsumes the one at hand. Second, we can start grouping tuples along the process in such a way that two tuples are in the same group, if one subsumes the other. In the remainder of this paper, we refer to this improved naive algorithm as the Simple algorithm. Its worst-case time complexity is still $\mathcal{O}(n^2)$ with n being the number of tuples in R.

In the following, we first present a partitioning technique (in the spirit of hash-based duplicate elimination [29]) that can be applied in addition to the Simple or any other subsumption algorithm to potentially further save runtime (Partitioning algorithm). We then present an alternative algorithm for computing subsumption, namely the Null-pattern-based algorithm, that potentially improves on the worst-case time complexity of the Simple algorithm.

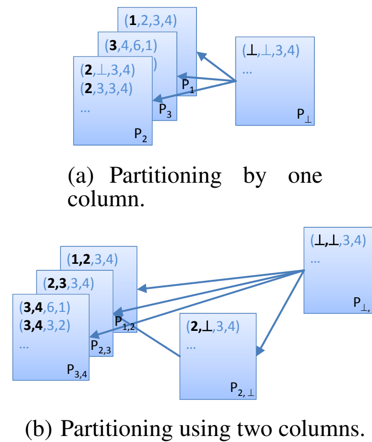
4.1 Input Partitioning

To improve the runtime for computing subsumption, we first propose to partition the input 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 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 the Null-pattern-based algorithm.

Essentially, we select a partitioning criteria, 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 c). Eventually, we also get an additional partition containing all tuples whose c-value is NULL and which we denote as the NULL partition $P_\perp$.

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 separately. The NULL partition $P_\perp$, in addition to have 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 Fig. 4(a).

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 algorithm $\mathcal{A}$ (e. g., in case of the Simple algorithm, by applying pairwise comparisons to the tuples in $P_\perp$). It also reads each NON-NULL partition P_i and removes all subsumed tuples within it by applying $\mathcal{A}$ (in case of the Simple algorithm, each tuple t in P_i is compared to the tuples in $P_i \backslash \{t\}$). After removing all subsumed tuples from P_i, the remaining tuples constitute $\beta(P_i)$ and are part of the output. Third, the algorithm has to check if there are tuples in the remaining NULL partition $\beta(P_\perp)$ that are subsumed by tuples in $\beta(P_1), \ldots, \beta(P_d)$. Therefore, each tuple t in $\beta(P_\perp)$ is compared to the tuples in the remaining NON-NULL partitions $\beta(P_1), \ldots, \beta(P_d)$ and, in case t is subsumed, it is removed from $\beta(P_\perp)$. Hence, after considering all tuples in $\beta(P_\perp)$, the remaining tuples in it are part of the output.



(a) Partitioning by one column.

(b) Partitioning using two columns.

Figure 4: Example partitioning by one or two columns and comparison schema given by the arrows

If we choose more than just one column to partition the input, we get 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 column value in P' is arbitrary.

An example for a partitioning by two columns is shown in Fig. 4(b). According to the given conditions, tuples from the complete ($P_{\perp,\perp}$) or the partial NULL partitions (e.g., $P_{2,\perp}$) need be compared only 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 algorithm $\mathcal{A}$ used to compute the subsumption operator β can be implemented in different ways. In our experiments we combine Partitioning both with the Simple and the Null-pattern-based algorithm, which is described in the next section.

The overall runtime of the Partitioning method depends both on the runtime of the algorithm $\mathcal{A}$ 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 argumentation holds. For an input relation of n tuples, let $T_{\mathcal{A}}(n)$ be the runtime of $\mathcal{A}$. Then, the overall runtime complexity of the Partitioning method when having d NON-NULL partitions is bounded

by

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

where $|P_i|$ is the number of tuples in partition P_i. Using the Simple 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 gets 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 over all partitions, 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 to select c. Experiments in Sec. 6 confirm this analysis.

We propose a heuristic approach that chooses a single partitioning column c based on the following rules applied in that order: 1) Choose a key column, 2) Choose a column that does not contain NULL values, 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.

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 stays the same when increasing the number of columns. At the same time, the average size of each partition decreases. We show in the experiments section that it is generally beneficial to choose as many columns (maximum m columns) as possible for partitioning.

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, e.g., if we can save the time for partitioning, i.e., if we can access tuples in order, for example by using an already 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. We show in the experiments section that we can already reach a pretty 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 algorithm as $\mathcal{A}$ and one column for partitioning. Second, we choose the k smallest columns with respect to D_c. The needed counts can easily be obtained from standard database statistics.

Algorithm 1 Null-pattern-based Algorithm

```
 1: for all tuple t ∈ T do
 2:     Compute the bit pattern (b_1, ..., b_m) w.r.t. t;
 3:     Insert t into its corresponding bucket B_(b_1,...,b_m);
 4: end for
 5: Let k be the minimal number of NULL values within any tuple;
 6: Output all tuples with k NULL values;
 7: for all i = k + 1, ..., m do
 8:     for all non-empty buckets B_j' with exactly i NULL values do
 9:         Collect all buckets B_j where j is constructed out of j' by setting
            at least one 0-bit (all direct and indirect parent buckets in Fig. 5);
10:         Generate π(B_j') (cf. Equ. 1) and sort all tuples in it in lexico-
            graphical order;
11:         for all tuples t' in B_j' do
12:             Generate π(t') out of t' (cf. Equ. 1) and test if π(t') is sub-
                sumed by a tuple in π(B_j') using binary search;
13:             if π(t') is subsumed by a tuple in π(B_j') then
14:                 Remove t' from B_j';
15:             else
16:                 Output t';
17:             end if
18:         end for
19:     end for
20: end for
```

4.2 Null-pattern-based algorithm

We now present the Null-pattern-based algorithm. The main idea of this algorithm is to use information about NULL values in the tuples. Simply by looking at patterns of NULL values, we can exclude tuples from the set of possibly subsuming tuples and avoid many tuple comparisons.

The Null-pattern-based algorithm, which is described in Algorithm 1, works in two steps. In the first step (lines 1 – 4) each tuple is inserted into a bucket according to its existing NULL values. Let $t = \{(a_1, v_1) \ldots, (a_m, v_m)\}$ be a tuple of the input relation, then t is inserted into the bucket $B_{(b_1,\ldots,b_m)}$, for which

$$\forall i \in \{1,\ldots,m\} : b_i = \begin{cases} 0 & \text{if } v_i \text{ is NULL} \\ 1 & \text{otherwise.} \end{cases}$$

At the end of this step each bucket contains all tuples of the input relation that have the same null-value pattern. Tab. 1 shows the buckets to which tuples in our example of Fig. 1 are assigned.

id	Name	DOB	Sex	Address	Blood	*Bucket*
1	Miller	7/7/59	m	12 Main	⊥	*11110*
2	Miller	⊥	⊥	12 Main	⊥	*10010*
3	Peters	1/1/53	m	34 First	⊥	*11110*
4	Peters	1/1/53	⊥	⊥	AB	*11001*
5	Peters	1/1/53	m	⊥	⊥	*11100*
6	Miller	⊥	f	⊥	B	*10101*
7	Miller	7/7/59	m	⊥	O	*11101*

Table 1: Bucket assignment in example scenario

In the second step (lines 5 – 20) we compare tuples to determine if one subsumes the other. To improve efficiency, we prune the set of buckets under consideration based on the following conditions that directly follow from the definition of subsumption: For a tuple t' to be subsumed by a tuple t, it is necessary that

1. the number of NULL values in t is smaller than the number of NULL values in t';

2. t and t' coincide in every column in which t has a NULL value, i. e. for each column where t has a NULL value, the same holds for t'

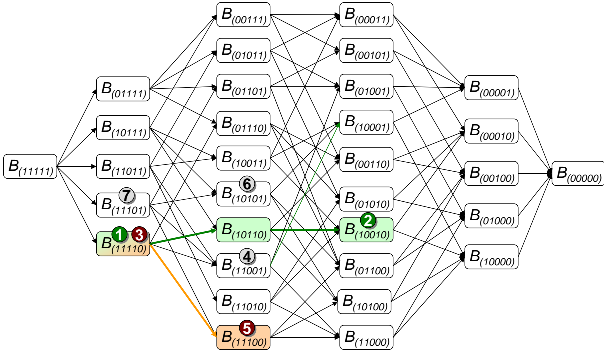


Figure 5: Dependency between buckets and subsuming tuples

From the first condition it directly follows that all tuples in the buckets with the least number of NULL values cannot be subsumed by any other tuple in R. Hence, these tuples can be output to the result without further computation. In our example, these correspond to the tuples 1, 3, and 7, which each have only one NULL value.

It further follows that tuples in a bucket B_j with i NULL values can potentially subsume only tuples in a bucket $B_{j'}$ with at least $i + 1$ NULL values, where all zero bits in j are also zero bits in j' and where at least one 1-bit in the bit pattern j is unset in the bit pattern j'. In the graph depicted in Fig. 5 nodes represent buckets and edges represent the relationship of unsetting one bit from the parent (left) to child node (right). We added the tuples of our example (represented as circles) to their respective buckets, and we highlight those paths where subsumption occurs. 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. We proceed incrementally and first consider all buckets with i NULL values before considering buckets with $i + 1$ NULL values (i. e., the buckets shown in Fig. 5 are processed from top to bottom and left to right). As soon as we determine that one tuple in $B_{j'}$ is subsumed using the method described in the next paragraph, we delete it from its bucket. Our traversal through the buckets can be viewed as a breadth-first search starting from the bucket $B_{(1\ldots1)}$, as we first consider all buckets with i NULL values before considering buckets with $i + 1$ NULL values.

To efficiently determine if a tuple t' in $B_{j'}$ is subsumed by a tuple t in B_j, we can do better than performing a linear search. Essentially, we perform a projection on t' as well as on all tuples in parent buckets in our graph representation, i.e., buckets that contain tuples that potentially subsume t'. Formally, let

$$\pi(t') := \pi_x(t') \quad \text{and} \quad \pi(B_{j'}) := \bigcup_{B_j} \{\pi_x(t) : t \in B_j\} \quad (1)$$

where x is the sequence of all attributes for which all tuples in $B_{j'}$ (and hence also t') do not have NULL values. Hence, $\pi(t')$ consists of all NON-NULL values of t'. Note, that the set $\pi(B_{j'})$ does not contain any NULL values. Therefore, it is possible to sort the tuples in $\pi(B_{j'})$ in lexicographical order. Afterwards, for each tuple t' in $B_{j'}$ we can decide within $\mathcal{O}(\log |\pi(B_{j'})|)$ time if t' is subsumed using the binary search method. If a tuple $t' \in B_{j'}$ is subsumed, we remove t' from $B_{j'}$. Once all tuples within $B_{j'}$ have been processed in the described way, we add the remaining tuples in $B_{j'}$ to the output, as it is guaranteed that these are not subsumed by tuples in yet unprocessed buckets.

As an example, consider tuple 5 in Fig. 5, which lies in the bucket $B_{(11100)}$. The parent buckets are $B_{(11110)}$ and $B_{(11101)}$, which contain tuples 1, 3, and 7. After applying the projection to 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 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 Fig. 5). 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 factor can become substantial, but in our experience for 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 algorithm of $\mathcal{O}(n \log n)$. In cases m does not satisfy this condition, the overall runtime is given by

$$\begin{array}{ll} \mathcal{O}(n^2) & \text{, if } m \in \mathcal{O}(\log(\frac{n}{\log n})) \\ \mathcal{O}(n^2 \log n) & \text{, otherwise} \end{array}$$

The first result can be obtained by substituting m in the overall runtime complexity given above:

$$\mathcal{O}(2^{\mathcal{O}(\log(n/\log n))} n \log n) = \mathcal{O}(\frac{n}{\log n} n \log n) = \mathcal{O}(n^2).$$

Therefore, as long as $m \in o(\log(\frac{n}{\log n}))$ the asymptotic runtime of the Null-pattern-based algorithm is smaller than that of the Simple algorithm.

The second result arises from the fact, that at most n of the 2^m buckets can be filled. Therefore, the number of iterations of the second step is bound by n, which results in an overall runtime of $\mathcal{O}(n \cdot (n \log n))$.

Therefore, we conclude that the worst-case time complexity of the Null-pattern-based algorithm is given by $\mathcal{O}(\min\{2^m, n\} n \log n)$. Especially for the common case that the number m of attributes is a constant, the time complexity of the Null-pattern-based algorithm is $\mathcal{O}(n \log n)$.

4.3 Comparison

Concluding the presentation of the two algorithms in the previous sections we briefly summarize their similarities and differences. Using the Partitioning algorithm as a starting point, the Null-pattern-based algorithm is a further extension of a Partitioning algorithm version that uses all m columns to partition the input relation. More specifically, this is done by combining 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 one bucket, and adding two additional features: the more intelligent processing order of the complete and partial NULL partitions and the binary search within the lexicographically sorted projections of the buckets.

However, as the Null-pattern-based 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 variable 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 algorithm on its own, b) the Partitioning algorithm in combination with the Simple algorithm, c) the Partitioning algorithm in combination with the Null-pattern-based algorithm, and d) the Null-pattern-based algorithm on its own. In the experiment Section (Sec. 6) we evaluate and compare all four possibilities.

5. SUBSUMPTION IN QUERY PLANS

So far we considered efficient implementations of the subsumption operator. We now focus on optimization at the logical level, i.e., how operators can be moved in a query plan to increase efficiency. More specifically, we study how subsumption can be moved in combination with other common relational operators. Tab. 2 shows the rules that are subsequently described in more detail.

Combinations with Outer Union	
(1)	β (A$\uplus$ B) $=\beta$ (A) $\uplus$ β (B), if there are no subsumptions across sources, and
(2)	β (A$\uplus$ B) $=\beta_1$ (β (A) $\uplus$ β (B)), in all other cases.
Combinations with Selection	
(3)	β (σ_c (A)) $\neq \sigma_c$ (β (A)), if c is of the following form: x IS NULL, with x being an attribute with NULL values, and
(4)	β (σ_c (A)) $=\sigma_c$ (β (A)), in all other cases
Combinations with Join	
(5)	β (A$\times$B) $=\beta$ (A) $\times\beta$ (B)
(6)	β (A$\bowtie$B) $=\beta$ (A) $\bowtie\beta$ (B), if selection can be pushed down
Combinations with Grouping and Aggregation	
(7)	$\lambda_{f(c)}$ (β (A)) $=\lambda_{f(c)}$ (A), for any column c and aggregation function $f \in \{\max, \min\}$
(8)	β ($\lambda_{f(c)}$ (A)) $=\lambda_{f(c)}$ (A), for any column c and any f
(9)	$\lambda_{c,f(d)}$ (β (A)) $=\lambda_{c,f(d)}$ (A), for any different columns c, d with c being the grouping column and not containing NULL values and aggregation function $f \in \{\max, \min\}$
(10)	β ($\lambda_{c,f(d)}$ (A)) $=\lambda_{c,f(d)}$ (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	
(11)	β (β (A)) $=\beta$ (A)

Table 2: Rewrite rules involving subsumption

Combinations with (Outer) Unions. Minimum union is defined as the combination of outer union with the subsequent removal of subsumed tuples (see Sec. 3). Therefore, we first examine the exchangeability of union and subsumption in operator trees.

As there may be subsuming tuples across sources, simply pushing subsumption through union is not possible without leaving an additional subsumption operation on top of the outer union (Rules (1) and (2) from Tab. 2). We use β_1 for the subsumption on top of the outer union to denote that in that place there does not need to be a full version of subsumption as we need to test for subsumed tuples only across sources. To compute β_1, let P_1 be the set of attributes and values private to $t_1 \in A$. Let P_2 analogously be the set of private attributes and values to $t_2 \in B$, 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 and t_2 from two sources to subsume one another, the following condition needs to hold in order for t_1 to be subsumed by t_2: $P_1 = \emptyset \wedge (C_1 \subset C_2 \vee (C_1 = C_2 \wedge P_2 = \emptyset))$. We use this condition in implementing β_1 and it could also be used in implementing minimum union as an independent operator.

Combinations with Projection. Potentially all attributes are needed to decide if one tuple subsumes another. A projection can be inserted, if it projects out columns that do not affect the removal of subsumed tuples (e.g., a column that contains only one single value, or a column that has the same value for all pairs of subsuming tuples). Unfortunately, testing these properties has the same complexity as performing the subsumption itself.

Combinations with Selection. As subsumption 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 through the subsumption operator (Rule 4).

If a column allows NULL values, pushing selection through subsumption alters the result only if a tuple subsuming another tuple is removed from the input of subsumption. We distinguish the following cases:

Case 1: Selections involving a column and a constant value v, e.g., $a_i < v$, $a_i = v$, $a_i \neq v$. 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 (Rule 4).

Case 2: Selections that test for NULL equality, i.e., a_i IS NULL. In this case, only the subsumed tuple passes the selection. Therefore we cannot exchange β and σ in this case (Rule 3).

Case 3: Selections that test for NULL inequality, i.e., a_i 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 (Rule 4).

Case 4: Selections involving two columns, i.e., $a_i < a_j$, $a_i = a_j$, $a_i \neq a_j$. Generally, these selections can be pushed through subsumption as in the cases with only one column (Rule 4).

Combinations with Join. When exchanging β and the Cartesian product $\times$ we must ensure that when applying $\times$ (i) no additional subsuming tuples are introduced if there have been none in the base relations, and (ii) all subsuming tuples in the base relations still subsume one another after applying $\times$. The former follows from the definition of subsumption: if two tuples do not subsume one another, no extension of the schema changes that fact. The latter follows from the fact that by Cartesian product, two subsuming tuples from one base relation are necessarily combined with the same tuples from the other base relation resulting in the two resulting tuples still being subsumed. When combining cross product and subsumption, Rule 5 holds.

When exchanging β and $\bowtie$ in the query plan, we need to apply the rules involving selection. Although those rules are not applicable for all possible join conditions, the most often occurring joins (key and foreign key joins) can be transformed (Rule 6).

Combinations with Grouping and Aggregation. In general, subsumption and grouping/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 an aggregation alone, the subsumption 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 (Rule 7).

dataset	size	%subsumed	#columns	%nulls
Gen1	10k - 5M	1, 5, 10	6	5
Gen2	10k - 5M	1, 5, 10	6, 20, 40	40
CDDB	$\approx$ 10k	0.1 – 0.4	6 – 7	21 – 24
Actors	$\approx$ 3.6M	0.5	12	53
Movies	$\approx$ 500k	0.04	27	81
CRM	$\leq$ 1M	0.03 – 0.8	5 – 8	13 – 40

Table 3: Dataset summary

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 (Rule 8).

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, such as min and max (see Case 1), and additionally if the grouping column does not contain NULL values (Rules 9 and 10). The latter condition is necessary, because NULL values in the grouping column are combined in a separate group. Tuples belonging to this group may be removed by subsumption resulting in a different aggregation for this group. Also, the group itself may be removed by subsumption after grouping/aggregation.

Other Combinations. Subsumption (and our implementations thereof) is not order-preserving. Therefore it does not make sense to exchange sorting and subsumption operators. Two subsumption operators can be combined into one (Rule 11).

6. EXPERIMENTS

For subsumption, we implemented the Null-pattern-based algorithm from Sec. 4.2 as well as two partitioning variants: one uses the Null-pattern-based algorithm as subroutine to perform subsumption whereas the other variant uses the Simple algorithm. We denote the variants as Partitioning (Null-pattern-based) and Partitioning (Simple), respectively. For comparison, we also implemented the Simple 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 applicable if a favorable ordering exists [24]. We focus on subsumption alone as the extension to minimum union does not add significant complexity and is also the same for all subsumption algorithms.

6.1 Data and Setting

We experimented both with synthetic and real world data. Tab. 3 provides a summary of the datasets subsequently used.

To evaluate our algorithms based on different characteristics, we implemented a data generator that allows to generate datasets of various sizes with a controllable amount of subsumed 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 key, 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 in order to test for scalability. We further varied the percentage of subsumed tuples from 1% to 5% to 10% for different amounts of overall NULL values (5% and 40%). We also varied the number of columns (6, 20, or 40 columns). The generated datasets are summarized in rows labeled Gen1 and Gen2 in Tab. 3.

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. The third dataset is from a CRM domain with an industry partner, whose identity cannot be disclosed due to privacy issues. The real-world datasets mainly store strings and float.

A 2.3GHz dual processor quad core server with 16GB of RAM was used to store the data in an IBM DB2 v9.5 DBMS and to perform the experiments that study the runtime of different algorithms on varying datasets. All our algorithms are implemented in Java 1.6 and the reported runtimes are median values over 5 runs.

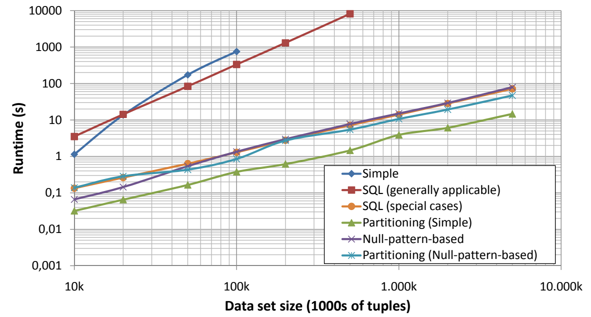


Figure 6: Comparison of subsumption algorithms on Gen2 (5% subsumed tuples)

6.2 Results on Subsumption

Experiment 1: Algorithm Comparison. The goal of this experiment is to compare different implementations of the subsumption operator in terms of runtime.

Methodology. We consider six different implementations of subsumption: Simple (the naive algorithm) and the generally applicable SQL statement serve as baseline algorithms that are either trivial or known from the literature. The data set also allows to use the SQL rewriting for special cases as described in [24] as a favorable ordering exists. The remaining three implementations correspond to the different variants presented in this paper, namely the Null-pattern-based algorithm, Partitioning (Simple), and Partitioning (Null-pattern-based). Fig. 6 shows the runtime (in seconds) obtained on dataset Gen2 when varying the size between 10,000 and 5 million and setting the percentage of subsumed tuples to 5%. The heuristic given in Sec. 4.1 is used to choose the partitioning column. Here, it is the column with the real-world identifier, resulting in many very small partitions.

Discussion. The algorithms presented in this paper clearly outperform both the Simple algorithm and the generally applicable SQL statement, and are 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 and our algorithms is already two orders of magnitude. The SQL rewriting is applicable for this dataset and performs comparable to our algorithms.

We also make the following observations: For all but small data set sizes, Partitioning 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).

[2]FreeDB: www.freedb.org

[3]IMDB: www.imdb.com

[4]Data kindly provided by Filmdienst: film-dienst.kim-info.de

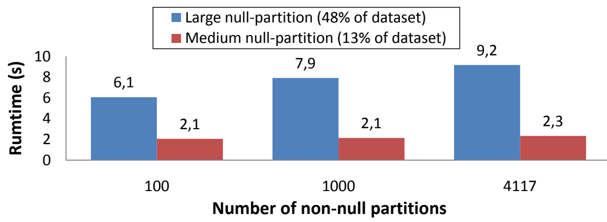


Figure 7: Comparing runtime of Partitioning (Simple) when using different columns for partitioning for two datasets

Experiment 2: Partitioning. In the previous experiment, Partitioning(Simple) performs best, but as this experiment demonstrates, its runtime highly depends on the size of the NULL partition and, to a less extent, on the number of partitions.

Methodology. We constructed two Gen2 datasets with 5% of subsumed tuples and 20,000 tuples. The datasets differ in the numbers of distinct and NULL values per column and therefore in the size of the NULL partition. Choosing different columns for partitioning results in different numbers of partitions. Fig. 7 shows the runtimes.

Discussion. We see that a poor column choice significantly increases the runtime of Partitioning (Simple) and using the Null-pattern-based algorithm in this case would be the better choice. Indeed, the first observation is that if the partitioning column has a large percentage of NULL values, the runtime is significantly higher than when choosing a column that yields a small NULL partition. The reason for this behavior is that every tuple in the NULL partition is compared with every other tuple in every other NON-NULL partition. This behavior supports our theoretical considerations in Sec. 4. In further experiments, e.g., the one reported in Fig. 12 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: Influence of 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 algorithm. 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 data set 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 Fig. 8 and results for the latter case in Fig. 9.

Discussion. In Fig. 8, we see that all three 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 Partitioning (Null-pattern). In Fig. 9, 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 Null-pattern-based algorithm. We explain this behavior with the fact that with a smaller number of NULL values, the distribution of tuples in the bucket structure (see Fig. 5) changes in such a way that more tuples are in the leftmost buckets, resulting in more overall comparisons needed in order to decide for a specific tuple if it is subsumed.

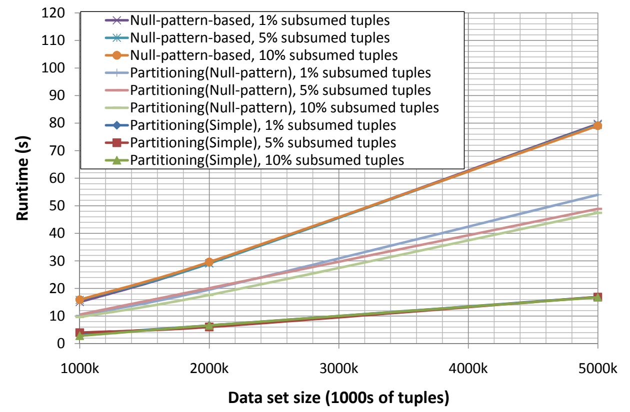


Figure 8: Comparing the influence of different percentages of subsumed tuples on the algorithms

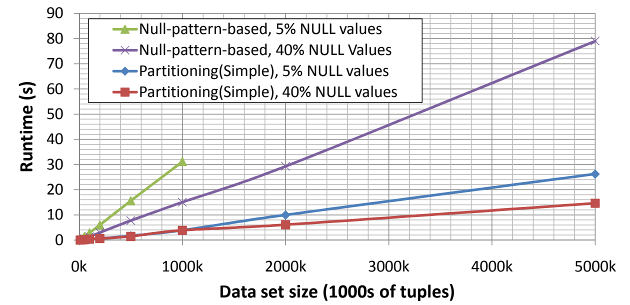


Figure 9: Runtime of Partitioning (Simple) and NPB-algorithm on Gen1 and Gen2 with 5% tuples being subsumed

Experiment 4: 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 Partition (Simple) usually performs best, so here we consider only this algorithm. We generate versions of data set Gen2 of different sizes and with 6, 20, or 40 attributes per relation. Runtime results are shown in Fig. 10.

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 accept-

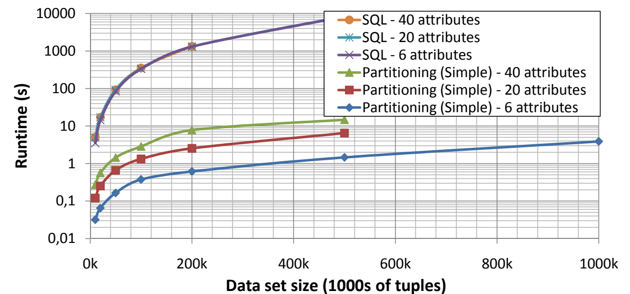


Figure 10: Comparing runtimes for different schema sizes for Partitioning (Simple) on Gen2 dataset

able: Tables with 40 attributes and 50,000 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 5: Subsumption on real-world data. The previous results for subsumption were obtained on generated datasets and we now evaluate our algorithms for subsumption on the different real-world datasets summarized in Sec. 6.1.

Methodology. We consider the real-world datasets Actor, Movie, CDDB, and CRM. Within the CRM database, we analyze three tables (similarly to the Movie database 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 Partitioning (Simple). Fig. 11 reports the results for the Actor and the CRM datasets. The results on the Movie 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 Fig. 12.

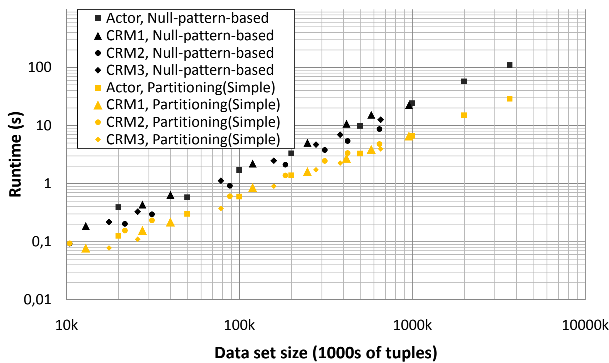


Figure 11: Subsumption runtime on real-world data

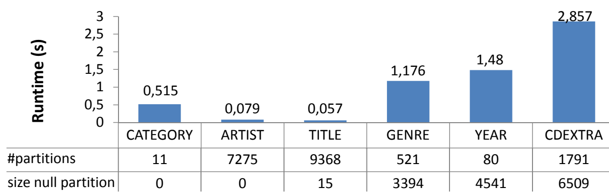


Figure 12: Runtime for different partitioning columns in CDDB

Discussion. In Fig. 11 as well as on the Movie 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. Both algorithms exhibit a nearly linear increase in runtime with respect to the increasing dataset size. The datasets considered in Fig. 11 all have a small number of attributes (i.e., at most 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 Movie dataset with 27 attributes is significantly higher than on the other datasets. This is in accord with our theoretical analysis in Sec. 4.1. As a final observation, we see that 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 due to the longer comparisons needed on string data present in real-world data compared to integer comparisons performed on the generated data.

Whereas the technique from [24] can be applied to the CDDB dataset, it cannot to the Actors dataset, as a necessary favorable order does not exist in this particular dataset.

Fig. 12 confirms the observations of Experiment 2 as it shows that when partitioning 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.

As a rule of thumb, whenever there is a key (or a pseudo-key) in a relation (e.g., TITLE & ARTIST in dataset CDDB) we are better off using the Partitioning (Simple) algorithm. If there is no such key or pseudo-key, the Null-pattern-based algorithm performs better.

Experiment 6: 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.

Methodology. We use the real world CDDB dataset for our experiments with more than one attribute when partitioning. We approximate runtime by computing the theoretical runtime given by the formula given in Sec. 4.1 and show it for all combinations of up to $k = 6$ attributes (Fig. 13) for the CDDB dataset. We also find the best combination of k attributes and compare it to the combination found by the greedy heuristics from Sec. 4.1 and the average over all possible solutions (Fig. 14).

Discussion. As already expected, we see in Fig. 13 that the approximated runtime differs much when comparing different possible partitions using k attributes. This underscores 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. Fig. 14 compares the approximate runtimes of the best solution, the average runtime of all possible solution 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 operations whereas the best solution is the partitioning by (ARTIST, TITLE, CATEGORY) with estimated 499 operations. This results in the greedy solution being 1.39 times the best solution. Although the greedy algorithms 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 Fig. 13 this is always among the best solutions

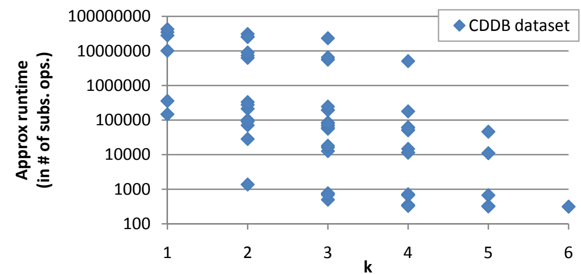
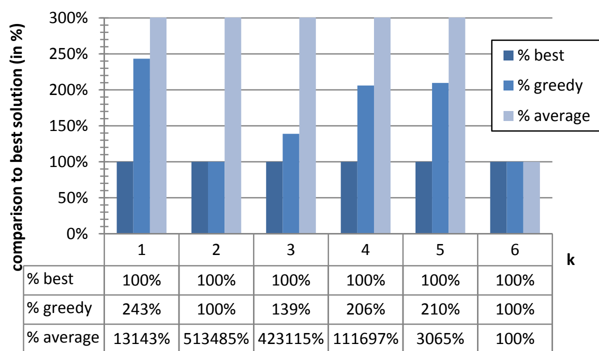


Figure 13: Approximated runtime in number of subsumption operations needed for all combinations of k attributes, k varying from 1 to 6 for the CDDB dataset



	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 14: Experiment on selecting partitioning attributes for the CDDB dataset (y-axis truncated at 300% for readability)

and by magnitudes better than the average solution combination.

7. CONCLUSION AND OUTLOOK

We addressed the problem of combining multiple tuples to a more concise representation in the context of data integration and considered the problem of combining tuples based on subsumption and complementation. Whereas the first operator is well-known, the second defines a novel semantics for combining tuples. When applying subsumption and complementation on top of the outer union operator, we obtain the well-known minimum union and the novel complement union operator, respectively. We presented several algorithms as physical implementations of the subsumption operator. Experiments showed that for subsumption, using a simple input partitioning is the most efficient one, as long as a good partitioning can be found, for instance using a real-world identifier. We also observed that runtime improves with increasing ratio of NULL values in the data. In general, choosing a partitioning attribute with few NULL values is preferable. We also examined partitioning by more than one column and showed that a simple greedy heuristic helps in choosing a good partitioning. We further presented how the subsumption operator can be moved within a query plan using transformation rules. Such transformations can improve query execution time by reducing the input cardinality of the operators.

Future research directions on the operators themselves include extending our work to other NULL semantics (e.g., labeled NULLs) and bag semantics for the complementation operator. We also plan to evaluate the inclusion of the removal of equal tuples into subsumption, and the removal of subsumed tuples into complement. The latter would be a unified operator for which new algorithms must be found. Concerning query optimization, we are considering transformation rules for both operators in combination with advanced aggregation functions, e.g., those proposed for conflict resolution in combination with minimum union [2]. Some potential improvements for the algorithms themselves remain: a partitioning scheme as used for set containment joins may further improve runtime. Finally, we have not yet explored the potential of bitmap indexes for NULL values.

Acknowledgment. This research was partly supported by the German Research Society (DFG grant no. NA 432).

8. REFERENCES

[1] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[2] J. Bleiholder and F. Naumann. Declarative data fusion – syntax, semantics, and implementation. In *Proc. of ADBIS*, pages 58–73, 2005.

[3] J. Bleiholder and F. Naumann. Data fusion. *ACM CSUR*, 41(1):1–41, 2008.

[4] J. Bleiholder, S. Szott, M. Herschel, and F. Naumann. Complement union for data integration. In *Proc. of NTII*, 2010.

[5] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[6] S. Cohen, I. Fadida, Y. Kanza, B. Kimelfeld, and Y. Sagiv. Full disjunctions: Polynomial-delay iterators in action. In *Proc. of VLDB*, pages 739–750, 2006.

[7] S. Cohen and Y. Sagiv. An incremental algorithm for computing ranked full disjunctions. In *Proc. of PODS*, pages 98–107, New York, NY, USA, 2005. ACM Press.

[8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[9] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *Proc. of ICDT*, pages 207–224, 2003.

[10] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: efficient management of inconsistent databases. In *Proc. of SIGMOD*, pages 155–166, New York, NY, USA, 2005. ACM Press.

[11] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *Trans. on Dat. Syst.*, 22(1):43–74, 1997.

[12] C. A. Galindo-Legaria. Outerjoins as disjunctions. In *Proc. of SIGMOD*, pages 348–358. ACM Press, 1994.

[13] S. Greco, L. Pontieri, and E. Zumpano. 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, 2001.

[14] M. A. Hernández, L. Popa, Y. Velegrakis, R. J. Miller, F. Naumann, and C.-T. Ho. Mapping XML and relational schemas with Clio. In *Proc. of ICDE*, pages 498–499, 2002.

[15] P.-Å. Larson and J. Zhou. View matching for outer-join views. In *Proc. of VLDB*, pages 445–456, 2005.

[16] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *Proc. of SWAT*, pages 260–272, 2004.

[17] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *Trans. on Dat. Syst.*, 28(1):56–99, 2003.

[18] N. Modani and K. Dey. Large maximal cliques enumeration in sparse graphs. In *CIKM*, pages 1377–1378, 2008.

[19] A. Motro, P. Anokhin, and A. C. Acar. Utility-based resolution of data inconsistencies. In *Proc. of IQIS Workshop*, pages 35–43. ACM Press, 2004.

[20] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. of VLDB*, pages 413–424. Morgan Kaufmann Publishers Inc., 1996.

[21] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *Proc. of VLDB*, Hong Kong, 2002.

[22] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[23] A. Rajaraman and J. D. Ullman. Integrating information by outerjoins and full disjunctions (extended abstract). In *Proc. of PODS*, pages 238–248. ACM Press, 1996.

[24] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *Proc. of SIGMOD*, pages 671–682. ACM Press, 2004.

[25] Y. Sagiv. Quadratic algorithms for minimizing joins in restricted relational expressions. *SIAM J. Comput.*, 12(2):316–328, 1983.

[26] E. Schallehn, K.-U. Sattler, and G. Saake. Efficient similarity-based operations for data integration. *Data Knowl. Eng.*, 48(3):361–387, 2004.

[27] V. Stix. Finding all maximal cliques in dynamic graphs. *Comput. Optim. Appl.*, 27(2):173–186, 2004.

[28] V. S. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu, A. Rajput, T. Rogers, R. Ross, and C. Ward. Hermes: A heterogeneous reasoning and mediator system. Technical report, University of Maryland, 1995.

[29] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[30] L. L. Yan and M. T. Özsu. Conflict tolerant queries in AURORA. In *Proc. of CoopIS*, page 279. IEEE Computer Society, 1999.