Eliminating NULLs with Subsumption and Complementation

Jens Bleiholder
Opitz Consulting Berlin GmbH, Berlin, Germany
jens.bleiholder@opitz-consulting.com

Melanie Herschel
University of Tübingen, Germany
melanie.herschel@uni-tuebingen.de

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

Abstract

*In a data integration process, an important step after schema matching and duplicate detection is data fusion. It is concerned with the combination or merging of different representations of one real-world object into a single, consistent representation. In order to solve potential data conflicts, many different conflict resolution strategies can be applied. In particular, some representations might contain missing values (*NULL*-values) where others provide a non-*NULL*-value. A common strategy to handle such *NULL*-values, is to replace them with the existing values from other representations. Thus, the conciseness of the representation is increased without losing information.*

Two examples for relational operators that implement such a strategy are minimum union *and* complement union *and their unary building blocks* subsumption *and* complementation*. In this paper, we define and motivate the use of these operators in data integration, consider them as database primitives, and show how to perform optimization of query plans in presence of subsumption and complementation with rule-based plan transformations.*

1 Data Fusion as Part of Data Integration

Data integration can be seen as a three-step process consisting of *schema matching*, *duplicate detection* and *data fusion*. Schema matching is concerned with the resolution of schematic conflicts, for instance through schema matching and schema mapping techniques. Next, duplicate detection is concerned with resolving conflicts at object level, in particular detecting two (or more!) representations of same real-world objects, called duplicates. For instance, considering two data sources describing persons, schema matching determines that the concatenation of the attributes *firstname* and *lastname* in Source 1 is semantically equivalent to the attribute *name* in Source 2. Duplicate detection then recognizes that the entry *John M. Smith* in Source 1 represents the same person as the entry *J. M. Smith* in Source 2.

This article focuses on the step that succeeds both schema matching and duplicate detection, namely data fusion. This final step combines different representations of the same real-world object (previously identified

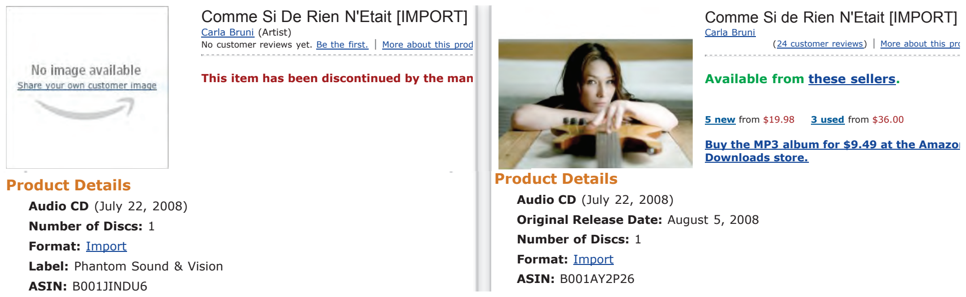
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Operator	Definition
Subsumption	A tuple t_1 *subsumes* another tuple t_2 ($t_1 \sqsupset t_2$), 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 [7]. The unary subsumption operator β removes subsumed tuples from a relation R, i.e., $\beta\ (R) = \{t \in R : \neg\exists t' \in R : t' \sqsupset t\}$.
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. A complementing set CS of tuples of a relation is a subset of tuples from R's extension 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. $S_1 \subset S_2$. 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.
Outer, minimum, and complement union	The *outer union* operator ($\uplus$) combines two relations $R_1 \uplus R_2$ with respective schemas S_1 and S_2 and extensions T_1 and T_2 by (i) constructing the result schema $S = S_1 \cup\ S_2$ and (ii) combining the two extensions to $T = T_1 \cup\ T_2$, where missing values are padded with $\perp$. The *minimum union* operator ($\oplus$) is defined as $A \oplus B = \beta\ (A \uplus B)$. The *complement union* operator ($\boxplus$) is defined as $A \boxplus B = \kappa\ (A \uplus B)$.

Table 5: Definitions for subsumption, complementation, outer union, minimum union, complement union [2]

during duplicate detection) into one single consistent representation. To do so, fusion has to resolve any conflicts existing among the duplicates. The data fusion result is a potentially more concise and more complete representation of a real-world object than the representations obtained from the individual sources.

We distinguish two types of conflicts between attribute values, namely *uncertainties* and *contradictions*. We qualify conflicts as uncertainties when all duplicates agree in an attribute value and at least one does not specify any value, i.e., contains a NULL value for the given attribute. Contradictions, on the other hand, exist between duplicates if they each do provide a value, albeit different. For instance, consider the two (out of five) representations of the CD "Comme si de Rien n'Etait" by Carla Bruni shown in Fig. 1. We observe that data concerning the music label or price information are only provided in one representation, hence, we qualify this as an uncertainty among the two representations. Opposed to that, both provide an ASIN value, but the values differ. Clearly, these values contradict each other.


Comme Si De Rien N'Etait [IMPORT]
Carla Bruni (Artist)
No customer reviews yet. Be the first. | More about this prod

This item has been discontinued by the man

Product Details
Audio CD (July 22, 2008)
Number of Discs: 1
Format: Import
Label: Phantom Sound & Vision
ASIN: B001JINDU6

Comme Si de Rien N'Etait [IMPORT]
Carla Bruni
(24 customer reviews) | More about this pro

Available from these sellers.

5 new from $19.98   3 used from $36.00

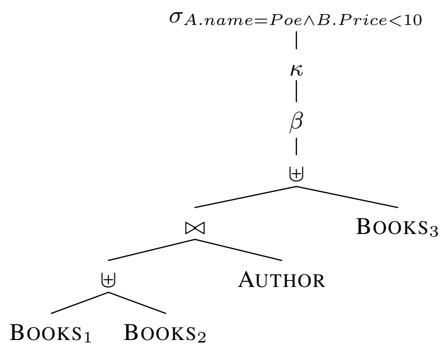
Buy the MP3 album for $9.49 at the Amazo Downloads store.

Product Details
Audio CD (July 22, 2008)
Original Release Date: August 5, 2008
Number of Discs: 1
Format: Import
ASIN: B001AY2P26

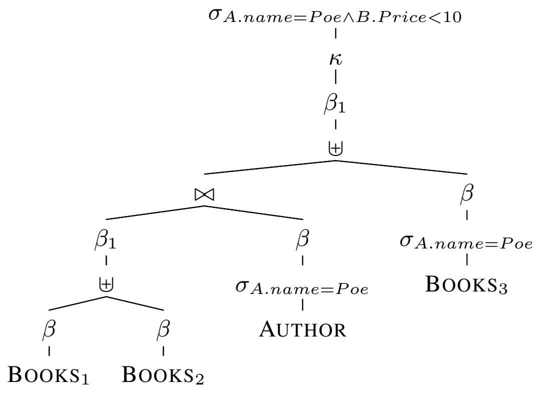
Figure 1: CD representations when searching for "Carla Bruni rien" on Amazon.com (January, 9th 2009)

There are many different strategies to handle conflicting data as surveyed in [1]. This paper focuses on resolving uncertainties using new techniques that follow the TAKE THE INFORMATION resolution strategy, which takes existing information and leaves aside NULL values [7, 8]. Our techniques stand out by being defined as database primitives. We previously defined relational operators and presented algorithms implementing them [2, 3] and repeat relevant definitions in Tab. 5.

The main contribution of this paper is a description of how these operators are used within relational query



(a) Plan A



(b) Plan B

Figure 2: Two query plans for a sample data fusion process

planning. More specifically, we present transformation rules for *minimum union* ($\oplus$) and *complement union* ($\boxplus$) and their building blocks *outer union* ($\uplus$) *subsumption* (β) and *complementation* (κ) [2, 3] in Sec. 2. Further, we show cost and selectivity estimates for use within a cost-based optimizer in Sec. 3 and some experimental results in Sec. 4. Sec. 5 briefly discusses related work and concludes.

2 Transformation Rules for Data Fusion Queries

To illustrate transformation rules, consider the following bookstore scenario: one bookstore stores its fiction books in relation BOOKS$_1$ and its 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. A second bookstore stores its books (authors included) in relation BOOKS$_3$. When integrating data from both bookstores, assume we are interested in (i) removing subsumed tuples, (ii) combining complementing tuples, and (iii) only selecting books written by Poe and that cost less than 10 EUR. This task can be expressed by a *data fusion query* and visualized by a query plan.

Fig. 2 depicts two query plans that express the above task. Plan A first unifies fiction and non-fiction books using *outer union* and then combines these with their authors. Next, the books from the second store are added. Uncertainties are then handled by subsequently applying *subsumption* and *complementation*. Finally, the desired *selection* is applied. In contrast, Plan B pushes *selection*s as far down as possible and removes subsumed tuples early in the process. Note that we distinguish two types of subsumption operators: β_1 is used when the input relation can be partitioned such that each partition does not contain subsumed tuples, which is the case when data without subsuming tuples has been previously combined, whereas β does not make use of such knowledge.

We summarize the query rewriting rules we have shown to be correct for moving the *subsumption* and the *complementation* operators around in query plans in Tab. 6. *Minimum union* and *complement union* can be moved around in a tree by splitting the operators into their individual components (*subsumption*/*complementation* and *outer union*) and moving these components separately.

Combinations with Outer Union: As there may be subsuming tuples across sources, simply pushing *subsumption* through *outer union*s is not possible without leaving an additional *subsumption* operation on top of the *outer union* (Rules S1 and S2). As mentioned above, β_1 denotes the *subsumption* operator that only tests for subsumed tuples across subsumption-free partitions of its input. Using a similar technique for *complementation*, i.e., a corresponding κ_1, 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 (Rules C1 and C2).

Combinations with Projection: 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*

Combinations with *outer union*	
C1	κ (A$\uplus$ B) $=\kappa$ (A) $\uplus$ κ (B), if there are no complementing tuples across sources, and
C2	κ (A$\uplus$ B) $\neq\kappa$ (A) $\uplus$ κ (B), in all other cases
Combinations with *projection*	
C3	κ (A) $\neq\kappa$ (π (A)), and
C4	π (κ (A)) $\neq\kappa$ (π (A))
Combinations with *selection*	
C5	κ (σ_c (A)) $\neq\sigma_c$ (κ (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
C6	σ_c (κ ($\sigma_{c\vee cnull}$ (A))) $=\sigma_c$ (κ (A)), in all other cases, with $cnull$ being the additional test for `NULL` values for all involved columns
Combinations with *cartesian product* and *join*	
C7	κ (A$\times$B) $=\kappa$ (A) $\times\kappa$ (B), if the base relations A and B do not contain subsumed tuples, and
C8	κ (A$\times$B) $=\kappa$ (κ (A) $\times\kappa$ (B)), in all other cases
Combinations with *grouping* and *aggregation*	
C9	$\gamma_{f(c)}$ (κ (A)) $=\gamma_{f(c)}$ (A), for any column c and aggregation function $f \in \{\max, \min\}$
C10	κ ($\gamma_{f(c)}$ (A)) $=\gamma_{f(c)}$ (A), for column c and any f
C11	$\gamma_{c,f(d)}$ (κ (A)) $=\gamma_{c,f(d)}$ (A), for columns c, d with c as grouping column not containing `NULL` values and aggregation function $f \in \{\max, \min\}$
C12	κ ($\gamma_{c,f(d)}$ (A)) $=\gamma_{c,f(d)}$ (A), for columns c, d with c as grouping column not containing `NULL` values and aggregation function $f \in \{\max, \min\}$
Other combinations	
C13	τ (κ (A)) $\neq\kappa$ (τ (A))
C14	δ (κ (A)) $=\kappa$ (δ (A))
C15	κ (κ (A)) $=\kappa$ (A)
C16	κ (A) $=$A, if A has only one or two attributes

(a) Rules for *complementation*

Combinations with *outer union*	
S1	β (A$\uplus$ B) $=\beta$ (A) $\uplus$ β (B), if there are no subsumed tuples across source relations, and
S2	β (A$\uplus$ B) $=\beta_1$ (β (A) $\uplus$ β (B)), in all other cases.
Combinations with *projection*	
S3	β (A) $\neq\beta$ (π (A)), and
S4	π (β (A)) $\neq\beta$ (π (A))
Combinations with *selection*	
S5	β (σ_c (A)) $\neq$ σ_c (β (A)), if c is of the following form: x `IS NULL`, with x being an attribute with `NULL` values, and
S6	β (σ_c (A)) $=\sigma_c$ (β (A)), in all other cases
Combinations with *cartesian product* and *join*	
S7	β (A$\times$B) $=\beta$ (A) $\times\beta$ (B)
S8	β (A$\bowtie_c$B) $=\beta$ (A) $\bowtie_c\beta$ (B), whenever *selection* with condition c can be pushed down
Combinations with *grouping* and *aggregation*	
S9	$\gamma_{f(c)}$ (β (A)) $=\gamma_{f(c)}$ (A), for any column c and aggregation function $f \in \{\max, \min\}$
S10	β ($\gamma_{f(c)}$ (A)) $=\gamma_{f(c)}$ (A), for any column c and any f
S11	$\gamma_{c,f(d)}$ (β (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 $f \in \{\max, \min\}$
(S12	β ($\gamma_{c,f(d)}$ (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	
S13	τ (β (A)) $\neq\beta$ (τ (A))
S14	δ (β (A)) $=\beta$ (δ (A))
S15	β (β (A)) $=\beta$ (A)
S16	κ (β (A)) $\neq\beta$ (κ (A))

(b) Rules for *subsumption*

Table 6: Transformation rules for *subsumption* and *complementation* in combination with other operators.

operator without changing the final result (Rules S3 and S4). For *complementation* and *projection*, the same considerations apply as for *subsumption* and *projection* (Rules C3 and C4).

Combinations with Selection: The main goal is to push *selection*s toward the base relations to decrease input cardinality. In case the *selection* is applied to a column c without `NULL` values (e.g., a key), we can indeed push *selection* down through the operator (Rule S6). 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 several cases resulting in Rules S5 and S6. 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 cases as for *subsumption* and need to assure that either both or none of the complementing tuples pass the *selection*. This property can be achieved for some cases when pushing *selection* through *complementation* by adding an additional condition B `IS NULL` to the selection predicate. However, as *complementation* combines its input tuples into new tuples, the original *selection* has to be applied again after *complementation* (Rules C5 and C6). If the operator tree includes a *selection* with conjuncts or disjuncts of conditions, we can push it entirely through *subsumption* or *complementation* if there is no single condition that prohibits the pushdown. Then, we first need to split the conditions using the standard transformation rules for *selection* and then push them according to the rules above.

Combinations with Join: When exchanging *subsumption* and *cartesian product* we must ensure that when applying *cartesian product*, (1) no additional subsuming tuples are introduced if there are none in the base relations, and (2) all subsuming tuple pairs in the base relations still exist after applying *cartesian product*. The former follows from the definition of tuple subsumption and 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 exchanging β and $\bowtie_c$ in the query plan, we need to apply the rules involving *selection* from the previous paragraph (Rules S7 and S8). Considering *complementation*, the same two properties as for *subsumption* have to be ensured. However, already the first property 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. We can fix this problem by introducing an additional κ operator on top that removes the newly introduced complementing tuples (Rules C7 and C8). Based on the above observation and the transformation rules for *selection* and *complementation*, a general rule for combining *join* and *complementation* cannot be devised.

Combinations with Grouping and Aggregation: In general, *subsumption* and *grouping* and *aggregation* are not exchangeable, because *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* (Rules S9-S13). If non-standard aggregation functions are allowed, 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*. Similar rules (Rules C9-C12) hold for *complementation*.

Other Combinations. The *subsumption* and *complementation* operators are not order-preserving (Rules S13 and C13). When dealing with bags of tuples, *subsumption/complementation* and *distinct* do not interfere with each other (Rules S14 and C14). Additionally, two *subsumption* operators can be combined into one (Rule S15). 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 (Rule S16). Finally, two *complementation* operators can be combined into one (Rule C15) and a relation needs to have at least three attributes for two tuples being able to complement each other (Rule C16).

Referring to the example in Fig. 2, Plan A can be transformed into Plan B by splitting the selection, applying rules C6, S6, pushing down selections, S1, S8, S1, and recombining the top selection again.

3 Cost Model and Selectivity Estimates for *Subsumption* and *Complementation*

To estimate runtimes of query plans we give a brief sketch of cost formulas for some of the implementations for computing *subsumption* and *complementation*. We consider the subsumption algorithms – presented in [2] – *Simple Subsumption* and *Null-Pattern-Based Subsumption* with special partitioning steps, which exhibited high efficiency in practice. For complementation we consider the *Simple Complement* algorithm and the *Partitioning Complement* algorithm, which were introduced in [3]. The cost formulas are based on the number of distinct tuple operations (e.g., tuple comparisons). 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 to 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.

Cost formulas: Let n be the number of tuples in the relation. The cost formula for the *Simple Subsumption* algorithm is given as $\mathcal{C}_{SMPS} = \frac{1}{2}n^2$. The cost formula for the *Null-Pattern-Based Subsumption* algorithm is given as $\mathcal{C}_{NPBS} = n \log n$. We cover its partitioning variant in more detail; all algorithms are given in [2]. Assuming $k = |P_\perp|$ as the size of the NULL partition and p as the number of partitions P_i, then the average size of a partition P_i is $\frac{n-k}{p}$ (assuming uniform distribution). Furthermore, we assume an intermediate selectivity of 100% (no tuple subsumes the other), assume using the *Simple Subsumption* algorithm and that creating the partitioning can be done at practically no cost while reading the relation into memory. Then, the cost formula for the *Partitioning(SMPS)* algorithm is: $\mathcal{C}_{\textit{Partitioning(SMPS)}} = \frac{1}{2}k^2 + \frac{1}{2}\frac{(n-k)^2}{p} + nk - k^2$. We consider two special cases: If there is no NULL partition, then $k = 0$ and the formula is simplified to $\mathcal{C}_{\textit{Partitioning(SMPS)}} = \frac{1}{p}\frac{1}{2}n^2$, thus showing the performance boost by partitioning. This effect has been verified in experiments. The second special case is to 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 $\mathcal{C}_{\textit{Partitioning(SMPS)}} = -\frac{1}{2}k^2 - \frac{1}{2}k + \frac{1}{2}n + nk$, 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. A cost formula for *Partitioning(NPBS)* can be devised accordingly: $\mathcal{C}_{\textit{Partitioning(NPBS)}} = k\log k + (n-k)\log\left(\frac{n-k}{p}\right) + (n-k)k$.

For *complementation* we consider cost functions for the *Simple Complement* algorithm and the *Partitioning Complement* algorithm given in [3]. 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 $\mathcal{C}_{\textit{Simple Complement}} = \frac{1}{2}n(n-1)2^m + n2^m$. 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 to 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: $\mathcal{C}_{\textit{Partitioning Complement}} = 2^m p\left(\frac{1}{2}\left(\frac{n-k}{p} + k - 1\right)\left(\frac{n-k}{p} + k\right) + \left(\frac{n-k}{p} + k\right)\right) + pk$. We consider two special cases: If there is no NULL partition, then $k = 0$ and we simplify to $\mathcal{C}_{\textit{Partitioning Complement}} = \frac{1}{2}\frac{1}{p}n^2 2^m + \frac{1}{2}n2^m$. Considering the 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: $\mathcal{C}_{\textit{Partitioning Complement}} = 2^m(n-k)(\frac{3}{2}k + \frac{1}{2}k^2 + 1) + nk - k^2$.

Selectivity estimation: Given a selectivity factor $\mathcal{S}$ for an operator, the output cardinality of an operation is computed by multiplying it with the input cardinality. In the case of subsumed and complementing tuples, selectivity estimation is accomplished by estimating how many tuples will be subsumed or complemented by computing probabilities of NULL values existing in tuples.

Consider the case of a simple relation with only one attribute. All tuples that are NULL in that single attribute are subsumed by others. The number of NULL values can easily be deduced from the relations histogram; the probability $\mathcal{P}(a_1 = \perp)$ that a tuple has a NULL value in attribute a_1 can be determined likewise. We present a simple model for estimating selectivity based on this limited information. However, estimating the correct number of subsumed tuples in general is a difficult task: Our experiences with real-world datasets show that (1) this number can significantly vary, and that (2) NULL values – which highly influence this number – are not always evenly distributed among tuples and columns.

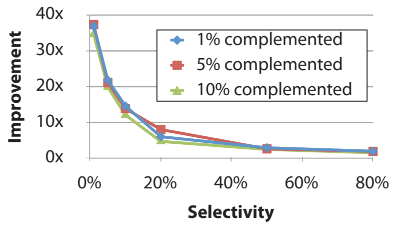
In the following we assume a) an equal distribution of values in an attribute and b) independence among attributes. We estimate the number of subsumed or complementing tuples by computing the number of tuples that contain NULL values, as this is the precondition for a tuple to be subsumed or complementing. 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 all NON-NULL attributes.

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 $\mathcal{P}_2$ that a tuple contains at least one NULL values is $1 - (1 - p_1^\perp)(1 - p_2^\perp)$. 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: $\mathcal{P}_k = (\text{tuple contains at least one NULL value}) = 1 - \mathcal{P}(\text{all } a_i = \perp) = 1 - \mathcal{P}(\bigwedge_{i=0}^k a_i \neq \perp) = 1 - \prod_{i=1}^k (1 - p_i^\perp)$ The likelihood $p_i^\perp$ can be approximated by the count of NULL values in the attributes from the relations histogram. If all values in a_i are uniformly distributed, then $p_i^\perp = \frac{1}{|a_i|}$ with $|a_i|$ being the number of different attribute values in a_i. Thus, $\mathcal{P}_k^= = 1 - \prod_{i=1}^k (1 - \frac{1}{|a_i|})$.

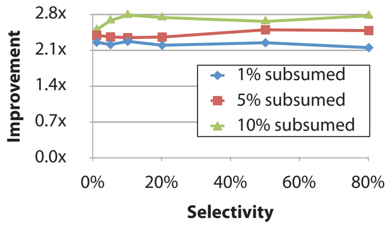
Selectivity estimation for *subsumption* and *complementation* then is determined using probability P_k: $\mathcal{S}_\beta = \mathcal{S}_\kappa = 1 - \mathcal{P}_k = \prod_{i=1}^{k}(1 - p_i^\perp)$ for operators β and κ and $p_i^\perp$ being the likelihood of a NULL value present in column a_i. Additionally, 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: $\mathcal{S}_\beta \geq \frac{\max|a_i|}{|R|}$ and $\mathcal{S}_\kappa \geq \frac{\max|a_i|}{|R|}$.

4 Experimentation

We exemplarily evaluate transformation Rules C6 and S8, measuring the effect they have on runtime. More specifically, in both cases, we measure the runtime of a set of simple queries on artificial data where a *selection*/*join* follows a *complementation*/*subsumption* and compare their runtime to the transformed queries where *complementation*/*subsumption* follows *selection*/*join*. In both cases, we respectively varied the percentage of complemented and subsumed tuples (1%, 5%, and 10%). Also, we tested different selectivities between 1% and 80% for the *selection* and *join* operator, respectively. We report the performance gain of both cases in Fig. 3, on generated tables with six columns. Runtimes are median values over ten runs and the figure shows the values for tables of (a) one million and (b) 20.000 tuples. As implementations of *subsumption* and *complementation*, we used *Partitioning Complement* and the *Simple* algorithm [2, 3], respectively.



(a) Pushing *selection* below *complementation*



(b) Pushing *subsumption* below *join*

Figure 3: Runtime improvement on artificial data when applying Rules C6 and S8.

Regarding Fig. 3(a), we see that pushing *selection* below *complementation* pays off. A similar improvement (not shown here) has been observed for the combination *selection*/*subsumption* (Rule S6). Indeed, the performance gain 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 (*selection* pushed down) is always faster for the tested combinations of selectivity, size, and percentage of subsumed tuples. Experiments with other table sizes show similar results. 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 results reported in Fig. 3(b) show an improvement of around 2.5x for all selectivities and the three percentages of subsumed tuples. Experiments with other table sizes show a comparable behavior. 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 considered operators: whereas the *subsumption* operator is optimized for performance, the main part of the queries' total runtime is consumed by the implementation of the *join*[1]. Essentially, Fig. 3(b) shows the improvement caused by the reduced input cardinality of the *join*. The improvement is higher for tables with 10% subsumed tuples (than it is for tables with 5% or even 1% subsumed tuples), because more subsumed tuples are removed from the input to the *join*. Although the improvement is not as large as in the other experiment, it shows that applying the transformation rule pays off.

When operators can be eliminated from the tree (e.g., C9-C12 and S9-S12) performance gain is evident. Other rules, such as C3-C4 and S3-S4, do not need evaluation. Experimentation for the remaining rules is deferred to future work.

[1]We implemented our operators in the XXL framework [5] and use its general-purpose Nested-Loop-Join implementation.

5 Related Work and Conclusions

Related work on conflict resolution and data fusion is widely covered in [1]. The *minimum union* operator [7] is used in many applications, e.g., in query optimization for *outer join* queries [6, 9]. However, an efficient algorithm for the general subsumption task is still considered an open problem therein. Different to our work is the assumption that the base relations do not contain subsumed tuples and the use of *join* instead of *union* to combine tables before removing subsumed tuples. [9] proposes a rewriting for *subsumption*, 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. [2] presents efficient algorithms to compute subsumption in the general case as well as more related work. Data fusion with the semantics of *complement union* has been first introduced in [2, 3], together with definitions, implementation details and a first draft of transformation rules. 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 [4].

To conclude, data fusion, despite its seemingly simple nature, is a complex and important task in the field of data integration. In this article we have addressed only cases in which a NON-NULL value competes with a NULL-value – deciding to choose the NON-NULL value for the fused record is natural. However, doing so in a consistent and efficient manner is not trivial. We have introduced the two concepts of subsumption and complementation as new algebraic operators, and have shown how to incorporate them into relational DBMS by providing corresponding transformation rules, a simple cost model and selectivity estimates. Future work lies in the direction of a refinement of the cost model.

Acknowledgments. This research was partly funded by the German Research Society (DFG grant no. NA 432) and was performed while the authors worked at the Hasso Plattner Institute.

References

[1] Jens Bleiholder and Felix Naumann. Data fusion. *ACM Computing Survey*, 41(1):1–41, 2008.

[2] Jens Bleiholder, Sascha Szott, Melanie Herschel, Frank Kaufer, and Felix Naumann. Subsumption and complementation as data fusion operators. In *International Conference on Extending Database Technology (EDBT)*, 2010.

[3] Jens Bleiholder, Sascha Szott, Melanie Herschel, and Felix Naumann. Complement union for data integration. In *International Workshop on New Trends in Information Integration (NTII)*, 2010.

[4] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[5] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. XXL - a library approach to supporting efficient implementations of advanced database queries. In *International Conference on Very Large Databases (VLDB)*, 2001.

[6] César Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)*, 22(1):43–74, 1997.

[7] César A. Galindo-Legaria. Outerjoins as disjunctions. In *ACM International Conference on Management of Data (SIGMOD)*, 1994.

[8] Sergio Greco, Luigi Pontieri, and Ester 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.

[9] Jun Rao, Hamid Pirahesh, and Calisto Zuzarte. Canonical abstraction for outerjoin optimization. In *ACM International Conference on Management of Data (SIGMOD)*, 2004.