Complement Union for Data Integration

Jens Bleiholder [#1], Sascha Szott [+2], Melanie Herschel [*3], Felix Naumann [#4]

[#] *Hasso-Plattner-Institut, Universität Potsdam, Germany*
[1,4] {jens.bleiholder | felix.naumann}@hpi.uni-potsdam.de
[+] *Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany*
[2] szott@zib.de
[*] *Universität Tübingen, Germany*
[3] melanie.herschel@uni-tuebingen.de

I. Introduction

A data integration process consists of mapping source data into a target representation (*schema mapping* [1]), identifying multiple representations of the same real-word object (*duplicate detection* [2]), and finally combining these representations into a single consistent representation (*data fusion* [3]). Clearly, as multiple representations of an object are generally not exactly equal, during data fusion, we have to take special care in handling data conflicts. This paper focuses on the definition and implementation of *complement union*, an operator that defines a new semantics for data fusion.

Complement union aims at resolving a special type of data conflict, called *uncertainty*. Intuitively, the operator considers 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. *Complement union* combines tuple sets by first computing the *outer union* and then combining tuples that *complement* each other. This modular specification gives us more flexibility during integration; complement can also be used to clean source databases before the integration process and it allows us to potentially use more sophisticated schema mapping algorithms than the outer union.

Example. Consider a disaster management scenario. We integrate two databases *Police* and *Hospital* (excerpts shown in Fig. 1 (a) and (b)). Note that *tid* is not a relational attribute but serves as a reference to tuples. The goal is to detect missing persons that have been admitted to a hospital and to contact their relatives when an address is available.

Fig. 1 (c) shows the result of applying complement union to the two sources. Complement union is the result of applying an outer union followed by *complementation*. The schema of the outer union of *Police* and *Hospital* is the union of schemas of both tables, e.g., (Name, DOB, Sex, Address, Blood) and each tuple from the sources is padded with nulls in the result of outer union. That is, tuple 1 becomes (Miller, 7/7/59, m, 12 Main, ⊥) and tuple 5 becomes (Miller, 7/7/59, m, ⊥, O). These tuples have same values for Name, DOB, and Sex. Tuple 1 contributes Address whereas tuple 5 contributes Blood to the final tuple during complementation.

Now, consider tuples 1 and 4. Assume that according to duplicate detection, they represent the same real-world object, i.e., the same person. They both store different non-null values for Sex. Complement union cannot solve this type of conflict, for which specialized methods exist (see [3] for a survey on fusion methods).

To the best of our knowledge, this work is the first that considers data fusion with the general semantics of complement union. A known operator that is able to combine complementing tuples from two sources is the full disjunction [4]. However, it cannot combine complementing tuples from the same source, nor can it combine tuples where overlapping/join attributes contain NULL values.

Real-world applications. In our experience with real-world data, we observe the usefulness of complementation. For instance, in a sample of the Internet Movie Database (IMDB)[1], we observe that complementation occurs with a frequency of up to 20 times higher than subsumption, the state-of-the art data fusion operator to handle uncertainty. Also, we find numerous examples on the web, e.g., Fig. 2 shows excerpts of two complementing descriptions of the same CD.

As complementation combines information that was not already combined in one of the sources, we obtain a more

[1] IMDB: http://www.imdb.com

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

(a) *Police*

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

(b) *Hospital*

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

(c) complement union of *Police* and *Hospital*

Fig. 1. Complement union of Police and Hospital databases

Fig. 2. Complementing entries at Amazon.com, as of Oct. 2nd, 2009. ASIN is not a real-world key like ISBN, but a generated product identifier. During data integration, we project it out and assign a new identifier afterwards.

complete description of an object, albeit at the risk of creating inaccurate combinations. This behavior is adequate in scenarios where occasional incorrect combinations have little negative impact on the application and the combined information is valuable, e.g., in our disaster management scenario.

Contributions & Structure. We define complementation as a database primitive, as defined in Sec. II. We contribute (1) efficient algorithms to compute complementation (Sec. III); (2) a study of how complement can be moved in operator trees for query optimization (Sec. IV), and (3) a comparative experimental evaluation on both artificial and real-world data (Sec. V), before we conclude in Sec. VI.

II. Definitions

Definition 1 (Tuple Complementation): Two tuples t_1 and t_2 complement each other ($t_1 \gtrless t_2$) if (1) t_1 and t_2 have the same schema, (2) values of corresponding attributes 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 NON-NULL attribute value combination in common. □

We introduce condition (3) to strictly separate equality, subsumption and complementation, and introduce condition (4) to assure that tuples are not completely unrelated. Complementing tuples t_1 and t_2 are combined into a tuple t_c, the *complement* of t_1 and t_2. Tuple t_c is created by coalescing the values of each attribute a from t_1 and t_2, i.e.,

$$t_c.a := \begin{cases} t_1.a, & t_2.a \text{ is NULL or } t_1.a = t_2.a \\ t_2.a, & t_1.a \text{ is NULL.} \end{cases} \quad (1)$$

Similarly, we construct the complement of more than two tuples. Tuple complementation is a symmetric relationship. It is neither reflexive nor transitive. In the following, we say that a tuple t_1 complements t_2 if $t_1 \gtrless t_2$ holds.

Definition 2 (Maximal complementing set): A complementing set CS is a subset of tuples of relation R where, for each pair t_i, t_j of tuples from CS it holds that $t_i \gtrless t_j$. A complementing set S_1 is a maximal complementing set MCS if there exist no other complementing set S_2 s.t. $S_1 \subset S_2$. □

A tuple of R that does not complement any other tuple forms a maximal complementing set of size one. A tuple can be in more than one MCS and in general there are multiple MCS for a relation R. A relation R can be uniquely divided into a set of MCSs. All tuples of a maximal complementing set can be combined into one tuple, the complement t_c.

Definition 3 (Complementation Operator): The unary complementation operator κ replaces each existing maximal complementing set MCS_i of a relation R by the single complement tuple t_{c_i} of all tuples in MCS_i. □

Definition 4 (Complement Union): The *complement union* operator ($\boxplus$) is the composition of *outer union* ($\uplus$) and *complementation* (κ) s.t. $A \boxplus B = \kappa\ (A \uplus B)$; complementing tuples in the result of the *outer union* of the two input relations A and B are replaced by their complement. □

Complement union is commutative but not associative. The operator forms complements of complementary tuples from the same source and from different sources.

III. Implementing Complementation

We present two algorithms for computing the complementation operator (κ), namely the *partitioning complement (PC)* and the *null pattern complement (NPC)* algorithm.

The main idea of partitioning complement (PC) is based on the maximal complementing sets defined in Sec. II. In a first pass the algorithm goes through all input tuples and groups tuples that complement each other, thus building all sets of complementing tuples. In a second pass, the algorithm goes through all these sets and outputs the complement of all the tuples of all maximal complementing sets.

In Fig. 3 we show the complement relationships among some tuples numbered 1 to 7 (*tid*). An edge represents a complement relationship. In such a graph representation, a maximal complementing set corresponds to a maximal clique.

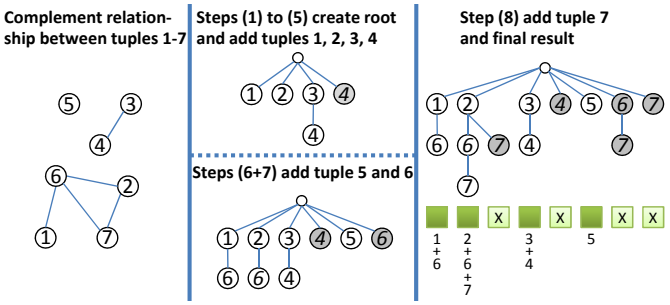

Fig. 3. Example for partitioning complement algorithm (PC)

To compute and store the sets of complementing tuples, we subsequently insert each tuple of the relation into a tree data structure. We collect the sets of complementing tuples that originate in that particular tuple in the subtree below it.

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

Then, tuples 5 and 6 are inserted. Tuple 6 is inserted below tuple 1 and below tuple 2 as it complements both. It is further inserted below the root (marked). Finally, tuple 7 is inserted in Step 8. It is inserted below the combination of tuples 2 and 6 (path root–②–⑥), due to the fact that it complements the complement of 2 and 6: If a tuple t complements two other tuples r and s then it also complements the complement of r and s. As tuple 7 complements both tuples 2 and 6 alone, it is also inserted as a child of both of them. Due to the depth-first order, in both cases it is marked. It is also inserted as child of the root node, also marked.

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

The worst case time complexity of the algorithm is dominated by the second pass, because the entire tree structure is traversed to visit, check and—where necessary—output the leaf nodes (together with their paths from the root node). The runtime complexity of this algorithm (including the time needed for enumerating all MCSs and building the complements) is then exponential in the size of the largest subtree (which is equal to the size of the largest MCS and usually much smaller than the number of tuples, n), i.e., $\mathcal{O}(n2^k)$ where k is the size of the largest MCS. We refer to this version as the *simple complement algorithm*.

To considerably improve runtime when computing complement, we partition the relation by the different values of a fixed column c and apply the algorithm to each partition individually. The NULL partition containing all tuples with a NULL value in attribute c needs to be handled differently, because complementation is not transitive. For each NON-NULL partition we add the tuples from the NULL partition to it and then replace complementing tuples. We then split the resulting set of tuples into two groups: (1) the set of all tuples that came from the null partition and have not been complemented and (2) all other tuples. Eventually, based on this division, we identify all tuples from the null partition that have not been used in *any* of the partitions and add these to the final output. We refer to this improved version as the *partitioning complement algorithm*.

We also implemented a second, different complementation algorithm, the *null pattern complement* algorithm. This algorithm partitions the input relation according to the patterns of tuples' NULL values and compares only tuples with complementing NULL patterns. It is a modification of an algorithm for removing subsumed tuples [5]. Due to space constraints we only report on experimental results.

IV. Complementation in Query Plans

We now examine how the complementation operator can be moved in logical query plans to potentially increase efficiency. We summarize the discussed rewrite rules in Table I.

Combinations with (Outer) Union. Complement and union are not exchangeable in general, due to the fact that complementation is not a transitive relationship. However, if there are no NULL values in common attributes (e.g., keys), complementation and outer union are exchangeable (Rule (1)).

Combinations with Selection. If selection is applied on a column without NULL values (e.g., key, `NOT NULL` column), we can push selection through complementation (Rule (2)). If a column allows NULL values, pushing selection through complementation alters the result only if a tuple complementing another tuple is removed from the input. We can however push the selection partially down (Rule (3)).

(1)	$\kappa\ (\text{A} \uplus \text{B}) = \kappa\ (\text{A}) \uplus \kappa\ (\text{B})$, only if $\nexists t \in A, t' \in B : t \gtrless t'$
(2)	$\kappa\ (\sigma_c\ (\text{A})) = \sigma_c\ (\kappa\ (\text{A}))$, if all columns involved in c do not contain NULL values (e.g., a key)
(3)	$\sigma_c\ (\kappa\ (\sigma_{c \vee cnull}\ (\text{A}))) = \sigma_c\ (\kappa\ (\text{A}))$, if columns involved in c may contain NULL values and c is not of the following form: x `IS NULL`, x `IS NOT NULL`, x being an attribute. $c \vee cnull$ is the original condition appended by the test for NULL values (`IS NULL`) for all the columns x involved in c.
(4)	$\kappa\ (\text{A} \times \text{B}) = \kappa\ (\text{A}) \times \kappa\ (\text{B})$, if the base relations A and B do not contain subsumed tuples, and
(5)	$\kappa\ (\text{A} \times \text{B}) = \kappa\ (\kappa\ (\text{A}) \times \kappa\ (\text{B}))$, in all other cases
(6)	$\kappa\ (\kappa\ (\text{A})) = \kappa\ (\text{A})$
(7)	$\gamma_{f(c)}\ (\kappa\ (\text{A})) = \gamma_{f(c)}\ (\text{A})$, for any column c and aggregation function $f \in \{\max, \min\}$
(8)	$\kappa\ (\gamma_{f(c)}\ (\text{A})) = \gamma_{f(c)}\ (\text{A})$, for column c and any f
(9)	$\gamma_{c,f(d)}\ (\kappa\ (\text{A})) = \gamma_{c,f(d)}\ (\text{A})$, for columns c, d with c as grouping column not containing NULL values and aggregation function $f \in \{\max, \min\}$
(10)	$\kappa\ (\gamma_{c,f(d)}\ (\text{A})) = \gamma_{c,f(d)}\ (\text{A})$, for columns c, d with c as grouping column not containing NULL values and aggregation function $f \in \{\max, \min\}$

TABLE I
Rewrite Rules for Complementation

Combinations with Join. In general, complementation and the Cartesian product cannot be exchanged: applying the Cartesian product introduces complementing tuples if the relations contain subsumed tuples (Rules (4) and (5)). A general rule for combining join and complement cannot be devised, so joins and complementation are not exchangeable.

Other Combinations. Two complementation operators can be combined into one (Rule (6)). In general, complementation and grouping/aggregation are not exchangeable as complementation may remove tuples that are essential for the computation of the aggregate. However, there are cases in which we can remove the complementation operator and leave only the grouping/aggregation (Rules (7), (8), (9), and (10)).

V. Experiments

To evaluate our algorithms, we experimented both with synthetic and real-world data. The synthetic datasets consist of integer data in six columns. To validate our approaches on real-world data, we used data integrated from IMDB, Filmdienst[2], and three other smaller movie sources. We discuss results obtained on Actor and Movie relations taken from this integrated data set. We also report on results obtained on a sample of the FreeDB database[3].

A 2.3GHz dual processor quad core server with 16GB of main memory was used to store the data in a relational database (IBM DB2 v9.5) and to perform the experiments that study the runtime of different algorithms on varying datasets. All algorithms are implemented in Java SE 6 and the reported runtimes are median values over 5 runs. We

[2]Data kindly provided by Filmdienst: `film-dienst.kim-info.de`
[3]FreeDB: `www.freedb.org`

evaluate the algorithms discussed in this paper, i.e., Simple, Partitioning Complement (PC), and Null-Pattern Complement (NPC) with algorithms adapted from the literature. Replacing complementing tuples by their complement in a relation is equivalent to finding all maximal cliques [6] in a graph that has been constructed by creating one node per tuple and an edge between nodes if the corresponding tuples complement each another. We also evaluate the performance of an adaptation of an algorithm, to which we refer to as the Johnson algorithm, for the dual problem of finding independent sets [7].

Experiment 1: Algorithm Comparison. We generate tables of varying size that contain a varying percentage of complementing tuples (1%, 5%, and 10%). Figure 4 shows the runtime for Simple Complement, PC, and Dynamic, the approach from [6] for 1% and 10% of complementing tuples.

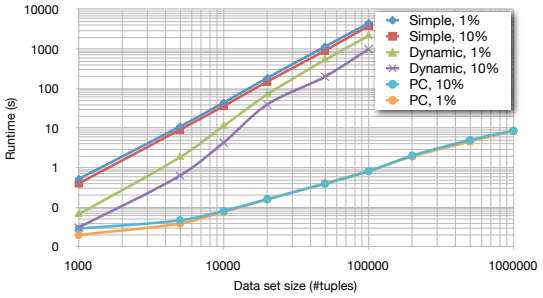

Fig. 4. Runtimes for different percentages of complementing tuples.

Discussion. We observe that Dynamic is considerably faster than the Simple Complement. However, as the runtime of Simple Complement decreases with increasing percentage of complementing tuples, the runtime of Dynamic increases. This is mainly due to the data structures used and the runtime of Simple Complement being mainly dominated by the size of the largest clique. The algorithm from [7] (not shown) performs very poorly, even for only 100 tuples (several seconds), due to its computational complexity. We also observe that partitioning pays off, as Partitioning Complement is the fastest among the shown algorithms. However, applying partitioning to all other approaches results in a comparable runtime, even for [7]. All partitioning algorithms scale well for relations of up to 1 million tuples. Varying the column for partitioning shows significant differences in runtime for all algorithms, so care must be taken in choosing a partitioning column.

Experiment 2: Complementation on real-world data. We now consider the real-world datasets Actor, Movie, and FreeDB. For each of these datasets, we take samples of different sizes and measure the runtime of the same algorithms as in the previous experiment (Fig. 5).

Discussion. For the FreeDB dataset, which consists of 10,000 tuples, NPC takes 0.16 seconds and thereby outperforms PC (0.34 sec. with best partitioning). Results on Actor an Movie datasets are also represented in Fig. 5. We see that for both Actors and Movies, the PC algorithm outperforms NPC. This is mainly due to the beneficial partitioning, although the number of columns (12 vs. 27 columns) also has some influence: PC better copes with large schemata. Interestingly,

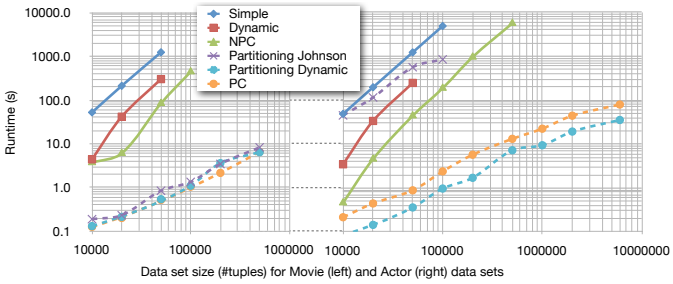

Fig. 5. Runtimes for complement algorithms for real-world datasets.

NPC seems better suited for datasets with less NULL values (Actors) whereas results show that the opposite is the case for PC: this algorithms performs slightly better on the dataset with more overall NULL values (Movies). However, in this case we cannot completely rule out the influence of the larger number of attributes of Movies (27 columns).

We further investigated the runtime of related algorithms, as in the previous experiments (see Fig. 5). The superiority of the approach from [6] (approx. 34s for 20k tuples) over simple complement (although being worse than NPC and PC) and the corresponding partitioning version over all other algorithms is mainly due to the small percentage of complementing tuples in the data sets (below 1%). In contrast to the generated data sets, the partitioning variants show differences in runtime, especially [7] performs worse than for the generated datasets (nearly 40s for 10k tuples).

VI. Conclusion and Outlook

This paper addressed the problem of combining multiple tuples to a more concise representation in the context of data integration. We defined the complement union operator and presented several algorithms as physical implementations of it. In the future, we plan to look more closely at different types of partitioning to further improve the algorithms. We also plan to study the interaction between the complementation and the subsumption operator, to potentially devise a combined algorithm for efficient overall data fusion.

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

References

[1] 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*, 2002.

[2] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, 2007.

[3] J. Bleiholder and F. Naumann, "Data fusion," *ACM Computing Survey*, vol. 41, no. 1, 2008.

[4] S. Cohen and Y. Sagiv, "An incremental algorithm for computing ranked full disjunctions," in *Proc. of PODS*. New York, NY, USA: ACM Press, 2005.

[5] J. Bleiholder, S. Szott, M. Herschel, F. Kaufer, and F. Naumann, "Subsumption and complementation as data fusion operators," in *Proc. of EDBT*, Lausanne, Switzerland, 2010.

[6] V. Stix, "Finding all maximal cliques in dynamic graphs," *Comput. Optim. Appl.*, vol. 27, no. 2, 2004.

[7] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis, "On generating all maximal independent sets," *Inf. Process. Lett.*, vol. 27, no. 3, 1988.