Schema mapping generation in the wild

DOI:
10.1016/j.is.2021.101904

Document Version
Accepted author manuscript

Schema Mapping Generation in the Wild

Lacramioara Mazilu, Norman W. Paton, Alvaro A.A. Fernandes, Martin Koehler

[a] *University of Manchester, School of Computer Science, Oxford Road, Manchester, M139PL, Greater Manchester, United Kingdom*

Abstract

Schema mappings enable declarative and executable specification of transformations between different schematic representations of application concepts. Most work on mapping generation has assumed that the source and target schemas are well defined, e.g., with declared keys and foreign keys, and that the mapping generation processes exist to support the data engineer in the labour-intensive process of producing a high-quality integration. However, organizations increasingly have access to numerous independently produced data sets, e.g., in a data lake, with a requirement to produce rapid, best-effort integrations, without extensive manual effort. As a result, there is a need to generate mappings in settings without declared relationships, and thus on the basis of inferred profiling data, and over large numbers of sources. Our contributions include a dynamic programming algorithm for exploring the space of potential mappings, and techniques for propagating profiling data through mappings, so that the fitness of candidate mappings can be estimated. The paper also describes how the resulting mappings can be used to populate single and multi-relation target schemas. Experimental results show the effectiveness and scalability of the approach in a variety of synthetic and real-world scenarios.

Keywords: mapping generation, profiling data, dynamic programming
PACS: 0000, 1111
2000 MSC: 0000, 1111

Email addresses: lacramioaramazilu@gmail.com (Lacramioara Mazilu), norman.paton@manchester.ac.uk (Norman W. Paton), fernandesaaa@gmail.com (Alvaro A.A. Fernandes), koehler.martin@gmail.com (Martin Koehler)

1. Introduction

A schema mapping generation algorithm constructs views for populating a target database schema from source schemas. Such algorithms may be informed by different information, such as schema matches, integrity constraints or instance data. The most substantial body of work on schema mapping generation was initiated in the Clio project (as reviewed in [1]), and has given rise to comprehensive results on subjects such as data exchange [2], merging of mappings [3], debugging of mappings [4] and mapping verification [5]. This line of research has focused primarily on supporting experts who are developing mappings between a single source schema and target schema, where the source and the target are database schemas with defined constraints, e.g., declared keys and foreign keys. In such a setting, the goal is to provide tool support that helps experts curate matches and develop mappings between (potentially complex) source and target schemas [6]. However, with the growing availability of open data sets, and the emergence of data lakes, mapping generation over independently-produced data sets, with minimal explicit constraints, is arguably as important as it is for schemas with declared keys and foreign keys.

More recently, the development of techniques for web data extraction, the publication of extensive open data sets, and the adoption of data lakes means that organisations typically have access to numerous sources in a domain of interest, and a requirement to integrate data on a topic at manageable cost. In such a setting, the relevant data may come from data sets from many independently-developed data sources, and thus the relationships between these data sets are unlikely to be declared explicitly. Instead, relationships between independent data sets may be inferred by different means, e.g., through data profiling [7]. The relationships that are inferred may include (partial) inclusion dependencies and candidate keys, and can also include inferred matches [8] between attributes in source and target tables. Where there are many data sources, such relationships give rise to a large space of candidate mappings, which are likely of variable utility. A mapping generation algorithm must explore the space of candidate mappings, making informed decisions as to which tables should be combined and how, based on the available matches and profiling data.

In this paper, we tackle the problem of mapping generation over independent data sets that come from different origins, i.e., without declared relationships. We refer to this as *mapping generation in the wild*. For this,

we describe an approach to mapping generation that seeks to populate a target schema using data from many source tables, potentially drawn from different domains, informed by automatically produced profiling data. The contributions of the paper are:

1. A dynamic programming algorithm that explores the space of candidate mappings, identifying opportunities for combining source relations on the basis of intra-source and inter-source information, viz. relational metadata and profile data [7], respectively. Reflecting its algorithmic basis, the proposal is referred to as Dynamap.

2. Rules for deriving profile data for mappings from their operands. As mapping generation is informed by profiling data, in particular candidate keys and (partial) inclusion dependencies, it is important that such profiling data can be propagated from source tables to candidate mappings, without the expense of evaluating candidate mappings and running a profiler (e.g. [9]) on their results. It is shown how to derive, and occasionally estimate, profiling data on the results of unions, joins and outer joins.

3. Techniques for pruning the space of candidate mappings. Where there are many sources, there can be a combinatorial explosion in the number of candidate mappings. We identify circumstances in which mappings can be pruned from the search space, thus supporting the generation of mappings that combine many source tables.

4. A method that populates a multi-relation target schema where the mapping generation algorithm tries to satisfy schema constraints (such as primary and foreign keys) when populating the target.

5. An empirical evaluation of the approach: *(i)* an exploration of applicability on various iBench [10] scenarios; *(ii)* an evaluation with real world data from two domains; and *(iii)* an analytical study with generated scenarios involving large numbers of sources. The experiments investigate mapping quality, the time spent generating mappings, the accuracy of inferred profiling data, and the impact of different pruning strategies.

This work extends that in [11] through the addition of two components to the search (Section 5) that tackle multi-relation target schemas that are subject to constraints. Also, we extend the set of experiments through five new ones including *i)* a new real-world domain, *ii)* a variation of the real-world experiment in [11], *iii)* an experiment for the efficiency of the pruning strategies, and *iv)* various iBench scenarios with multi-relation target schemas with constraints.

The remainder of this paper is structured as follows. Section 2 situates Dynamap in relation to other work on mapping generation. Section 3 describes the problem we are tackling in this paper. Section 4 describes the application of dynamic programming to mapping generation, the decision procedure for combining sources in mappings, and the propagation of profiling data through mappings. Also, as the search space can be prohibitively large, Section 4 describes how less relevant portions of this space can be pruned, and a fitness function that can be used to score candidate mappings. Section 5 describes pre-processing and post-processing steps to the search component that extend the applicability of the approach to multi-relation target schemas. In Section 6, Dynamap is evaluated with respect to output data and processing time on a set of real-world and synthetic scenarios. Section 7 concludes.

2. Related Work

Schema mappings can be created manually if there are experts that understand the characteristics of the sources and of the desired target, such as the data model descriptions, format and constraints. However, manual authoring is labour-intensive, and the automation of schema mapping generation has been the subject of significant research and development effort. Indeed, mapping generation has become ever more relevant considering the growth in available datasets that need integration [6]. In this section, we review work on *mapping generation for databases*, on *mapping generation in the wild*, and on *mapping generation as search*.

Schema mapping generation. A schema mapping generator has the signature $MapGen(S, T, MD_S, MD_T, MD_{S \to T}) \to M$, where M is a set of generated mappings expressed in a query language; S is a (potentially singleton) set of source schemas; T is the target schema; MD_S is metadata about the sources; MD_T is metadata about the target; and $MD_{S \to T}$ is metadata that relates S to T. The mappings in M are often expressed as *source-to-target*

tgds [12], as this abstracts over the conceptual model underlying the database system, but, in order to execute them over the data, they are typically translated into an executable query language, e.g., SQL. A mapping generation operation such as *MapGen* could, for example, be made available as part of a library of data preparation operations that support model management [13].

Schema mappings for databases. In relation to *mapping generation for databases*, probably the most influential proposal is Clio [1], where MD_S and MD_T include not only type information, but crucially also key and foreign key constraints; and $MD_{S \to T}$ consists of pairwise associations between attributes from S and T. Importantly, Clio was produced to support an integration expert in the development of schema mappings, and S is assumed to be a single schema. As a result, although the Clio algorithm can be run over multiple source schemas, the transformations used in mapping generation tend to assume that the source contains declared keys and foreign keys.

A significant body of work can trace its technical ancestry to Clio, though typically retaining the assumptions that MD_S includes declared foreign keys and that mapping generation is being performed to support an expert in the construction of a high-quality integration. For example, techniques have been developed for the case where $MD_{S \to T}$ includes instance-level data, where these instances are typically provided by expert users, e.g.,[14]. Further work has sought to support the debugging of schema mappings [4], or to steer the mapping generation algorithm using feedback [15]. In this paper, we compare Dynamap in experiments with ++Spicy [16], which presents an environment for developing mappings, underpinned by an approach to generating mappings that extends that of Clio in several ways, including in relation to target constraints, e.g., keys [17].

Given a set of mappings, *data exchange* provides techniques for evaluating these mappings in ways that minimize redundancy in the target [18], where the redundancy results from the presence of multiple mappings that share source and target tables. The approach to minimizing redundancy (viz., computing the core) may form part of mapping evaluation (e.g., [19, 20]), or involve transformations to the mappings (e.g., [16, 21]). As such, data exchange relates to mapping evaluation, and not to mapping generation, and thus data exchange techniques can be used with different mapping generation algorithms. Although in this paper the generation algorithm is cast in terms of algebraic operators, these can be translated for evaluation using existential

rules that are implemented using the chase procedure, and indeed we have an implementation of Dynamap that generates Vadalog [22]. Data exchange has been investigated for different mapping languages, including those with target constraints [18].

In contrast with work on *Schema mappings for databases*, in Dynamap, S is expected to contain data from independent sources, and as a result MD_S does not include declared keys and foreign keys. Thus Dynamap has to contend with less dependable relationships between source tables, and must scale to generate mappings that correlate data from potentially numerous sources.

Schema mappings in the *wild*. As discussed above, and, e.g., in [6], Clio primarily support experts in the development of mappings between a single source and a single target, e.g., building on declared foreign keys between source tables. In practice, this means that mapping generation can benefit from precise and exhaustive descriptions of relationships within the source schemas, as well as human-curated matches between the source and the target. In contrast, *mapping generation in the wild* must contend with arbitrary numbers of source schemas, where there may be no declared relationships between the tables in the source schemas. As a result, a focus for mapping generation research has shifted toward managing the resulting uncertainty.

For example, UDI [23] describes an approach to generating mappings for a mediated schema that is automatically inferred from the underlying data sources. As a result, there is no pre-existing target schema, and the technical focus is on aligning source attributes with counterparts to yield a mediated schema, and not so much on the generation of mappings that combine data from different sources. Also seeking to operate at web scale, Mahmoud and Aboulnaga [24] cluster single table sources, and then map keyword queries to the domains represented by the clusters. In such approaches, there is an attempt to provide some measure of integration with little additional information about the user's requirements (e.g., T may not be provided) and with little additional information about the sources (e.g., MD_S may be minimal). However, there is little evidence on how to combine data from different sources, and thus mappings tend not to be expressive. In contrast with these results, Dynamap builds on profiling data to inform the generation of more expressive mappings (involving union, join and outer-join), and thus has both to explore the resulting search space efficiently and to infer profiling

data for intermediate mappings.

The challenge of discovering tables that are related to a target has been addressed by recent work, e.g., [25, 26, 27, 28]. A common approach is to detect if attribute values coming from different sources are part of the same domain. Based on this, it is determined whether the sources are candidates to be joined or unioned to populate a target. This research addresses the lack of declared relationships between the sources that could be found in well-behaved schemas, e.g., foreign keys. This work is on a problem we share, but the focus is on inferring relationships between the sources w.r.t. to a target, which complements the focus in Dynamap on using (inferred) relationships to build mappings between multiple sources and a target.

Given that the research focus has shifted to generating *best-effort* mappings, several proposals inform mapping generation in the wild using feedback on results of candidate mappings (e.g., [29]). As such, MD_T includes tuples annotated as true positives, false positives or false negatives. In such work, there is no assumption that there is a correct mapping to be found, but rather alternatives are generated and scored by users based on the suitability of their results. This work also complements Dynamap, in that feedback could be collected on the mappings generated by Dynamap, for example to identify which are the most relevant to the user.

There is some other work that seeks to combine data from numerous structured data sources. For example, Data Tamer [30] targets the integration of enterprise data sets. However, the integration effort is focused less on schema mapping than on the instance-level, through entity resolution and fusion, with ongoing human input, for example in the form of training data. The goals of the Data Civilizer project seem similar, although with a greater emphasis on discovery and cleaning [31]. To date there are few details on mapping generation in Data Civilizer, although alternative join paths are associated with quality metrics, which may be presented to users. In AutoPipeline[32], data integration pipelines are generated automatically, informed by instance-level data used and produced by Python programs. This work reflects the requirement for efficient data integration for data scientists, but uses evidence that, in contrast with profiling data, is not always readily available.

The work described in this paper has in common with the work on *mapping generation for databases* the fact that we assume a target schema is given, and that we generate expressive mappings (i.e., that include project, union, join and outer join operations). On the other hand, $|S|$ can be greater

than 1 as we assume the data can come from various sources with different schemas. In addition, we do not depend upon declared keys and foreign keys in MD_S, and instead make use of a wider range of (less dependable but more widely available) results from data profiling [7]. In contrast with most previous work related to *mapping generation in the wild*, we combine tables using expressive mappings. This seems impractical without some additional constraints on the problem, so instead of creating a mediated schema we assume that the target T is given, and that we have access to profiling data on sources [7].

Mapping generation as search. Other proposals view mapping generation as a search problem, using either generic or bespoke strategies. In relation to generic strategies, Tupelo [33] is a mapping discovery algorithm that performs search within the transformation space of example instances based on a set of mapping operators. These operators combine to create complex mappings that carry out structural transformations or manipulate the data by creating relationships between schema components, e.g., attributes. The mapping discovery is done using only the syntax and structure of the input examples using a best-first search. Clio [1] and (++)Spicy [17] do not use a classical search strategy. These are custom to solving the problem of mapping generation. Their approaches are based on using key and foreign key constraints from the source or/and the target schemas so as to combine the sources with a view to satisfying the schema of the target subject to constraints on it. For example, ++Spicy uses *egds* to join the sources in various ways such that the target key constraints are populated with unique data values. In relation to Dynamap, these approaches are complementary; we contribute to work on mapping generation as search by scaling mapping generation to large numbers of sources through pruning the search space, and inferring profiling data on candidate mappings so that we have the same types of profiling data for generated mappings as for profiled sources without materializing any mappings.

3. Problem Statement

In this section we provide more details on the problem to be tackled in this paper. *The process of mapping generation is formalized as* $MapGen(S, T, MD_S, MD_T, MD_{S \to T}) \to M$, *where* M *is a set of generated mappings,* S *is a collection of source tables, from potentially many data sources,* T *is the*

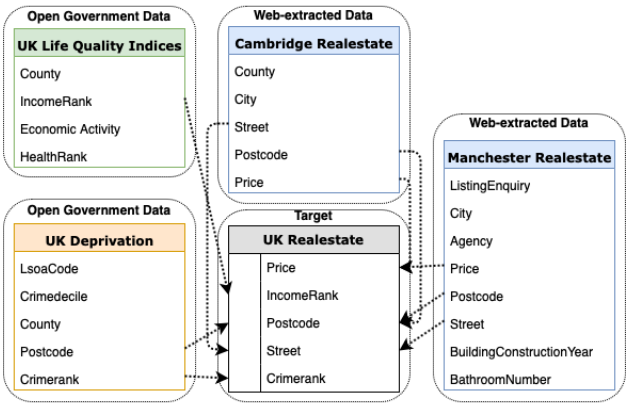


Figure 1: Mapping generation scenario for a simple target

target schema, MD_S is metadata on the source tables, in particular candidate keys and (partial) inclusion dependencies, MD_T is metadata on the target tables in the form of schema constraints, including key and foreign key constraints, $MD_{S \to T}$ is metadata comprising source-to-target matches. Typically, MD_S and $MD_{S \to T}$ will be generated by data profiling and schema matching software, respectively. The generated mappings are selected on the basis of their fitness, preferring mappings that provide many tuples with few nulls.

In the remainder of this section we will pin down the challenges of mapping generation *in the wild* that we aim to address in this work. To describe these, we use two running examples throughout the rest of the paper, and illustrate the applicability of mapping generation for them. These are depicted in Figures 1 and 6. Both examples are from the real-estate domain, exhibiting the same information and requiring in the target almost the same data. Figure 2 contains an example of source tables from the two scenarios. Each scenario exhibits different challenges for our tackled problem:

Simple target. Let us consider the scenario in Figure 1. The figure depicts four source relations with two real-estate data sources from *Manchester* (MA) and *Cambridge* (CA), and two sources showing quality of life indices, e.g., *UK Deprivation (UKD)* and *UK Life Quality Indices (UKQ)*. The sources do not have any declared relationships between one another as each has a different origin: the two real-estate agencies are web-extracted data, and the other two are open-government data supplied by different pub-

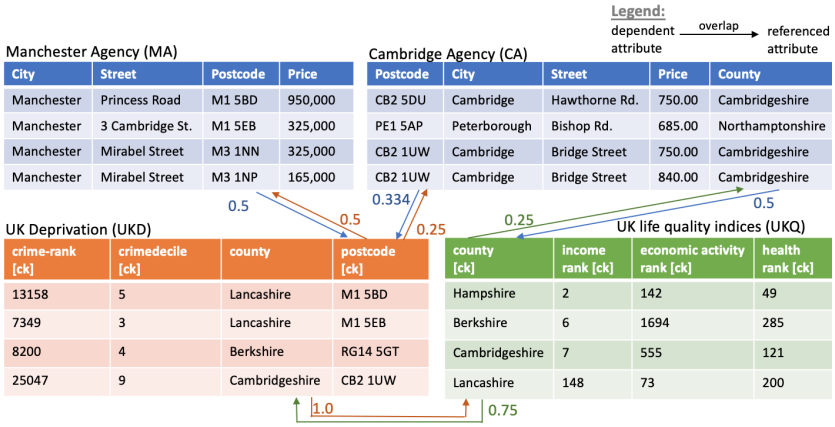


Figure 2: Example data with profiling data for sources

lic bodies. By *simple target* we mean a single-relation target schema with no constraints: the chosen target has a schema with the table *UK Realestate* (UKR), which is to be populated with values from the four source relations. As a result, we need to combine the sources in ways that are suitable for populating the target.

Challenges. The challenges raised by this scenario for the mapping generation algorithm arise from the fact that there are no declared join paths so there is no defined way of joining the sources. In Dynamap, we consider how sources can be merged (using union, join or outer-join) to populate a target table. However, there could be many ways of merging the sources, so there is a question of which is the most suitable mapping for populating the target. For example, in this scenario, the sources could be merged by *i)* first unioning MA and CA and then joining their result with UKD (on *postcode*) and UKR (on *county*); or *ii)* join CA with UKD (on *postcode*), MA with UKD (on *postcode*), union their result, and then finally join with UKQ (on *county*), etc. Decisions on which sources to join and how are informed by profiling data, as illustrated in Figure 2, however mapping generation must also decide how to join sources with candidate mappings, or how to join candidate mappings with each other. This requires that profiling data is able to be inferred for mappings. Furthermore, there is a combinatorial explosion in the number of ways in which tables can be combined, leading to the need to prune the search space without discarding promising candidates. These challenges are addressed in Section 4.

Complex target. Let us consider the scenario in Figure 6. The figure depicts three source relations where each contributes different attributes to the target: one real-estate data source (MA) and two sources showing quality of life indices (UKD and UKQ). The sources are the same as the corresponding ones in the *simple target* scenario. In this scenario, the chosen target contains two tables, *Area Info* (AI) and *UK Realestate* (UKR), with primary keys and that share a foreign key constraint.

Challenges. Similarly to the *simple target* scenario, we need to combine the sources so as to obtain correlated data to populate the target; exploring the space of ways of combining the sources is the same as for a *Simple target*. The challenges created by this scenario come from the fact that the mapping generation algorithm could create mappings that violate the target constraints, thus, they need to be taken into consideration. However, for mapping generation *in the wild*, one cannot expect to have sources that merge without violating the constraints, so the mapping generation algorithm must aim to generate mappings that satisfy the target constraints as much as possible. Note that, although a target schema may specify key or foreign key constraints, the data obtained from the underlying sources may not conform to such constraints. We address these challenges in Section 5.

4. Mapping Generation for a Simple Target

In this section we describe a proposal for schema mapping generation between a (set of) source schema(s) and a simple target. By *simple target*, we mean a target schema with a single target relation and no constraints.

The challenges identified in Section 3 for the *simple target* are addressed through four research contributions: *(i)* a mapping generation search algorithm based on the dynamic programming paradigm; *(ii)* a method for merging autonomous sources based on inferred relationships using profile data; *(iii)* a method for propagating profile data to intermediate mappings that result from merging other mappings in the same search space; and *(iv)* a technique that keeps the search space contained through a set of pruning strategies.

4.1. Algorithm Overview

Figure 3 illustrates the components that contribute to mapping generation on a *simple target*, which are numbered as follows:

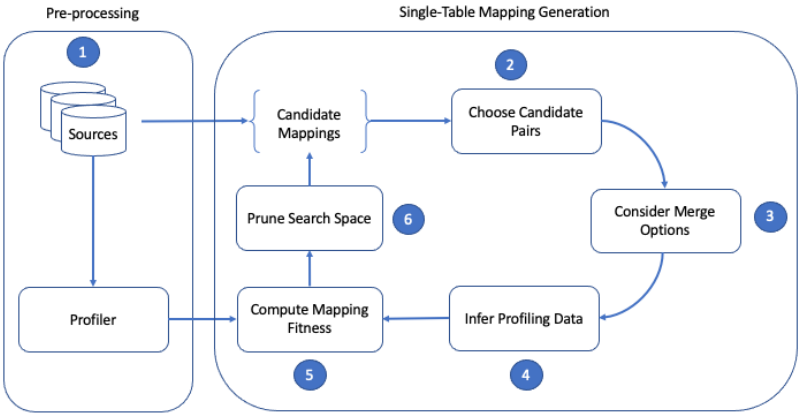


Figure 3: Dynamap for a Single Relation Target

1. *Pre-Processing.* The mapping generation process acts on a set of tabular sources, with limited metadata. Each of these sources s that has at least one match with the target gives rise to a single base *Candidate Mapping* of the form $\pi_A(s)$, where each element $a_s \in A$ is either *i)* a match to the target, $a_s \rightarrow a_t$, where a_s is an attribute in the source and a_t is a matching attribute in the target; or *ii)* a_s is an attribute that shares profiling data with other sources, i.e., may enable a merge at some point in the search.

   The input tables are also subject to profiling that infers candidate keys and inclusion dependencies between sources. For the running example, (partial) inclusion dependencies are illustrated in Figure 2. In addition, database statistics are computed for the sources, i.e., relation sizes, number of nulls and number of distinct values in the attributes. More details about the profiling data, metadata and statistics are provided in Section 4.2.

   For example, for the sources and target in Figure 1, there are the following base mappings:

$\pi_{IncomeRank,County}(UKQ)$
$\pi_{Postcode,Crimerank,County}(UKD)$
$\pi_{Price,Postcode,Street,County}(CA)$
$\pi_{Price,Postcode,Street}(MA)$

2. *Choose Candidate Pairs.* Pairs of candidate mappings can be merged using *union*, *join* or *outer-join* to produce further candidate mappings. For example, given the source tables in Figure 2, all pairs of base mappings from these source tables can be considered for merging: $\{(MA, CA), (MA, UKD), (MA, UKQ), (CA, UKD), (CA, UKQ), (UKD, UKQ)\}$. The resulting mappings can then be considered for merging with the base mappings and each other. The space of potential pairs is explored using Dynamic Programming, as described in Section 4.3.

3. *Consider Merge Options.* Given a pair of candidate mappings from the previous step, it is possible that they can be combined by *union*, *join* or *outer-join*. For example, the base mappings for UKQ and MA are candidates to be joined on *postcode* because profiling can identify that *Postcode* is a candidate key for UKD and there is a (partial) inclusion dependency between the *Postcode* attributes in the two tables. The result of this step is referred to as an *intermediate mapping*. The details on how operators are selected for merging mappings are provided in Section 4.4.

4. *Infer Profiling Data.* Profiling data is inferred for the intermediate mappings. Note that this includes (partial) inclusion dependencies with all sources and all candidate mappings. The new profile data is needed to *i)* compute the fitness for the mapping, and *ii)* find merge opportunities between the new mapping and the other already-generated mappings. For example, we need to know inclusion dependencies for the *Postcode* of the intermediate mapping that joins UKQ and MA, as we need to know if this mapping can be joined with UKD and/or CA. The derivation of profiling data is detailed in Section 4.5.

5. *Compute Mapping Fitness.* The inferred profiling data is then used to compute a fitness value for the mapping. When there is more than one way of combining the same tables, this is used to select a preferred approach, and the fitness is also used to select the top k overall mappings. The fitness function is defined in Section 4.6.

6. *Prune the Search Space.* When new intermediate mappings are produced, it is possible that these mappings have characteristics that allow

the search space to be pruned. For example, a new mapping may subsume an existing one. Cases where candidate mappings can be removed from further consideration are discussed in Section 4.7.

Many plausible candidate mappings may be identified. As a result, there is a need to select a subset of these mappings. In the current approach, Dynamap outputs the best k mappings where k is an integer. The output mappings merge subsets of i source relations, $1 \leq i \leq n$, where n is the total number of input source relations, which were obtained during the dynamic programming search, and that are ranked according to their fitness.

4.2. Pre-Processing

Before mapping generation, a pre-processing step generates profiling data for each source. The profile data includes statistics, such as the cardinality of each relation, the number of distinct values for each attribute, and the number of nulls for each attribute, as required for a fitness function to choose between candidate mappings. Furthermore, access to the following profiling data over the input sources (which could be produced using a tool such as Metanome [9] or SINDY [34]) is assumed:

candidate keys – a column (or a combination thereof) that has unique values in the relation in which it occurs;

(partial) inclusion dependencies – given two projections R and S with identical arity over relations R' and S', resp., we define the inclusion dependency $I_{R,S} = R \subseteq_{\theta_{R,S}} S$, where $\theta_{R,S}$ represents the overlap of values between attributes R and S, i.e., the ratio of distinct values from R included in the values of S:

1. if $R \cap S = \emptyset$, then $\theta_{R,S} = \theta_{S,R} = 0$, and, based on profiling evidence, we say that R and S are disjoint and there is no inclusion dependency.
2. if $R \cap S = R$, then $\theta_{R,S} = 1.0$, and we say that, based on profiling evidence, $R \subseteq S$ and there is a *(total) inclusion dependency* from R to S.
3. if $R \cap S \neq R$ and $R \cap S \neq S$, then $\theta_{R,S} = \frac{V(R \cap S)}{V(R)}$, where $V(X)$ denotes the number of distinct values in attribute X. If $0 < \theta_{R,S} < 1$, then the inclusion dependency is *partial*. Note that $\theta_{R,S} \neq \theta_{S,R}$.

In addition to profiling data, we assume we have access to *matches*. In our setting, a *match* is an association between a source attribute a_s and a target attribute a_t, such that a_s is a candidate to provide values for a_t. In support of mapping generation in the wild, we assume that matches can be

inferred [8]. Furthermore, in this section, as mapping generation combines tables using union, join and outer-join operations, a *mapping* is a relational algebra query over the source tables that projects attributes that belong to the target; relational algebra also provides a foundation for the propagation of profiling data.

4.3. Dynamic Programming for Choosing Candidate Pairs

Dynamic programming is a method that divides a complex problem into a collection of simpler sub-problems, and then combines the sub-solutions into a solution to the original compound problem. We use dynamic programming for mapping generation as it systematically explores the search space, starting with single-table mappings, and progressing incrementally to mappings that involve more and more tables. In this context, we use the term *merge* to represent the application of *union*, *join* or *outer-join* operators to tables or mappings to yield a new mapping. For mapping generation, the *dynamic programming* method is applied as follows.

- The *compound problem* is finding mappings involving multiple input relations with attributes that match the same target relation. The compound problem of merging multiple input relations is divided into sub-problems that involve pairs of subsets of the input relations, and then merging the results from each pair of sub-problems. For example, for the tables in Figure 2, the compound problem is to generate a mapping that combines the tables MA, CA, UKD and UKQ.
- A *sub-problem* involves trying to find a mapping for fewer relations than in the initial input. Each sub-problem represents an iteration in the mapping generation process. Given N initial source relations, in each iteration i, $1 \leq i \leq N$, the algorithm searches for the best way to merge any i source relations. The mappings that result from combining each subset of source relations characterize new relations that are referred to as *intermediate mappings*, the collection of which comprises the solution at iteration i. For example, for the compound problem involving the 4 tables in Figure 2, the immediate sub-problems, in iteration $i = 3$, involve generating mappings for the following sets of 3 tables: $\{MA, CA, UKD\}$, $\{MA, CA, UKQ\}$, $\{MA, UKD, UKQ\}$, $\{CA, UKD, UKQ\}$.

The mapping generation search starts with the recursive method of the algorithm, GenerateMappings, listed in Algorithm 1. It is first called with

Algorithm 1 Mapping generation - the recursive method of *dynamic programming*

```
 1: function GENERATEMAPPINGS(i)
 2:     if sub_solution[i] exists then
 3:         return sub_solution[i]
 4:     else
 5:         iteration_maps ← []
 6:         for j ← 1, ceil(i/2) do
 7:             b1 ← GENERATEMAPPINGS(j)
 8:             b2 ← GENERATEMAPPINGS(i − j)
 9:             new_maps ← MERGEMAPPINGS(b1, b2)
10:             iteration_maps.add(new_maps)
11:         sub_solution[i] ← iteration_maps
12:     return sub_solution[i]
```

i set to N, the total number of source relations. When running GENERATEMAPPINGS for iteration i, the sub-solutions from iterations j and $(i-j)$ are merged (lines 6-10). The resulting merged mappings for this iteration are *memoized* as sub-solutions for iteration i (line 11) so that this sub-solution is reused in subsequent GENERATEMAPPINGS calls (lines 2-3). For $i = 1$, the sub-solution represents the set of mappings where a mapping is generated for each input relation that can (partially) populate t. This sub-solution represents the base solution which is generated before the first call of GENERATEMAPPINGS.

After iteration N of Algorithm 1, a set of mappings which merge all or subsets of the initial source relations is obtained. The schema of the output mappings is that of the target relation.

Example 1. *For the example in Figure 1, where N←4, in the last iteration* $i \leftarrow 4$*, the algorithm tries to merge the mappings from iteration 3 with the mappings from iteration 1, and then pair-wise merge the mappings from iteration 2. For example, assume that in iteration 2, the following mappings were found (n.b., these are not the complete set), where* merge *abstracts over the specific operation used to combine its operands:*

$m_{2,1} \leftarrow merge(MA, CA)$
$m_{2,2} \leftarrow merge(UKQ, UKD)$
$m_{2,3} \leftarrow merge(CA, UKD)$

Then, in iteration 4, GENERATEMAPPINGS *tries to merge each of the*

Algorithm 2 Merge pairwise the mappings from 2 sets of mappings

```
1: function MERGEMAPPINGS(batch1, batch2)
2:     new_maps ← [ ]
3:     for each map_i in batch1 do
4:         for each map_j in batch2 do
5:             operator ← CHOOSEOPERATOR(map_i, map_j)
6:             if operator not null then
7:                 new_map ← NEWMAPPING(operator)
8:                 md ← COMPUTEMETADATA(new_map)
9:                 if ISFITTEST(new_map) then
10:                    new_maps.add(new_map)
11:    return new_maps
```

mappings with the other: ($m_{2,1}$, $m_{2,2}$), ($m_{2,2}$, $m_{2,3}$) and ($m_{2,1}$, $m_{2,3}$). Notice that by merging $m_{2,1}$ with $m_{2,2}$, a mapping that covers all the input sources is obtained.

MERGEMAPPINGS (Algorithm 2) is called by GENERATEMAPPINGS in line 9. Its purpose is to combine batches of mappings from sub-solutions. Specifically, given pairs of mappings *map_i* and *map_j* from batches of mappings from two iterations, MERGEMAPPINGS calls CHOOSEOPERATOR (line 5) to identify if *map_i* and *map_j* can usefully be merged. If so, then, on lines 7-8, NewMapping builds a new intermediate mapping for the chosen operator, and ComputeMetadata computes its metadata, i.e., fitness value, and profile data. IsFittest checks if the intermediate mapping has the highest fitness of any mapping involving the same initial sources, if so, it is retained (lines 9-10).

4.4. Consider Merge Options

In this section, we describe how Dynamap decides which operator to use for merging intermediate mappings. The output of this component is either a relational operator (one of *union, join,* or *full outer join*) or *null* if no merge opportunity is found. Whilst there is always a possible merge, not all merges offer opportunities to properly populate the target. Because of this, the algorithm checks whether a candidate merge satisfies a set of conditions, and whether these conditions suggest that the merge would correlate the data generated by the two input mappings and therefore be suitable for populating the target.

We call this component CHOOSEOPERATOR and formalize it in Algorithm 3. This method takes as input two parameters, viz., two (base) intermediate

Algorithm 3 Choose suitable merge operator

```
1: function ChooseOperator( map1, map2)
2:     \\ t_rel is the target relation and it's a global variable
3:     map1_ma ← FindMatchesAttr(map1, t_rel)
4:     map2_ma ← FindMatchesAttr(map2, t_rel)
5:     operator ← null
6:     if DiffMatches(map1_ma, map2_ma) then
7:         operator ← ChooseOperatorDiff(map1, map2)
8:     else
9:         operator ← Union(map1, map2)
10:    return operator
```

mappings (*map1* and *map2*), and operates in the global state through two pieces of information, viz., the target relation(t_rel) and the profile data (pd).

ChooseOperator decides how to combine two intermediate mappings by considering how these relate to the given target table. Specifically, the sets of matched target attributes are retrieved for each mapping w.r.t. the target relation by a call to FindMatchesAttr[1] (lines 3-4). Each input mapping will have a corresponding set of matched target attributes. Matching different target attributes means that the two sets of matched target attributes, i.e., *map1_ma*, and *map2_ma*, may or may not be disjoint, i.e., they either *(i)* both have matches that are for the same target attributes while also possibly having matches for different target attributes, or *(ii)* they match entirely different target attributes in the same target relation, t_rel. DiffMatches checks whether the matches are for different target attributes. If they are, they become candidates for joining, to be decided by ChooseOperatorDiff (line 7). If the matches are for the same target attributes, then the two mappings are unioned (line 9). Finally, on line 10, the output, i.e., either an operator that merges the two input mappings, or *null* if no merge was found, is returned.

Example 2. *In Figure 1, the matches with the target for both the MA and CA relations are postcode, price and street, so they are candidates for unioning. However, UKD has different matches, viz., postcode and crime_rank, to those of MA, and thus MA and UKD are candidates for joining.*

[1]In this paper, if a method name is in this font, then its explanation was omitted due to space limitations, but its purpose is briefly explained in the calling method's explanation.

Algorithm 4 Generate operator when two mappings match different target attributes

```
1: function ChooseOperatorDiff(map1, map2)
2:     \\ t_rel, pd(profile data) are global variables
3:     op ← null
4:     subsumedMap ← IsSubsumed(map1, map2)
5:     if subsumedMap not null then
6:         discard(subsumedMap)
7:         return op
8:     map1_keys ← FindKeys(pd, map1)
9:     map2_keys ← FindKeys(pd, map2)
10:    ind ← MaxInd(pd, map1_keys, map2_keys)
11:    if ind exists then
12:        if ind.overlap = 1.0 then
13:            op ← Join(map1, map2, ind.attributes)
14:        else
15:            op ← OuterJoin(map1, map2, ind.attributes)
16:    else
17:        map1_ind ← MaxInd(map1_keys, map2.attributes)
18:        map2_ind ← MaxInd(map2_keys, map1.attributes)
19:        ind ← MaxCoef(map1_ind, map2_ind)
20:        if ind exists then
21:            if ind.overlap = 1.0 then
22:                op ← Join(map1, map2, ind.attributes)
23:            else
24:                op ← OuterJoin(map1, map2, ind.attributes)
25:        else
26:            map1_mk ← FindMatchedKeys(map1, t_rel)
27:            map2_mk ← FindMatchedKeys(map2, t_rel)
28:            if SameMatches(map1_mk, map2_mk) then
29:                op ← OuterJoin(map1, map2, < map1_mk, map2_mk >)
30:    return op
```

ChooseOperatorDiff (Algorithm 4) decides which join operator to apply between pairs of mappings where the target attributes that are matched in one mapping are disjoint or only partially overlapping with the target attributes matched in the other. This method uses the same parameters as ChooseOperator.

In lines 4-7, IsSubsumed determines whether, on attributes that match the target, the profiling data has inclusion dependencies between an attribute in one mapping and a corresponding attribute in the other mapping. If so, the subsumed mapping is discarded from the set of kept mappings (for further

reference, in Section 4.3, we refer to the kept mappings as *memoized sub-solutions*) and *null* is returned. In lines 8-9, FindKeys retrieves the *candidate keys* from the profile data for both input mappings. Then, on line 10, MaxInd retrieves from the profile data the *(partial) inclusion dependency* (*ind*) with the highest overlap between a pair of keys from the two sets of candidate keys (*map1_keys* and *map2_keys*). If there is a pair of overlapping keys, i.e., if *ind* exists (line 11), then the overlap is checked:

- if $\theta = 1.0$, then the inclusion dependency is total and the chosen operator is *join* because a foreign key relationship is inferred between two mappings on their candidate key attributes (lines 12-13);
- if $\theta \in (0, 1.0)$, then the inclusion dependency is partial and the operator is a *full outer join* because a foreign key relationship cannot be inferred so the algorithm joins the tuples that can be joined and keeps the remaining data (lines 14-15).

In both cases, the join condition is built from the key attributes involved in the chosen inclusion dependency.

If there is no overlap between the pairs of keys, the algorithm tries to infer a foreign key relationship between a candidate key from one relation and attributes of the other relation that may not be candidate keys (lines 17-18). If there are several (partial) inclusion dependencies, MaxCoef compares them and chooses the one with the highest overlap (line 19). If such an inclusion dependency exists, the type of merge is decided by the overlap level, as before.

Next, if a foreign key relationship cannot be inferred, then, on lines 26-27, FindMatchedKeys retrieves the candidate keys from both mappings that match target attributes and checks if they match the same target attributes (line 28). If they do, then the two mappings are merged using *full outer join*, where the join condition is on the attributes that meet the requirements (line 29). The intuition behind this last step is that even if there is no overlap between the attribute values of the two mappings, it could be that there is *instance complementarity* between the two mappings, in which case performing a *full outer join* vertically aligns the key attributes that match the same target attributes.

4.5. Profiling Data Propagation

In searching the space of candidate mappings, Dynamap requires metadata about these mappings that indicates how they relate to each other (e.g.,

can they be joined or unioned), and a means of comparing their fitness. Dynamap assumes the availability of profiling data, in the form of cardinalities (to compute mapping fitness), and in the form of keys and inclusion dependencies (to inform how mappings can be combined). Such profiling data can be obtained for source data sets using a profiling tool such as Metanome [9], but must be derived for candidate mappings without the costly requirement to materialize and profile intermediate mappings.

These characteristics are computed from the profile data of the parent mappings and the operator used to combine the parent mappings. The result sizes and the numbers of distinct values returned by relational operators can be estimated using established techniques (e.g., [35]). However, estimating properties of the relationships between mappings in the wild is the subject of active investigation; for example, recent results have described probabilistic approaches to estimating the unionability [27] and joinability [26] of attributes in large data sets, indexed using Locality Sensitive Hashing. Such solutions approximate relationship measures, e.g., overlap or containment, between attributes using special hash functions applied on their extents. In this paper, given that we do not materialize intermediate results to obtain the attribute extents, we cannot use such hash-based approximation techniques and, therefore, our focus is on propagating profiling data from source tables through mappings. We propagate the profile data using specific formulas for each type of merge, i.e., *lossy* and *lossless* merges, and we generate profile data for new candidate mappings, including inferring the relationships between the candidate mappings and other source tables.

4.5.1. Candidate Keys and Inclusion Dependencies

This section details how profile data in the form of candidate keys and (partial) inclusion dependencies are propagated to the results of the algebraic operators used in mappings, and thus how profile data can be propagated to new candidate mappings.

Given an inclusion dependency $S \subset_{\theta_{S,Q}} Q$, where S, Q are attributes in different mappings, the inferred inclusion dependency is of the form $R \subset_{\theta_{R,Q}} Q$, where R is an attribute in the newly created intermediate mapping and given that attribute S is a parent attribute of R. Given an inclusion dependency $Q \subset_{\theta_{Q,S}} S$, the inferred inclusion dependency is of the form $Q \subset_{\theta_{Q,R}} R$, given that S attribute is a parent attribute of R.

Tables 1 and 2 show the formulas for inferring overlaps (θ) when both attributes from both parents are involved in the inclusion dependency, and

	Dependent	**Referenced**	**Conditions**	**Overlap(s) for inferred inclusion dependency**
1	$S(parent)$	$P(parent)$		$\theta_{R,P} = \frac{V(P)}{V(R)}$, $\theta_{S,R} = 1$
2	$S(parent)$	Q	$\theta_{S,P} = 0$	$\theta_{R,Q} = \frac{V(P)*\theta_{P,Q}+V(S)*\theta_{S,Q}}{V(R)}$
3			$\theta_{S,P} = 1$	$\theta_{R,Q} = \frac{V(P)*\theta_{P,Q}}{V(R)}$
4			$\theta_{P,S} = 1$	$\theta_{R,Q} = \frac{V(S)*\theta_{S,Q}}{V(R)}$
5			$\theta_{S,Q} = 1$	$\theta_{R,Q} = \frac{V(S)-V(S)*\theta_{S,P}+V(P)*\theta_{P,Q}}{V(R)}$
6			$\theta_{P,Q} = 1$	$\theta_{R,Q} = \frac{V(P)-V(P)*\theta_{P,S}+V(S)*\theta_{S,Q}}{V(R)}$
7			$\theta_{Q,P} = 1$ or $\theta_{Q,S} = 1$	$\theta_{R,Q} = \frac{V(Q)}{V(R)}$
8			$\theta_{P,Q} = 0$	$\theta_{R,Q} = \frac{V(S)*\theta_{S,Q}}{V(R)}$
9			$\theta_{Q,S}$, $\theta_{S,Q}$, $\theta_{Q,P}$, $\theta_{P,Q}$, $\theta_{P,S}$, $\theta_{S,P} \in (0,1)$	$\theta_{R,Q} = \frac{V(S)*\theta_{S,Q}+V(P)*\theta_{P,Q}-V(S)*\theta_{S,P}}{V(R)}$
10	$X_1(parent)$	Q		$\theta_{X,Q} = \theta_{X_1,Q}$
11	Q	$S(parent)$	$\theta_{S,P} = 0$	$\theta_{Q,R} = \frac{V(P)*\theta_{P,Q}+V(S)*\theta_{S,Q}}{V(Q)}$
12			$\theta_{S,P} = 1$	$\theta_{Q,R} = \frac{V(P)*\theta_{P,Q}}{V(Q)}$
13			$\theta_{P,S} = 1$	$\theta_{Q,R} = \frac{V(S)*\theta_{S,Q}}{V(Q)}$
14			$\theta_{S,Q} = 1$	$\theta_{Q,R} = \frac{V(S)-V(S)*\theta_{S,P}+V(P)*\theta_{P,Q}}{V(Q)}$
15			$\theta_{P,Q} = 1$	$\theta_{Q,R} = \frac{V(P)-V(P)*\theta_{P,S}+V(S)*\theta_{S,Q}}{V(Q)}$
16			$\theta_{Q,P} = 1$ or $\theta_{Q,S} = 1$	$\theta_{Q,R} = 1$
17			$\theta_{P,Q} = 0$	$\theta_{Q,R} = \frac{V(S)*\theta_{S,Q}}{V(Q)}$
18			$\theta_{Q,S}$, $\theta_{S,Q}$, $\theta_{Q,P}$, $\theta_{P,Q}$, $\theta_{P,S}$, $\theta_{S,P} \in (0,1)$	$\theta_{Q,R} = \frac{V(S)*\theta_{S,Q}+V(P)*\theta_{P,Q}-V(S)*\theta_{S,P}}{V(Q)}$
19	Q	$X_1(parent)$		$\theta_{Q,X} = \theta_{Q,X_1}$

Table 1: Inclusion dependencies propagation for *lossless* attribute merges

when one of the parents is involved in an inclusion dependency with another attribute. The tables differ based on whether the merge of two parent attributes, S and P, causes the loss of distinct values to the attribute R. In both tables, the propagation formula is chosen based to the satisfied conditions for the parent attributes (in **Conditions** column). Also, the notation $V(X)$ denotes the number of distinct values in attribute X.

The formulas in Tables 1 and 2 are derived based on set operations using the following general expressions:

- to infer the number of distinct values in R:
  $V(R) = V(S) + V(P) - V(S \cap P)$ for *lossless* merges and
  $V(R) = V(S \cap P)$ for *lossy* merges
- to express the number of values contained by the intersection of two sets, e.g., S and P:
  $V(S \cap P) = \theta_{S,P} * V(S) = \theta_{P,S} * V(P)$
- to infer the number of distinct values in the intersection of R and Q:

	Dependent	Referenced	Conditions	Overlap(s) for inferred inclusion dependency
1	$S(parent)$	$P(parent)$	$\theta_{S,P} = 1$	$\theta_{R,P} = 1$, $\theta_{S,R} = 1$
2	$P(parent)$	$S(parent)$	$\theta_{S,P} = 1$	$\theta_{R,S} = 1$, $\theta_{P,R} = \frac{V(S)}{V(P)}$
3	Q	S *or* $P(parent)$	$\theta_{S,P} = 1$	$\theta_{Q,R} = \theta_{Q,S}$
4	S *or* $P(parent)$	Q	$\theta_{S,P} = 1$	$\theta_{R,Q} = \theta_{S,Q}$
5	$X_1(parent)$	Q	$\theta_{S,P} = 1$	$\theta_{X,Q} = \theta_{X_1,Q}$
6	Q	$X_1(parent)$	$\theta_{S,P} = 1$	$\theta_{Q,X} = \theta_{Q,X_1}$
7	$Y_2(parent)$	Q	$\theta_{S,P} = 1$	$\theta_{Y,Q} = \begin{cases} 1, & if\ \theta_{Q,Y_2} * V(Q) > \lvert r \rvert \\ \frac{\theta_{Y_2,Q} * V(Y_2)}{\lvert r \rvert}, & otherwise \end{cases}$
8	Q	$Y_2(parent)$	$\theta_{S,P} = 1$	$\theta_{Q,Y} = \begin{cases} \frac{\lvert r \rvert}{V(Q)}, & if\ \theta_{Q,Y_2} * V(Q) > \lvert r \rvert \\ \theta_{Q,Y_2}, & otherwise \end{cases}$

Table 2: Inclusion dependencies propagation for *lossy* attribute merges

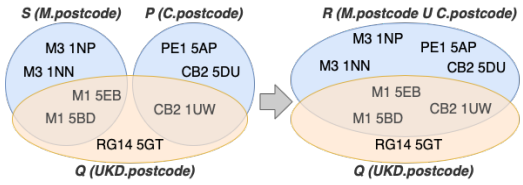


Figure 4: Example for propagating inclusion dependencies

$$V(R \cap Q) = V(S \cap Q) + V(P \cap Q) - V(S \cap P \cap Q)$$

- to infer the overlap values between R and Q:
  i) $\theta_{R,Q} = \frac{V(R \cap Q)}{V(R)}$
  ii) $\theta_{Q,R} = \frac{V(R \cap Q)}{V(Q)}$.

Example 3. *In Figure 2, assume the union of Manchester (M) and Cambridge (C) sources, i.e., a new mapping:*

$\boldsymbol{r} \leftarrow \pi_{postcode,street,price}(M) \cup \pi_{postcode,street,price}(C)$

is created without any loss of values. The merge on postcode attributes is represented in a new attribute $R \leftarrow M.postcode \cup C.postcode$. *Figure 4 shows the postcode distinct values in* M, C *and* UKD *(UK Deprivation) and their overlaps: there are overlaps between the deprivation postcode and the real-estate postcodes, e.g.,* $\theta_{C,UKD} = 0.33$, $\theta_{UKD,C} = 0.25$ *etc., but not between the postcode attributes of the two real-estate sources. Now that the two real-estate sources are merged, the overlaps between the newly created postcode attribute,* R, *and the deprivation postcode,* $UKD.postcode$, *needs to be computed, i.e.,* $\theta_{R,UKD.postcode}$, $\theta_{UKD.postcode,R}$.

Lossless merge. Table 1 shows how to estimate the overlap when the merge of the parent attributes is *lossless*, i.e., for the projected attributes in a *union* operation, or for the join condition attributes of a *full outer join* operation. By *union* we mean $r \leftarrow m_1 \cup m_2$, where r is the result of m_1 and m_2. We use the notation $r.R \leftarrow m_1.S \cup m_2.P$ to represent R as a new resulting attribute from the merge of S and P. Similarly, by *full outer join*, we mean $r \leftarrow m_1 \mathbb{D}\!\!\!\!\bowtie\!\!\!\!\subset_{S=P} m_2$, where $r.R \leftarrow m_1.S \cup m_2.P$. For both operations, the overlaps of the parents, i.e., $\theta_{Q,P}, \theta_{P,Q}, \theta_{S,Q}, \theta_{Q,S} \in (0, 1]$, where $Q \in schema(m_3)$, $m_3 \neq m_1, m_3 \neq m_2$, need to be propagated for the newly created attribute R with Q, i.e., compute $\theta_{Q,R}$ and $\theta_{R,Q}$.

In Table 1, through X_1 we generically represent a parent attribute that was not used in the join condition of a *full outer join*, i.e., its values are not merged with the values from the other parent mapping.

Lossy merge. Table 2 shows how to estimate the overlap when propagating the inclusion dependencies when the merge of the parent attributes is *lossy*, i.e., when the chosen operator is *join* and one of the relations may lose attribute values. By *join* we mean $r \leftarrow m_1 \bowtie_{S=P} m_2$, and $r.R \leftarrow m_1.S \cap m_2.P$, where $R \subseteq schema(r)$. Similarly to the *lossy* merge, the overlaps $\theta_{Q,R}$ and $\theta_{R,Q}$ beed to be computed.

In Table 2, through X_1 we generically represent a parent attribute in m_1 that was not used in the join condition, i.e., its values are not merged with any attribute values from m_2, and m_1 is the mapping that does not lose attribute values. Through Y_2 we generically represent a parent attribute in m_2 that was not used in the join condition, and m_2 is the mapping that potentially loses attribute values due to the merge.

Example 4. *Continuing Example 3, where M and C are unioned, the partial inclusion dependencies between these two relations and UKD need to be propagated to the newly created mapping:* $\boldsymbol{r} \leftarrow \pi_{postcode,street,price}(M) \cup \pi_{postcode,street,price}(C)$. *For union operations, the merge is lossless, so Table 1 is used for new overlaps. For the propagation of* $M.postcode \subset_{0.5} UKD.postcode$ *into* $r.postcode \subset_{\theta} UKD.postcode$ *the overlap estimation (*θ*) needs to be computed. In Table 1, the dependent attribute of the parent mapping is S, i.e., M.postcode, and the referenced attribute is not a parent, i.e., Q is UKD.postcode, while the other parent attribute P is C.postcode. The two parent attributes (M.postcode and C.postcode) are disjoint, thus the condition on the second row is satisfied (*$\theta_{S,P} = 0$*),* $\theta = \frac{3*0.334+4*0.5}{6} = 0.5$.

Operator	Conditions for propagating a candidate key (CK)
Union: $r.X \leftarrow m_1.X_1 \cup m_2.X_2$	- $r.X$ is CK if $m_1.X_1$ & $m_2.X_2$ are CKs and $m_1.X_1 \cap m_2.X_2 = \emptyset$
Join: $r \leftarrow m_1 \bowtie_{X_1=X_2} m_2$, where $X_1 \subseteq_1 X_2$, $Y_1 \in schema(m_1)$, $Y_1 \neq X_1$ $Z_2 \in schema(m_2)$, $Z_2 \neq X_2$	- $r.X$ is a CK if $m_1.X_1$ and $m_2.X_2$ are CKs. - $r.Z$ is a CK if $m_1.X_1$ is a CK - $r.Y$ is a CK if $m_2.X_2$ is a CK
Full Outer Join: $r \leftarrow m_1 \mathbb{D}\bowtie_{X_1=X_2} m_2$	- $r.X$ is a CK if X_1, X_2 are CKs

Table 3: Candidate keys propagation

Algorithm 5 Fitness function

```
1: function FITNESS(map)
2:     atts ← REMOVEOUTLIERS(map.attributes)
3:     attr_nulls = {⟨a, count(v)⟩|a ← atts, v ← a.values, v = null}
4:     max_attr_nulls = {max(n)|⟨a, n⟩ ← attr_nulls}
5:     return map.size − max_attr_nulls
```

Propagating candidate keys means detecting whether the unique constraint still holds. Candidate keys are identified if there is no possibility of duplicates or null creation. The conditions for propagating the candidate keys are depicted in Table 3.

4.6. Mapping Fitness

A fitness function is used to compare candidate mappings that share source tables. Different fitness functions could be used; here the fitness of a mapping is based on an estimate of the number of largely complete tuples it will return, an approach that was found to be effective in practice. FITNESS is listed in Algorithm 5. Specifically, given a mapping with a list of attributes, RemoveOutliers returns the set of attributes that are not outliers with respect to the number of nulls they contain (line 2). Outliers are identified using the *Median and Interquartile Deviation Method*. Then, for the remaining attributes, i.e., *atts*, the attribute predicted to have the most nulls is identified (lines 3-4). The number of largely complete tuples in the mapping is then estimated to be the cardinality of the mapping minus the number of nulls in the attribute with the most nulls. This fitness prefers mappings with larger results (thus retaining more source data) and with fewer nulls.

The fitness function can be replaced, to prefer mappings with other characteristics. Other options could prefer: the lowest ratio of estimated nulls, which would favor mappings with as few nulls as possible; the highest number of distinct values on matched attributes, which would favor mappings

that bring data from sources that are as disjoint as possible; the highest cardinality, which would favor mappings that merge data from (possibly) many sources; or the best coverage for the chosen target, which aims to populate as many attributes as possible. Our fitness function is primarily used to distinguish between alternative ways of combining the same source tables to populate the target. In this setting, the fitness function prefers mappings that provide many tuples with few nulls.

4.7. Pruning the Search Space

Mapping generation *in the wild* may have to contend with large numbers of sources. While it is possible to use dataset discovery techniques [28] to identify promising sources, the search space can still be problematic. In this section we first present the algorithmic complexity of the approach, and then describe techniques for reducing the number of candidate mappings considered within a search. The efficiency and impact of the proposed strategies is evaluated in Section 6.5.

4.7.1. Algorithm Complexity

As explained in Section 4.3, dynamic programming is the method we use for merging multiple smaller mappings to create larger mappings. For an input of N source relations, in each iteration i ($i \leq N$), the algorithm generates intermediate mappings that merge subsets of i relations.

Although the maximum number of memoized mappings for an iteration i is C_N^i, i.e., combinations of N relations taken i at a time, the algorithm tries to merge many more pairs of mappings than are actually memoized: for each iteration i, the algorithm tries to merge pairwise the mappings from previous iterations j and $i-j$, where $1 \leq j \leq \frac{i}{2}$. Considering the maximum number of mappings that can be generated in an iteration, and the number of pairs of sub-solutions that are merged to compute the mappings for an iteration, the algorithm makes a maximum total number of attempts at merging defined by $\sum_{i=1}^{N} \sum_{j=1}^{\frac{i}{2}} C_N^j C_N^{i-j}$.

The most complex operator search unfolds when searching for a way to merge two mappings through a join (Algorithm 4). The algorithm analyses all profiling data, i.e., inclusion dependencies and candidate keys for each mapping. The search for keys in each mapping (FindKeys in Algorithm 4) is a linear search as each attribute in the mapping is checked. The search for the inclusion dependency that has the highest overlap (MaxInd in Algorithm 4) reaches an upper limit when all the attributes of the two mappings are keys

and they pairwise share inclusion dependencies. In this worst scenario, when merging a mapping with a maximum number of m attributes with another mapping with the maximum number of attributes, the algorithm needs to parse $m \times m$ inclusion dependencies to pick the pair of attributes with the maximum overlap.

Thus, the overall algorithm complexity can be approximated as a function of the total number of sources (N) and the maximum number of attributes that a mapping can have (m):

$$o(N, m) = \sum_{i=1}^{N} \sum_{j=1}^{\frac{i}{2}} (C_N^j C_N^{i-j} m^2) \tag{1}$$

This upper limit would be reached if *i)* all the sources would be schema-complementary w.r.t. the target schema so they would be merged through joins, *ii)* all their attributes would be candidate keys, and *iii)* all attributes would pairwise share inclusion dependencies.

Even though this worst case behavior will not be encountered in practice, the combinatorics of the problem mean that the approach can only be practical if: *i)* the fraction of the mappings that can be combined by MergeMappings is small – the fraction is a property of the integration scenario; and *ii)* the search space is pruned to avoid the retention of less promising candidate mappings.

4.7.2. Preliminaries

Let $r, m_1, m_2 \in M$, where $r \leftarrow merged(m_1, m_2, t)$, M is the space of mappings, and t a target relation. Let $parents(r) \leftarrow \{m_1, m_2\}$, where m_1 and m_2 are the two mappings that merged when creating r.

Let $ancestors(r) \leftarrow \{m_1 ... m_n\}$, where m_i, $i \in [1, n]$, are the initial input relations that created r.

Let $mergeable(m, t)$ be the set of mappings with which a mapping $m \in M$ could possibly merge w.r.t. target t.

Let $joinable(m_1, m_2, t)$ and $unionable(m_1, m_2, t)$ be true if m_1 can join/union with m_2 w.r.t. target t, or false otherwise.

Let $\delta(U \subset_\theta V)$ be the *degree of degradation* associated with an inclusion dependency, showing how many times the overlap has been approximated through propagation. After a new inclusion dependency is propagated, the degree of degradation grows whenever a new overlap cannot be accurately

computed and must be approximated, otherwise the new degree of degradation is equal to the one of the inclusion dependency from which it was derived. For example, in Table 1 on rows 9 and 18, and in Table 2 on rows 7 and 8, the overlaps are approximated; the degradation increases by 1 if an inclusion dependency is propagated using any of these formulas.

Let $preserved(r, m_1, t)$ be the set of *preserved mappings* for child mapping r, parent m_1, and target t. We define the set of preserved mappings as the set of mappings with which a parent mapping had an opportunity to merge, and now those mappings are transferred as merge opportunities to the child mapping. Given a mapping n, where n is not a parent of r, with which the parent mapping m_1 has a merge opportunity ($n \in mergeable(m_1, t)$), then n may be a *preserved mapping* for the child mapping r if n has a merge opportunity with r ($n \in mergeable(r, t)$), as well. Also, the merge opportunity between them needs to be *as good as* the merge between m_1 and n. A merge is considered *as good as* the previous merge if the degree of degradation of the inferred inclusion dependencies to the child r does not increase. If the degree of degradation of the inferred *ind* ($\delta(r.a_1 \subset n.a_2)$) is equal with the degradation of the propagated *ind* ($\delta(m_1.a_1 \subset n.a_2)$), then the merge opportunity m_1 has with n is preserved under similar conditions between n and r, thus n is a *preserved mapping*. The set of *preserved mappings* for child mapping r, parent m_1, and target t is formalized:

$$\begin{aligned} preserved(r, m_1, t) = \{n | n \in mergeable(m_1, t) \wedge \\ n \in mergeable(r, t) \wedge n \notin parents(r) \wedge \\ \delta(r.a_1 \subset n.a_2) = \delta(m_1.a_1 \subset n.a_2)\} \end{aligned}$$

4.7.3. Pruning Techniques

In searching the space of candidate mappings, the sub-solutions produced by each call to GenerateMappings (Algorithm 1) are memoized, so that they can be reused in subsequent calls. As discussed in Section 4.7.1, the number of intermediate mappings can grow rapidly, which in turn increases the search space. This section identifies ways in which the search space can be pruned, by retaining only promising mappings.

Removing unnecessary parent mappings. After a merge, parent mappings are discarded if the child mapping has better fitness, while preserving the same merge opportunities as the parent. A parent mapping m_1 is discarded if:

- the child r has merge opportunities with all the mappings with which the parent can merge, i.e.,
  $mergeable(m_1, t) \subset preserved(r, m_1, t) \cup parents(r)$
- and the fitness of r is at least the same as the fitness of m_1:
  $fitness(m_1) \leq fitness(r)$

Example 5. *In iteration 2, after MA and CA are merged, mapping* $m_{2,1} \leftarrow union(MA, CA)$ *is created w.r.t. to target t. Now, Dynamap checks if the parents can be discarded. The conditions for removing unnecessary parent mappings are checked. First, Dynamap computes:*

$parents(m_{2,1}) = \{MA, CA\}$, $fitness(m_{2,1}) = 8$, $fitness(CA) = 4$

$mergeable(CA, t) = \{UKD, UKQ, MA\}$, $mergeable(m_{2,1}, t) = \{UKD, UKQ\}$

$preserved(m_{2,1}, CA, t) = \{UKQ, UKD\}$

Now it is checked if the same merge opportunities are preserved for $m_{2,1}$*, and this is true as*

$mergeable(CA, t) \subset \{preserved(m_{2,1}, CA, t) \cup parents(m_{2,1})\}$

and, in addition, $fitness(CA) \leq fitness(m_{2,1})$.

The inclusion dependencies between $m_{2,1}$ *and UKD are propagated using formulas in cases 2 and 11 in Table 1, and with UKQ they are propagated using formulas 8 and 17 in the same table. Their degradation did not increase as these overlaps were not approximated. As both conditions that are necessary for pruning are met, it can be concluded that CA can be discarded.*

Preventing creation of superfluous mappings. This pruning technique exploits the associativity and commutativity of union and join. Before a merge, the algorithm detects whether the mapping that would be generated is a superfluous variation of another mapping that was already memoized. Let r be a memoized mapping, where m_1 and m_2 are the current candidates for merging. The merge is superfluous if:

- r covers the same initial relations as m_1 and m_2:
  $ancestors(r) = ancestors(m_1) \cup ancestors(m_2)$
- r contains only *union* or only *join* operations, and that same type of operation would be used to merge m_1 and m_2:
  $operations(r) = operations(m_1) \wedge$
  $operations(r) = operations(m_2) \wedge$
  $(operations(r) = join \vee operations(r) = union)$

Building on the same operation properties, *viz.*, associativity and commutativity, we also prevent the generation of join or union mappings that

would otherwise become redundant in future, i.e., generated in subsequent iterations, equivalent or subsuming mappings. Let $m_1, m_2, m_3, r \in M$, where m_1 and m_2 are the candidates for merging. The merge is superfluous if:

- the candidate join operation between m_1 and m_2 can be applied in a subsequent iteration on the union of m_2 with another mapping m_3 and m_3 is also joinable with m_1 w.r.t. t:
  $joinable(m_1, m_2, t) \wedge joinable(m_1, m_3, t) \wedge unionable(m_2, m_3, t)$
- the candidate union mapping between m_1 and m_2 would become subsumed by a future chain-union:
  $parents(r) = \{m_2, m_3\} \wedge operations(r) = union \wedge unionable(r, m_1) \wedge unionable(m_1, m_2)$

Pruning subsumed *union* mappings. Previously generated mappings that are subsumed by a new mapping are discarded. In union-dominated scenarios, mappings that are created in early iterations can become subsumed in later iterations as the *union* operator gathers all their tuples in larger extents. This type of pruning most often discards the parent mappings. In this situation, the algorithm does not check whether the child has the same merge opportunities as no data is lost through the merge. A subsumed union mapping m is discarded upon the creation of a new mapping r if:

- the initial relations used in the creation of mapping m are included in the set of initial relations used for mapping r: $ancestors(m) \subset ancestors(r)$,
- and both m and r were created using only *unions*: $operations(m) = union \wedge operations(r) = union$.

5. Mapping Generation for a Complex Target

Here, we describe a proposal for schema mapping generation between a (set of) source schema(s) and a *complex target*. By complex target we mean a target schema with multiple relations that have primary and foreign key constraints. In terms of the challenges outlined in Section 3, we propose a method for generating mappings between source schemas that do not have any explicitly declared relationships, while aiming to satisfy target constraints involving primary keys and foreign keys.

Objectives of the approach include: *i)* to reuse the mapping generation algorithm for simple target relations from Section 4; and *ii)* to minimise the cost of mapping generation, by reducing the number of times simple target mapping generation must be called. To meet these objectives, we use

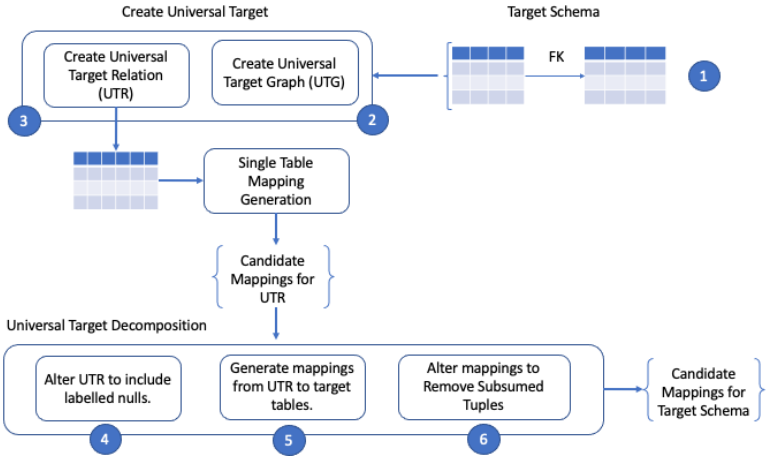


Figure 5: Dynamap for a Complex Target Schema

universal target relations (UTRs) that comprise all target attributes in tables linked through foreign keys, and then run simple target mapping generation for these target tables. A similar approach was taken by Clio [1], where they build several target tables that comprise the attributes of subsets of tables with foreign keys. Our approach is to use a single target table for each complete join path, and to generate mappings that partially populate the target from the generated UTRs. Thus, our method captures the use cases where the source schema can only partially populate the target, as does Clio. Our approach covers these cases by building the mappings in a bottom-up fashion, where the mappings generated at the beginning of the process are likely to only partially cover the target, while the mappings output in the last iteration are more likely to cover the entire target (if the pool of sources covers all target attributes). In addition to these cases, i.e., generating mappings between a single multi-table source schema and a multi-table target schema, our approach also operates *in the wild*, i.e., with sources that may come from multiple origins and whose relationships need to be inferred through profiling data.

5.1. Algorithm Overview

Figure 5, illustrates the components that contribute to mapping generation on a complex target, which are numbered as follows:

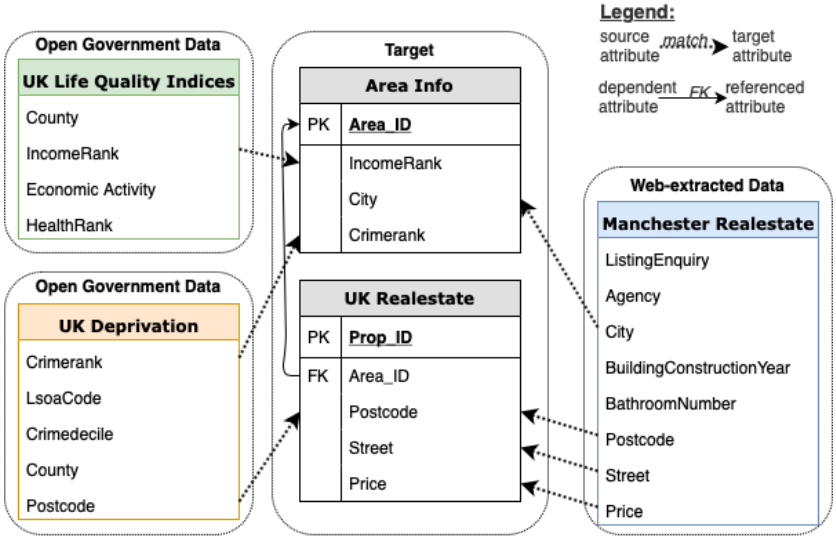


Figure 6: Mapping generation scenario for a complex target

1. *Target schema.* The mapping generation process aims to populate a relational schema that includes relationships represented as foreign keys.

2. *Create Universal Target Graph (UTG).* The UTG is a directed graph with a node for each table in the target schema, and an edge for each foreign key relationship, referring from the referenced to the dependent table. The UTG supports the generation of labelled nulls. For the example in Figure 6, the UTG includes *Area Info* and *UK Realestate* nodes, and an edge from *Area Info* to *UK Realestate*. The creation of the UTG is described in Section 5.2.

3. *Create Universal Target Relation (UTR).* A UTR is created for each connected join path in the target. For example, for the example in Figure 6, the UTR contains all the attributes from the *Area Info* and *UK Realestate* tables. Simple target mapping generation is called once for each UTR. The creation of the UTR is described in Section 5.2.

4. *Alter UTR to include labelled nulls.* The mappings generated for populating the UTR do not know about the complex target schema. However, attributes that have *null* values in the UTR may be used in the target to specify relationships using foreign keys. This step replaces

regular *null* values with labelled nulls, where these are required to capture relationships in the target. The addition of labelled nulls to the UTR is described in Section 5.3. For the example in Figure 6, there is no match for the *Area_ID* attribute (in either table) so it will not be populated with data from the sources. Thus, the values for this attribute need to be inferred so that the foreign key relationship between *Area_Info* and *UK Realestate* tables is maintained.

5. *Generate mappings from UTR to target tables.* Each UTR may provide data for several tables in the target. This step creates mappings from the UTR to target tables, as described in Section 5.3.

6. *Remove subsumed tuples.* As the tuples in a target relation may have been produced by several mappings, it is possible for the populated target to contain tuples that subsume each other. As a final step, these subsumed tuples are removed, as described in Section 5.3.

5.2. Universal Target Generation

This section describes the method for creating a *universal target generation*. This is formalized in Algorithm 6 and corresponds to *Create Universal Target* in Figure 5. The *universal target* has two representations: as a relation (UTR) and as a graph (UTG). Each representation has a different purpose in creating the mappings for the initial target relations: the UTR is used in the mapping generation search, where the UTR is the single target relation for which mappings are sought, whereas the UTG is used in the decomposition of the generated UTR mappings (Section 5.3). In Algorithm 6, COMPOSE takes as input a target schema (ts) and outputs a (set of) pair(s) of universal target relation(s) together with the corresponding graph(s), i.e., the output (line 15) has the form $[(utr_1, utg_1), \ldots, (utr_n, utg_n)]$, where n is the number of (disjoint) join graphs in the target schema. Dynamap generates a separate set of mappings for each pair (utr, utg) by running the search component (Section 4) each time. The algorithm first creates the UTG, then uses it to create the UTR.

Create Universal Target Graph. The purpose of the UTG is to determine how to *generate* and *decompose* the UTR mappings. The UTG not only preserves the initial format of the target relations, but provides an order for altering each UTR mapping, which is important for creating labelled nulls,

and subsequently for preserving inclusion dependencies between the initial relations. We use the definition of a directed-acyclic graph from [36]. In our setting, the *UTG* components are: *(i)* the nodes are target relations; *(ii)* the edges are represented by foreign keys; and *(iii)* the direction of an edge is given by the foreign key: the direction is from the referenced to the dependent relation. The direction of the edges determines the order in which labelled nulls are created for each initial target relation (as described in Section 5.3).

The *Create Universal Target Graph* component is illustrated in Figure 5 and detailed in lines 4-10 in Algorithm 6). In lines 4-5, all target relations are added to a graph, *target_graph*. In lines 6-7, the declared foreign keys are used to create the directed edges between the nodes. In line 8, FindConnGraphs finds all connected graphs within *target_graph* (one connected graph per join graph).

Create Universal Target Relation. The *UTR* is used to create a single table as target, a requirement for the mapping generation method described in Section 4. In this way, the data in the sources is first aligned in the format of the *UTR* and, then, we use the *UTR* to obtain the mappings for the initial target relations bundled up in the *UTR*. Populating the *UTR* first, and then *splitting* its data into the format of the target tables ensures that the source data is first correlated, and then the correlation between different tuples is maintained after for populating target tables and foreign keys.

The *Create Universal Target Graph* component is illustrated in Figure 5 and detailed in lines 11-13 in Algorithm 6. In order to create a *UTR*, the algorithm uses one *UTG* (which is a representation of a join graph). By following the edges in the *UTGs*, the algorithm creates each *UTR* as a single new relation comprising all target attributes in all the relations in the graph (lines 12-13). In Algorithm 6, the (*UTR*, *UTG*) pairs are created, added to the final output (line 14) and returned (line 15).

Example 6. *In Figure 6, the target schema comprises two relations, Area Info (AI) and UK Realestate (UKR). A join path connects the two relations as they share a foreign key, viz.,* $UKR.Area\_ID \rightarrow AI.Area\_ID$*. The corresponding UTR has the following schema:*

$UTR(Area\_ID, IncomeRank, City, Crimerank,$
$Prop\_ID, Postcode, Street, Price)$

where the foreign key attributes are represented by one attribute, viz., $Area\_ID$*. If there was no foreign key between the relations, then the algorithm would*

Algorithm 6 Generate UT

```
1: function COMPOSE(ts)
2:     output_uts ← []
3:     target_graph ← ()
4:     for each tr in ts.relations do
5:         target_graph.addNode(tr)
6:     for each fk in ts.foreign_keys do
7:         target_graph.addEdge(fk)
8:     conn_graphs ← FINDCONNGRAPHS(target_graph)
9:     for each cg in conn_graphs do
10:        utg ← cg
11:        utr ← ()
12:        for each tr in cg.nodes do
13:            utr.addAttributes(tr.attributes)
14:        output_uts.addPair(utr, utg)
15:    return output_uts
```

create a UTR for each target relation, and the dynamic programming search would be run twice.

5.3. Universal Target Decomposition

This section describes the method for *decomposing the UTRs*, i.e., the method for populating the original target from the UTR. The decomposition aims to satisfy the target constraints through the creation of labelled nulls and removal of subsumed tuples. This part of mapping generation for a complex target is illustrated by the *Universal Target Decomposition* component in Figure 5 and is formalized in Algorithm 7.

Algorithm 7 (DECOMPOSE) takes as input an array of objects that abstract over a set of *UTR* mappings (*utr_maps*[]) and its corresponding universal target graph (*utg*). For simplicity, we will refer to each *UTR* mapping object as being a *UTR* mapping, although it is just a representation of a mapping that is used for creating the output mappings, i.e., the mappings for each initial target relation. The number of output mappings for each *UTR* mapping is equal to the number of initial target tables that were used in the creation of the UTR. Each output mapping in *output_maps* corresponds to one node (i.e., one target table) in the *UTG*. Through TopoSort, in line 3, the algorithm sorts the set of nodes in *utg* in *topological sorted order* [37]. The purpose of the ordering is to generate labelled nulls for referenced relations

Algorithm 7 Decomposition of *UTR* mappings

```
1: function DECOMPOSE(utg, utr_maps[])
2:     output_maps ← []
3:     sorted_nodes ← TOPOSORT(utg.nodes)
4:     for each utr_map in utr_maps do
5:         for each tr in sorted_nodes do
6:             key ←FINDPKEY(tr)
7:             skolem_atts ← tr.attributes − {key}
8:             utr_map ←SKOLEMIZE(utr_map, skolem_atts, key)
9:         for each tr in sorted_nodes do
10:            target_map ←PROJECTION(utr_map, tr.attributes)
11:            target_map ←SUBSUMPTION(target_map)
12:            output_maps.add(target_map)
13:    return output_maps
```

before dependent relations. In lines 4-12, each *UTR* mapping is processed and the mappings for each initial target relation are obtained. In lines 5-8, the algorithm sequentially modifies the *UTR* mapping objects so that, when the executable mappings are created, the attributes in the *UTR* that correspond to keys in the initial target tables are populated with labelled nulls where source-extracted data is missing. In lines 9-12, based on the skolemized *utr_map*, the algorithm creates a new mapping object for each initial target relation by selecting corresponding attributes (equivalent to applying projection on the view of an executed *UTR* mapping – Projection, in line 10). Then, the new mapping object is modified so that, when the corresponding mapping is executed, it does not produce subsumed tuples (SUBSUMPTION, in line 11). The mappings for the initial target tables are added to the output set (line 12) which is returned (line 13). Next, we explain SKOLEMIZE for labelled nulls generation, and SUBSUMPTION for eliminating subsumed tuples.

Labelled nulls generation. We describe an algorithm that populates a multi-relation target schema where constraints such as candidate keys and foreign keys are tackled by populating their corresponding attributes in ways that take account of the constraints. In order to avoid redundant tuples, when labelled nulls are created, we consider for each table *all* the attributes in the tuple (without extending to any dependent/referenced relations). The labelled nulls only replace null values, and leave the data unmodified if it comes from sources. If an attribute is only partially populated, then the labelled nulls only replace the missing values for that attribute. Through

the SKOLEMIZE function (line 8), the algorithm sequentially adds skolem functions for each key constrained attribute in the *UTR*. The order in which the key attributes are processed is given by the topological sorting of the nodes in the *UTG*. We define SKOLEMIZE as:

$$skolemize(m, a[\,], a_{pk}) = \{t | t \in m, t[pk] \leftarrow coalesce(t[a_{pk}], f_{SK_{pk}}(t[a]))\}$$

The above definition is explained as: *skolemize* takes as input *(i)* a mapping (object) m, *(ii)* a (sub)set of its attributes $a \subseteq schema(m)$, and *(iii)* a single attribute a_{pk} which is a primary key in one of the initial target tables, $a_{pk} \notin a$. Then, for each tuple in m, replace the nulls on the primary key ($t[a_{pk}]$) with labelled nulls generated by a skolem function $f_{SK_{pk}}(t[a])$ that creates a unique value based on the values on attributes a for tuple t. Here, we chose the function f_{SK} to generate a hash value on the concatenation of the non-null values from $t[a]$. The behaviour of the function can be changed, though. The *coalesce* function is used to express that if the $t[a_{pk}]$ value is non-null, then it is left unmodified, otherwise it is replaced with a labelled null.

Example 7. *Consider a simplified case of the scenario in Figure 6 where only Manchester matches the target. The corresponding UTR is in Example 6. The mapping between Manchester Realestate (M) and the UTR is (for simplicity, the unmatched source attributes are omitted):*

$$\forall c, s, pc, pr : M(c, s, pc, pr) \rightarrow \exists AID, IR, CR, PID : UTR(AID, IR, c, CR, PID, pc, s, pr)$$

Given the target constraints, the first pass through SKOLEMIZE *(lines 5-8 in Algorithm 7) will add a skolem function that populates UTR.Aid from which both AI.Aid and UKR.Aid will draw values when projection is applied (line 10 - Algorithm 7):*

$$\forall c, s, pc, pr : M(c, s, pc, pr) \rightarrow \exists f_{Aid}, IR, CR, PID : UTR(f_{Aid}(c), IR, c, CR, PID, pc, s, pr)$$

After altering the UTR mapping for Aid attributes, Algorithm 7 proceeds to add the skolem function for $UKR.Pid$:

$$\forall c, s, pc, pr : M(c, s, pc, pr) \rightarrow \exists f_{Aid}, f_{Pid}, IR, CR : UTR(f_{Aid}(c), IR, c, CR, f_{Pid}(f_{Aid}(c)|pc|s|pr), pc, s, pr),$$

where $a|b$ means that a is concatenated with b.

After the UTR is altered to generate labelled nulls, the mappings for the target relations are the corresponding projections on the altered UTR as follows (for

Pid	Aid	Postcode	Street	Price
$f_{Pid}(f_{Aid}(Manchester)\|M15BD\|PrincessRoad\|950,000)$	$f_{Aid}(Manchester)$	M1 5BD	Princess Road	950,000
$f_{Pid}(f_{Aid}(Manchester)\|M15EB\|3CambridgeSt.\|325,000)$	$f_{Aid}(Manchester)$	M1 5EB	3 Cambridge St.	325,000
$f_{Pid}(f_{Aid}(Manchester)\|M31NN\|MirabelStreet\|325,000)$	$f_{Aid}(Manchester)$	M3 1NN	Mirabel Street	325,000
$f_{Pid}(f_{Aid}(Manchester)\|M31NP\|MirabelStreet\|165,000)$	$f_{Aid}(Manchester)$	M3 1NP	Mirabel Street	165,000

Table 4: UK Realestate tuples after skolemization

Aid	City
$f_{Aid}(Manchester)$	Manchester

Table 5: Area Info tuples after skolemization

simplicity, the existential variables and the unmatched source attributes are omitted).

The mapping for Area Info (AI) target table:

$UTR(f_{Aid}(c), IR, c, CR, f_{Pid}(f_{Aid}(c)|pc|s|pr), pc, s, pr) \rightarrow AI(f_{Aid}(c), c)$

The mapping for UK Realestate (UKR) target table:

$UTR(f_{Aid}(c), IR, c, CR, f_{Pid}(f_{Aid}(c)|pc|s|pr), pc, s, pr) \rightarrow$
$UKR(f_{Pid}(f_{Aid}(c)|pc|s|pr), f_{Aid}(c), pc, s, pr)$

The resulting populated target relations are in Tables 4 and 5. Note that there is only one tuple representing Manchester city in the Area Info relation, although in the initial Manchester table it appears several times (see Figure 2). This is due to the fact that there is no other evidence to help understand whether there is more than one Manchester city corresponding to the tuples in UK Realestate. Were there more information on the tuples in Area Info, the algorithm would generate different labelled nulls for the different cities regardless of the name being the same.

Subsumed output tuples. By eliminating subsumed tuples, the generated mappings may come to satisfy candidate key constraints, which would otherwise contain duplicate values.

After the labelled nulls are created for each key constrained attribute in the *UTR* mapping, Dynamap alters the already modified *UTR* mappings such that subsumed tuples are discarded when the mappings are materialized. This step is shown in Figure 5 as the third post-processing step, *Alter Mappings to Remove Subsumed Tuples*, and as SUBSUMPTION in line 11 in Algorithm 7. We consider a tuple to be subsumed if it satisfies the conditions outlined in [38]:

A tuple $t_1 \in T$, where T is a relation, subsumes another tuple $t_2 \in T$ if (i) t_1 and t_2 have the same schema, (ii) t_2 contains more null values than t_1, and (iii) t_2 coincides in all non-null attribute values with t_1.

Thus, we define the SUBSUMPTION method as:

$subsumption(m) = \{t | t \in m, X \in schema(m), \nexists t' \in m \text{ s.t. } |nulls(t')| \leq |nulls(t)| \wedge t[X] = t'[X] \wedge t[X] \neq null\}$

Example 8. *Consider the following three tuples:*

$t_1(1, 131\ 782, Manchester, Lancashire, \perp)$

$t_2(1, \perp, Manchester, Lancashire, \perp)$

$t_3(1, \perp, Manchester, GreaterManchester, \perp)$

Tuple t_1 subsumes tuple t_2 as they have the same schema (belong to the same relation), t_2 has the same non-null values as t_1, but t_2 has one more null than t_1 (on the second attribute), thus t_2 is subsumed by t_1. Similarly, t_3 has the same schema as t_1 and t_2 and some of the same values and nulls on the same positions, but the non-null value on fourth position is not the same as in t_1 and t_2. Thus, it cannot be concluded that it is subsumed by either of these.

How do results on data exchange, for example relating to value invention [39] or target constraints [40] carry forward into this setting? We note that in our *in the wild* setting, we cannot make many assumptions about source or target constraints. For example, the user may be of the view that the target should satisfy some equality generating dependencies. However, in practice, for example due to inconsistent value representations in different sources, these may not hold for the retrieved data. Furthermore, key constraints in the target may not be satisfied; different sources may provide alternative views of the world that are not easily reconciled without resorting to entity resolution techniques.

6. Evaluation

In this section we evaluate the performance of Dynamap on two types of synthetic scenario (Sections 6.1 and 6.6) and two real-world scenarios (Sections 6.2 and 6.3). In Section 6.4 we evaluate the accuracy of the propagation rules (Section 4.5), and in Section 6.5 we show the efficiency of the pruning strategies (Section 4.7).

Setup. Dynamap and ++Spicy were run over the same sources, and the same target. For storage, we used PostgreSQL 9.6. Given that ++Spicy uses explicit schema constraints, based on the profile data, foreign keys between the sources are inferred where possible, i.e., a candidate key shares a (full)

inclusion dependency with an attribute from another relation. Tuples in the synthetic scenarios were generated using *Datafiller* [41]. The experiments were run over an Intel Core i5 with 2×2.7 GHz, and 8 GB of RAM. We report the results over the average of 10 runs when runtime is measured.

Evaluation metrics: *iBench scenarios.* We used a metric that was previously used in [10]: mappings that output less constants and less nulls are considered to be desirable as it means that the data is correlated better.

Realcase scenarios. The result of the output mapping is compared with that of a ground-truth mapping, reporting the precision, recall and f-measure at the attribute value level, based on the following definitions:

A *true positive* is a correct non-null output value.

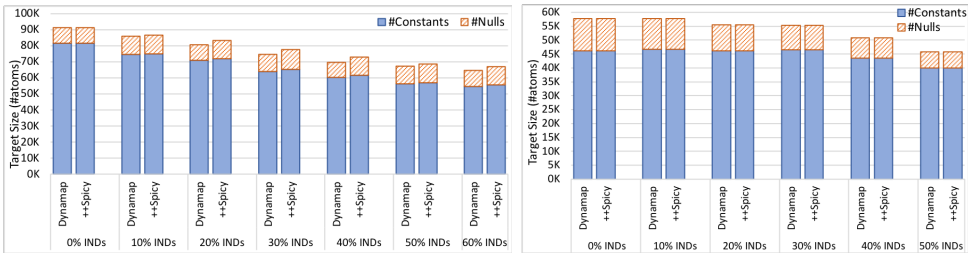
A *true negative* is a correctly output null, i.e., it was expected to be null in the ground truth.

A *false negative* is a missing output value (a null).

A *false positive* is a non-null incorrect output value.

Mapping selection. For the experiments, for Dynamap, mappings were selected by choosing from the fittest k mappings (Section 4.1) as few mappings as possible such that all initial relations are involved. The mappings are selected by applying the *set-cover* method [36] to the subsets of initial relations merged in each mapping. For ++Spicy, we used the generated SQL script that is considered to contain the best mappings that populate the chosen target. In both cases, other mapping selection techniques could be applied, e.g., considering properties of the extents of the mappings [42].

Comparison with ++Spicy. In this section, comparisons are drawn with ++Spicy [17] in terms of result quality and runtime performance. Dynamap and ++Spicy were run over the same mapping task with the same input sources, their output mappings were executed and the output tuples are compared to the output of the ground truth. ++Spicy has been chosen as it is publicly available, and represents the state-of-the-art in *mapping generation for databases*; we think that ++Spicy does what it was designed to do rather well. We note that ++Spicy was not designed to support *mapping generation in the wild*, and thus that in places the comparison with Dynamap may not seem entirely fair. However, this reflects the fact that *mapping generation in the wild* presents new challenges, and we know of no other more suitable system with which to conduct comparative evaluations.



(a) Simple target (b) Complex target

Figure 7: iBench experiments

6.1. iBench*: Integration Metadata Generator*

iBench [10] is a tool that generates data integration/exchange scenarios, where the sources have explicit keys and foreign keys. Although not *in the wild*, these scenarios are relevant for our purpose as they consist of a variety of base case primitives that mapping generation algorithms should be able to tackle. iBench denotes a *primitive* as a scenario that involves one source schema and one target schema where a specific type of merge is needed to transfer the data from the source to the target. The merge involves a variation of copying and/or joining source relations to populate the target.

Here, the experiments follow the methodology presented in [10], where iBench is used to compare several mapping generation algorithms. The measures proposed in [10] imply that the mappings that produce smaller target instances produce less incompleteness, so they measure the *size* of the target which consists of the number of atoms [3] that could be a *constant* or a *null*.

Simple target scenarios. The scenarios are built as:

Target. In order to keep the target schema fixed, we used a user defined primitive where the target schema is given, and set the corresponding iBench parameters to reuse 100% of the target schema. The target schema is a nine-attribute target relation.

Input sources. Each generated scenario has 20 source relations with 4-12 attributes, each with 400-600 tuples that are generated with Datafiller [41]. We chose to create scenarios with only 20 input source relations as the purpose of this experiment is to investigate specific mapping generation patterns, not how the algorithm scales (which is investigated in Section 6.6). We used the following iBench primitives: *add attribute*, *add-delete*, *delete*, *copy* and *merge-add*.

In generating the different scenarios, we varied the number of primitives so that the 20 input source relations created have 0% to 60% of their relations linked by inclusion dependencies. In other words, the generated scenarios depict cases where the source relations are mostly unionable w.r.t. the target relation (but having different matches to the target) to cases where the number of relations that are joinable increases, i.e., by increasing the number of source relations that are linked by inclusion dependencies. The reuse of the source relations is set to 0%, i.e., each primitive has its own associated source relations, as sharing the same source relation for several primitives changes the target schema by adding target relations.
Matches. All sources will match the target; the matches are generated by iBench according to the primitives.
Profile data. The profile data is generated according to the inclusion dependencies in each scenario and the defined primary keys in each relation.
Results. The results are in Figure 7(a), where the output atoms of the mappings generated by Dynamap are compared with the output of ++Spicy mappings. It can be observed that for the scenario with 0% INDs, their output is identical in terms of number of constants and nulls, but once the scenarios start having relations with inclusion dependencies, their output is slightly different. This difference comes from the fact that for the *merge-add* primitives Dynamap outputs only the joined tuples, while ++Spicy outputs all tuples, regardless of whether the tuples could be combined or not. Dynamap chooses the mappings that output only the merged tuples as it prefers mappings that have fewer incomplete tuples, thus, the mappings that produce tuples that bring more nulls than constants are not produced. However, either of these outputs could be considered to be reasonable. In terms of values, all Dynamap values were identical with the corresponding tuple values of ++Spicy.

Complex target scenarios. We reproduced as closely as possible one of the experiments in [10], where the number of foreign keys in the target schema is varied using different primitives for vertical partitioning (VH, VI, and VNM) and ADD primitives. In [10], they describe these primitives as part of *Ontology* scenarios as each *i)* VH primitive creates two target relations in a HAS-A relationship, *ii)* VI creates two target relations in an IS-A relationship, *iii)* VNM creates three target relations in a many-to-many relationship, and *iv)* ADD creates one target relation that copies a source relation, but needs more attributes than that source.

Dataset	Data for the target		#DS	#A	#T
	simple	complex			
Manchester real-estate	street, price, city, postcode	s_name, price, c_name, postcode	5	5-9	20 - 171
London real-estate	street, price, city, postcode	s_name, price, c_name, postcode	2	6-13	20-35
Oxford real-estate	price, street, postcode	price, s_name, postcode	4	10-14	28-152
Manchester deprivation	postcode, crimerank	postcode, crimerank	1	28	391
London deprivation	postcode, crimerank	postcode, crimerank	1	28	54
Manchester & Oxford addresses	street, city, postcode	s_name, c_name, postcode	1	4	235

Table 6: Real-estate datasets

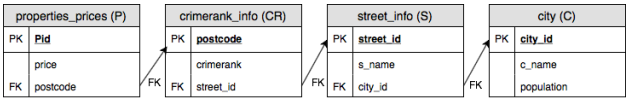


Figure 8: Real-estate target schema with constraints

Target schema. We set the number of target tables to 40, their arities to vary between 3 and 7 attributes, and the corresponding iBench parameters to reuse 0% of the target schema. For each iBench scenario, the number of target constraints (foreign keys) varies from 0% to 50%. This means that the scenarios shift from ADD-dominant scenarios to VP-dominant scenarios.
Input sources. Varying the number of primitives the number of sources ranges from 40 source relations (with 0% target INDs) to 28 relations with (50% target INDs), with arities from 3 to 7 attributes, and each relation has 400-600 tuples that are generated with Datafiller [41].
Matches and profile data. Created as for *simple target*.
Results. The results are shown in Figure 7(b). The performance of both algorithms is the same because the scenarios do not present any challenges, i.e., there are no alternative ways of merging the sources: the sources are disjoint, they are only vertically partitioned into two (for VI&VH primitives) or three (for VNM primitive) foreign key tables, tasks which both algorithms are able to perform successfully. In terms of attribute values, all output values of Dynamap were identical with the corresponding ones of ++Spicy. The only inessential difference is in the format of labelled nulls.

6.2. Real-estate Domain

In this section, we investigate a real world scenario in which web-extracted datasets from the real-estate domain are combined with data from the UK open government data portal. Below, we evaluate Dynamap for *simple* and *complex targets* for data quality. These experimental results complement the

ones described in our previous work from [11], where we show that Dynamap can successfully tackle *simple target* scenarios.

Scenarios. For both scenarios, we used the same input sources and profiling data, but the target schema and the matches are changed accordingly. The input sources and profiling data are described below.

Input sources. The input sources contain data from three categories: real-estate data, deprivation and addresses. Details about the input datasets are found in Table 6 where the first column states what the data represents, the second and the third outline which information from the dataset is necessary to populate the simple and the complex targets, respectively, the third column represents the number of input data sources that contain that type of data, the fourth is the arity range, and the fifth represents the cardinality range.

Profile data. To obtain the profile data on the input sources, HyUcc [9] was run to detect the candidate keys, and SINDY [34] was run to obtain the (partial) inclusion dependencies. The input profile data contains: 68 candidate keys, 1735 partial inclusion dependencies, and 509 full inclusion dependencies. Given that ++Spicy uses explicit schema constraints, based on the profile data, foreign keys are inferred where possible, i.e., if a candidate key shares an inclusion dependency with another relation's attribute then a foreign key is inferred.

Simple target. The target is a single-relation target schema without constraints:

Target (postcode, city, street, price, crimerank)

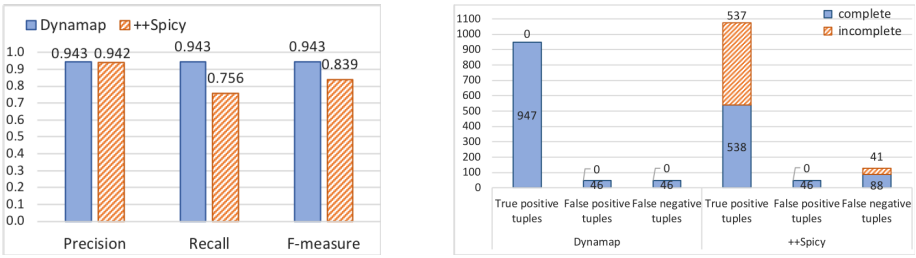
Complex target. The target is a four-relation schema with constraints, i.e. primary and foreign keys. This is depicted in Figure 8 where the primary keys are underlined attributes and foreign keys are arrows from the dependent to the referenced table.

Ground truth. The ground-truth mappings were created by a human expert.

Dynamap and ++Spicy were run over the same mapping task with the same input sources, their output mappings were executed and the output tuples are compared to the output of the ground truth.

Results for a simple target. The results of the two mappings against the ground truth mapping can be seen in Figures 9(a) and 9(b).

Attribute level. The results at attribute level are shown in Figure 9(a). Both algorithms perform similarly in terms of precision, i.e., close to all the attribute values that Dynamap and ++Spicy output are the same as in the



(a) Attribute level

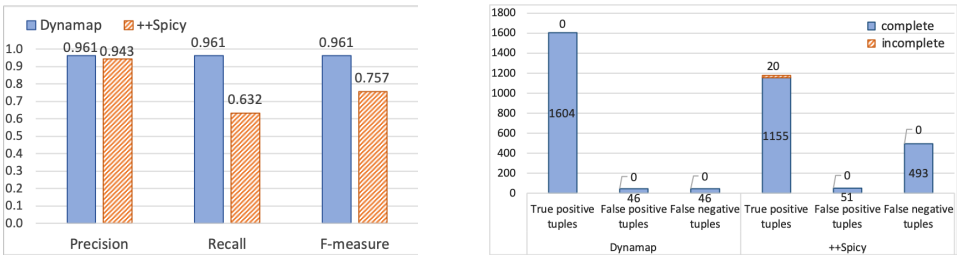
(b) Tuple level

Figure 9: Performance of Dynamap and ++Spicy on a *real-estate* scenario for a *simple target*

ground truth. The difference between their recall is caused by the fact that Dynamap manages to correlate more data, which leads to fewer but more complete tuples, while ++Spicy does not merge relations that match the same target attributes unless those attributes can be used in the join condition.

Tuple level. The results at tuple level are depicted by Figure 9(b). Both algorithms perform similarly in terms of total number of true positive tuples, i.e., Dynamap produces 947, and ++Spicy outputs 1075. The difference between their results comes from the fact that Dynamap manages to correlate more data, which leads to fewer but more complete tuples. Also, all the TP tuples from Dynamap are complete, i.e., all attribute values are the same as in the ground truth tuples, while ++Spicy identifies only 538 complete tuples and 537 incomplete TP tuples. Incomplete TP tuples are expected in the ground truth, but they are only partially correct as some attribute values are correct, but others are either missing or incorrect. The false positive tuples that both produce are due to the fact that there are 46 tuples in each of their outputs that have *null* on the key attribute and, thus, could not be compared to any of the ground-truth tuples. This behavior reflects the fact that the input sources are heterogeneous and autonomous so not everything can be readily combined as in a well-behaved schema. The false negative tuples that both Dynamap and ++Spicy produce stem from missing key values in the sources. The additional false negatives produced by ++Spicy are created because some of these tuples were candidates for joining, but they were not merged, thus, producing more false negatives than Dynamap (as they also have missing keys so cannot be correlated with ground-truth tuples).

Results for a complex target. The results of the two mappings against



(a) Attribute level

(b) Tuple level

Figure 10: Performance of Dynamap and ++Spicy on a *real-estate* scenario for a *complex target*

the ground-truth mapping can be seen in Figures 10(a) and 10(b).

Attribute level. Figure 10(a) shows the results at attribute level. In this scenario, the results for the two algorithms are not significantly different as they both manage to align most of the source tuples and populate the target as expected. Some tuples produced by both algorithms were not expected because there are agency properties that do not have a corresponding *postcode* in the sources. Because of this, there is no value on *postcode* to compare it to the ground truth tuples. However, they are the same (and correctly transformed from the sources) for both algorithms. The low recall on the performance of ++Spicy is due to false negatives. We explain below (at tuple level) why these occur.

Tuple level. The results at tuple level are depicted in Figure 10(b). At tuple level, it can be observed an increase in false negative tuples for ++Spicy. The false negative tuples are actually tuples that ++Spicy manages to discard as they are subsumed by other tuples. It manages to do so by using *egds* on the *postcode* attribute and by materializing various combinations of tables, thus, it manages to eliminate from the output the tuples that had overlapping *postcode* values. The SQL script that ++Spicy creates materializes 119 intermediate relations in order to discard subsumed tuples. It would be unfeasible for a human to hand craft such complicated scripts for the creation of the ground truth, thus, the created ground truth is designed to correlate data as best as it can without using materialized intermediate data. This scenario is advantageous for ++Spicy which is designed to tackle especially well scenarios where *all* available sources match the *same* target key attribute. However, in a real-world data lake (with thousands of sources) this is unlikely

Dataset	Data for the target		#DS	#A	#T
	simple	complex			
All schools	dfe code, school name, headteacher	dfe_code, s_name, ht_name	1	16	99
Free meals eligibility	dfe code, school name, elig. students	dfe_code, s_name, pupils_FSM_eligible	1	4	85
Additional languages	dfe code, school name, pupils_EAL	dfe_code, s_name, pupils_EAL	6	3-6	24 - 88
Road & Safety training	school name, school type	s_name, type_name	1	3	46
Bikeability courses	school name, bikeability courses	s_name, bikeability_courses	1	6	87

Table 7: Schools domain datasets

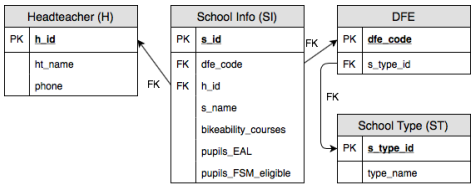


Figure 11: Schools domain target schema with constraints

to happen. We show a different behaviour in a more challenging scenario in Section 6.3.

Comparing the populated target tables separately, one major difference between the two approaches is on the labelled nulls they generate. For instance, the size of the *city* relation populated by Dynamap has 117 tuples compared to 850 tuples in the case of ++Spicy. From the ++Spicy tuples, 609 have one non-null value and then an entirely empty row, e.g., 97 have labelled nulls for primary keys, and 512 tuples have *Manchester* value on *c_name* but nulls on the other attributes. The creation of such redundant labelled nulls happens because, even though the *city* information is missing, there is information for the other attributes in a foreign key table, e.g., for *street_info* table. So it links two such tuples although tuple in *city* even though there is no information in it. The creation of duplicate values happens because, the ++Spicy method for creating labelled nulls generates a separate entry in *City* relation for each tuple that has different information in it, e.g., *price*, or *street name*, in *Manchester*.

6.3. Schools Domain

Similarly to Section 6.2, we evaluate Dynamap on two scenarios with real-world datasets with both *simple* and *complex targets*.

Scenarios. For both scenarios, we used the same input sources and profiling data, but the target and the matches are changed accordingly.

Input sources. The data sources contain information about schools, more

specifically, about the facilities in those schools. Table 7 contains their details.

Profile data. The same method as in Section 6.2 was used to obtain the profile data. The input contains 48 candidate keys, 681 partial inclusion dependencies, and 47 full inclusion dependencies. Similarly to Section 6.2, we created explicit foreign keys between the sources to give as input to ++Spicy.

Simple target. The target is a single-relation schema:

Target(dfe code, school name, school type, headteacher, #bikeability courses, #pupils_EAL, #eligible students).

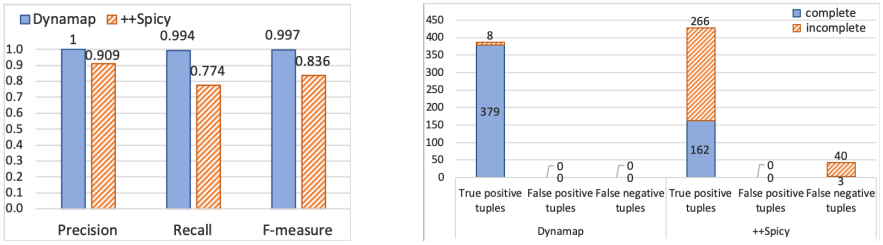
Complex target. Figure 11 depicts the target comprising four relations with constraints.

Ground truth. The ground-truth mapping was handcrafted by an expert.

Results for a simple target. The results of the two mappings against the ground-truth mapping can be seen in Figures 12(a) and 12(b).

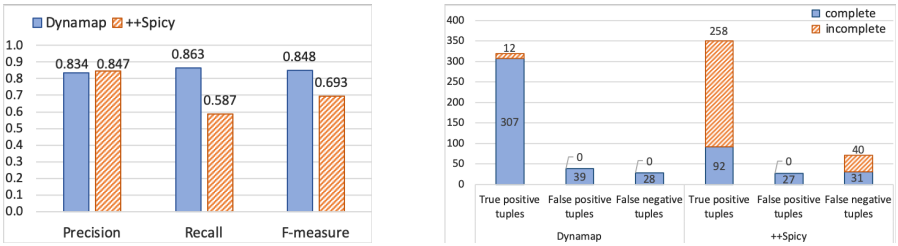
Attribute level. The results can be seen in Figure 12(a). The precision of both mappings is high, i.e., the majority of their identified attribute values match the ground truth. The discrepancy in recall happens because, although ++Spicy makes use of explicit join paths and removes redundancy by using equality generating dependencies, these steps are not enough as this is not a well behaved scenario. In this scenario, ++Spicy merges the relations with matches to attributes that are keys in the target, and this reduces redundancy in the output, but relations that do not have such matches are not merged with the other relations. For example, *All schools, Free meals eligibility* and *Schools with additional languages* all match the key target attributes, and thus are joined, but *Road and Safety Training* and *Bikeability* only match non-key target attributes, so ++Spicy does not consider them for merging with the other relations. Dynamap follows join paths defined by partial inclusion dependencies, and resorts to outer joins when foreign keys cannot be inferred, and thus more fully combines data from the source tables.

Tuple level. In Figure 12(b), considering the completeness of the output tuples, it can be observed that Dynamap outperforms ++Spicy. This is because Dynamap combines the source tables almost as expected in the ground truth, with only 8 partially correct tuples, whereas ++Spicy outputs 266 partially correct tuples, and only 162 tuples with all information correct. This discrepancy in the output quality happens for the same reason stated above (for attribute level). Thus, ++Spicy is not able to correlate all the information so it outputs mostly partially correct tuples (266), 40 tuples are considered



(a) Attribute level (b) Tuple level

Figure 12: Performance of Dynamap and ++Spicy on a *schools* scenario for a *simple target*



(a) Attribute level (b) Tuple level

Figure 13: Performance of Dynamap and ++Spicy on a *schools* scenario for a *complex target*

incomplete false negative tuples as they have a few correct values, but mostly *null* values where in the ground truth there were expected non-nulls.

Results for a complex target. The results of the two mappings against the ground-truth mapping can be seen in Figures 13(a) and 13(b).

Attribute level. Figure 13(a) shows the results at attribute level. It can be observed that ++Spicy seems to outperform Dynamap by 0.013 in precision. This is because ++Spicy produces more tuples that have the same school name than Dynamap as ++Spicy fails to do the expected joins with the sources that do not match target keys, as ++Spicy uses target *egds* to remove redundancy [17]. However, although the data produced by ++Spicy is missing some correct non-null values on those uncorrelated tuples, some null values are considered as correct more than once (the ones that are expected to be nulls in the ground truth), thus, increasing the number of true negatives. In terms of recall, Dynamap has a better performance than ++Spicy by 0.276. Dynamap does not achieve the maximum because it produces 249 false negatives. These FNs are produced as, although Dynamap joins the

	Percentage of estimation θ with error in the range									
	[0.0, 0.1)	[0.1, 0.2)	[0.2, 0.3)	[0.3, 0.4)	[0.4, 0.5)	[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]
Realestate	93.3	3.1	1.1	0.8	0.6	0.5	0.4	0.3	0.1	0
Schools	56.6	19.4	10.3	8.7	2.4	1.8	0.5	0.3	0.1	0

Table 8: Error ranges

correct relations, not all joins are performed on the expected condition so some tuples are not correlated as expected in the ground truth. ++Spicy has a lower recall because it misses join opportunities as it relies on *egds*, a technique that is not suitable in this scenario as not all sources match target key attributes (as in Section 6.2). Thus, the number of false negative cell values for ++Spicy (907) is 3.6 times higher than for Dynamap (249).

Tuple level. Figure 13(b) shows the results at tuple level which depict the ability of both algorithms to correlate data in the target. The discrepancy between the results of Dynamap and ++Spicy is caused by the same reason as explained above. Also, in Figure 13(b), it can be observed that Dynamap produces 39 false positives. This is because of the nature of the source data in which the same entity has different representations, i.e., the names of the schools (although correct) differ, in the output of Dynamap, but chosen from another source to represent the same entity. Hence, there are 39 Dynamap tuples that do not have a school name corresponding to the ground truth because of variations in the name. The tuples without a ground-truth counterpart are considered false positives in their entirety. One would say that this can be a common scenario for mapping generation over autonomous sources as many sources may contain data about the same entities, in different formats.

6.4. Profiling Data Propagation Accuracy

In this section we investigate the accuracy of the propagated inclusion dependencies (Section 4.5) for the scenarios in Sections 6.2 and 6.3. To measure the accuracy, we materialized the intermediate mappings generated in all iterations, and we used *SINDY* [34] to obtain the ground-truth inclusion dependencies with accurate overlap. The results of the experiment can be seen in Table 8, which identifies the percent of the estimated overlaps in different error ranges.

Real-estate scenario. For this scenario, 112,147 inclusion dependencies were compared to the ground truth that was generated using 785 materialized intermediate mappings. 83,617 overlaps were equal to the ground truth

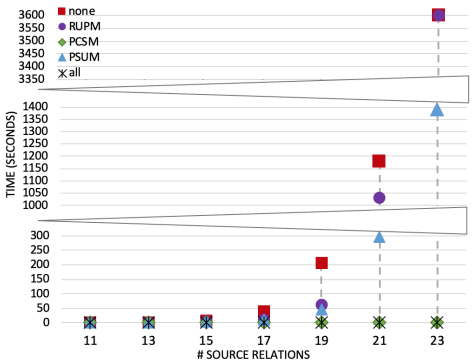


Figure 14: Pruning strategies impact

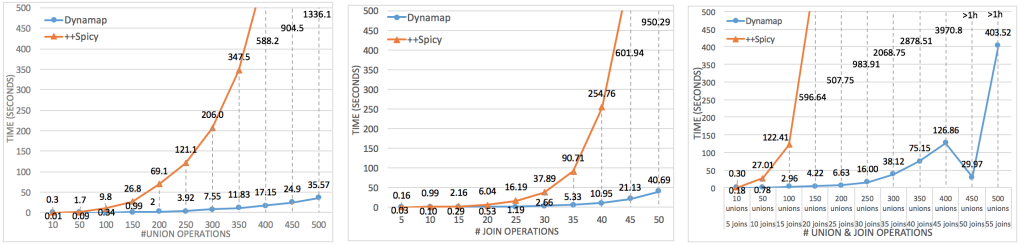
overlaps, and 28,530 were different. It can be observed from Table 8 that most differences were in the range $[0, 0.1)$, (i.e., 20,964 of the 28,530 estimates that were different from the ground truth), meaning that the estimates were close to the true value. The *mean average error* over all overlaps is 0.025.

Schools scenario. For this scenario, 36,917 inclusion dependencies were compared to the ground truth which was obtained from 116 materialized intermediate mappings. 9319 overlaps were equal to the ground-truth overlap, and 27,598 were different. In Table 8, it can be observed that most estimates had no error or an error below 0.1. The *mean average error* is 0.12. The mean average error is larger than in the real estate scenario because only ≈6% of the inclusion dependencies in the *schools* scenario are full inclusion dependencies, c.f. ≈23% for the *real-estate* scenario.

6.5. Effectiveness of Pruning Strategies

We show the effectiveness of the pruning strategies by measuring the runtime with each pruning strategy active so as to quantify how each strategy impacts on the overall run-time. The results are depicted in Figure 14. In each case, the runtimes were capped to one hour.

We report runtimes for five different cases: with all pruning strategies active (*all*), with none active (*none*), and, with each pruning strategy activated separately, i.e., *removing unnecessary parent mappings* (*RUPM*), *preventing creation of superfluous mappings* (*PCSM*), and *pruning subsumed union mappings* (*PSUM*). In each case, Dynamap generated the expected mapping.



(a) Instance complementarity scenarios (b) Schema complementarity scenarios (c) Mixed scenarios

Figure 15: Runtime of mapping generation for synthetic data scenarios

Scenarios. The synthetic scenarios were created by increasing the number of expected join and union operations in the ground truth, while keeping them equal, e.g., the smallest scenario contains 5 unions and 5 joins, and the largest contains 11 of each type of operation.

Results. In Figure 14, it can be observed that for the smallest two scenarios, the pruning strategies do not significantly improve the running time as the search space is not especially large. However, once the number of sources increases beyond 19, the runtime starts to be affected by the combinatorial properties of dynamic programming, such that for the largest scenario (23 sources) the runtime without pruning goes beyond one hour (not depicted in Figure 14), whereas with all pruning strategies active it runs in less than a second. The *PCSM* strategy has the highest impact of all. This significant improvement is due to the fact that it can prevent the creation of mappings, thus, the search space is contained by not creating mappings that will be discarded in subsequent iterations. *PSUM* and *RUPM* are effective at removing already created, but unnecessary mappings. Nonetheless, the fact that mappings are created and added to the search space considerably affects subsequent iterations, thus increasing the runtime.

6.6. Scalability Experiments

In this section, we investigate the impact of specific properties of the integration scenarios on the runtime performance of Dynamap and ++Spicy. The objective has been to provide scenarios that provide substantial search spaces, over numbers of tables that might be identified by searches over data

lakes [28, 43, 27]. To this end, we developed SYNTHEGRATE[2], a generator of integration scenarios that provides control over schema properties, such as arity, cardinality, number of candidate keys, number of source/target relations, and number of source schemas. It also allows control over the number of expected join and union operations, reuse of join attributes in other merge opportunities, and ratio of explicit foreign keys. Matches and profile data are created automatically by SYNTHEGRATE, reflecting the database schemas and the extents produced using Datafiller [41]. The ground truth mapping is also created automatically by SYNTHEGRATE.

SYNTHEGRATE complements the functionality of iBench through its ability to create complex integration scenarios while keeping the target schema fixed. In IBENCH, if the target schema is fixed then the number of scenarios that can be created is rather limited. Complex IBENCH scenarios are mostly generated by adapting the target schema to the new input parameters.

Figures 15(a), 15(b), and 15(c) show the processing time of Dynamap and ++Spicy under different types of integration scenarios. The measured runtime for Dynamap reflects the processing time to output the mapping in the final iteration (if found) and the runtime for ++Spicy includes the computation of core mappings. For both algorithms, the runtime measurements include only the mapping generation and the generation of SQL scripts (needed to evaluate the output). However, as explained in [16], the generation of the SQL script by ++Spicy can represent a significant amount of the total running time. For all experiments, we fixed a timeout of one hour. If the experiment was not completed by that time, it was stopped.

Instance complementarity scenarios. The number of *union* operations in the correct mapping is varied. These scenarios are a synthetic representation of cases where the relations that need to be merged contain the same type of information as is needed in the target.

Results. The mapping generation times for different numbers of union operations in the mapping scenario are in Figure 15(a). In terms of result quality, the result tuples are exactly as in the ground truth for both algorithms. In Figure 15(a), it can be seen that a mapping containing 500 unions has been generated by Dynamap in less than a minute, while ++Spicy generates it in $\approx$ 22 minutes. The time increase for both algorithms comes from the fact that, in such scenarios, all permutations of the input relations are reasonable

[2]https://github.com/MLacra/Synthegrate.git

candidate mappings. For Dynamap, this type of scenario provides a significant test for the pruning techniques that *prevent creation of superfluous mappings*, *prune subsumed union mappings* and *remove unnecessary parent mappings*, without which the search space for Dynamap would have grown following formula in Section 4.7.1.

Schema complementarity scenarios. In this type of scenario, we vary the number of *join* operations in the correct mapping. These scenarios are a synthetic representation of real-world cases where the relations that need to be merged contain different attributes of the data that are needed in the target, e.g., by bringing together information about a school from many sources.

Results. The results can be seen in Figure 15(b). In terms of result quality, the result tuples that Dynamap produces are exactly as in the ground truth for all scenarios. On the scenarios with fewer join operations, we were able to evaluate the output of ++Spicy and observe that it produces all the merged tuples which appear in the ground truth, but also the tuples that were not joined with other tuples and these are considered *false positives* as they were not expected in the output. For the large scenarios we were not able to execute the generated ++Spicy mapping on the input database.

In Figure 15(b), it can be seen that Dynamap generates a mapping containing 50 joins in less than a minute, while ++Spicy runs in approximately 15 minutes. However, it seems unlikely that mappings with upwards of 50 joins will be common in practice. In this type of scenario both algorithms have a similar approach for discovering the mapping, i.e., following foreign key join paths between the sources. The time difference comes from the fact that, although in this type of scenario the opportunities for combining relations are fewer than in a union dominant case, Dynamap identifies opportunities for pruning that depend significantly on *preventing creation of superfluous mappings* and *removing unnecessary parent mappings*, whereas ++Spicy tries to remove redundancy by combining the same sources in multiple variations.

Instance & schema complementarity scenarios. In this type of scenarios, we vary the number of *union* and *join* operations expected in the correct mapping. Through the variation of operators, unintentional merge opportunities were created as union relations can partially overlap with relations that are expected to merge with other relations, i.e., through the partial (or even full) overlap they can become candidates for joining although it was not by design. These scenarios represent real-world cases where some but not all

relevant source relations contain the same type of information.

Results. The results can be seen in Figure 15(c). In terms of result quality, the result tuples that Dynamap mapping produces are exactly as in the ground truth in 10 out of 11 cases. ++Spicy does not produce the expected tuples in any of the chosen scenarios (that ran under one hour): it identifies the union opportunities, but not all the correct join opportunities, leading to many output tuples padded with nulls. ++Spicy does not behave as expected because the majority of the join-condition attributes do not match the target key attributes, thus, ++Spicy is unable to use *egds* to merge the sources, while Dynamap does not rely on matched target keys to identify merge opportunities. The case where Dynamap generated only parts of the expected mapping was the case with 50 join and 450 union operations. This partial detection is due to the complexity of the scenario. In some cases, the approximated profile data is close to the actual values, but not equivalent, i.e., the overlaps of the inclusion dependencies can become partial instead of full, thus, an expected *join* is detected as *outer join*.

In Figure 15(c), it can be seen that a mapping containing 550 join and union operations was generated by Dynamap in less than 7 minutes, while ++Spicy runs in over an hour. For the scenario with 50 join and 450 union operations, the running time for Dynamap is significantly reduced. This is because, as explained in Section 4.5, the propagation of the profile data is influenced by the chosen operators. In this case, some mappings became unavailable to merge with other mappings as less profile information was transferred to them from the parent mappings because of the use of a *full outer join* instead of a *join*. This scenario provides a significant test for the pruning techniques that *prevent creation of superfluous mappings*, *prune subsumed union mappings* and *remove unnecessary parent mappings*, without which the search space would have grown much more rapidly than is reflected in Figure 15(c).

7. Conclusions

Schema mapping generation is important for reducing the currently prohibitive cost of data preparation. The proliferation of data sets, for example in data lakes, motivates the development of schema mapping generation algorithms for large and heterogeneous repositories. In such settings, schema mapping generation involves a search over a large space of candidate mappings.

Schema mapping generation has been cast as a search problem before. For example, in Clio [1], mapping generation involved searching the space of join paths in source and target tables, and, for example, in both Tupelo [33] and S4 [44] search algorithms are developed for exploring a space of candidate operator applications, informed by example instances. However, such proposals have been developed primarily for mapping from a single (potentially complex), but well-defined multi-table schema. In our setting, where there may be a plethora of multi-table schemas, originating from various publishers, we must depend on less reliable profiling information to describe source tables and the relationships between them, with a view to identifying promising ways of populating the target from the sources.

Here we revisit the contributions from the introduction, to make explicit the contributions made to understanding of mapping generation in the wild:

1. *A dynamic programming algorithm that explores the space of candidate mappings.* A dynamic programming algorithm has been chosen as it systematically explores the space of candidates, creating n-table mappings from combinations of mappings with fewer tables. The exploratory approach reflects the fact that we cannot build on declared relationships *in the wild*, as has been the focus of schema mapping generation in databases, building on Clio [1].

2. *Rules for deriving profile data for mappings from their operands.* Mapping generation proposals can be characterised in terms of the evidence they use, and this has been diverse, including declared schemas [1], example instances [14] and feedback [45]. In this paper, we use (readily available) inferred profiling data on candidate keys and (partial) inclusion dependencies to provide information on when tables can be joined. However, although such information can be derived on base tables [9], deriving profiling data on all candidate mappings would be prohibitively expensive. To solve this problem, we have developed systematic techniques for propagating profiling data on keys and inclusion dependencies through mappings. Such techniques can be used independently of the rest of Dynamap.

3. *Techniques for pruning the space of candidate mappings.* Although the dynamic programming based search is systematic in its exploration of the space of mappings, there can be a combinatorial explosion in the number of candidate mappings. Mapping generation in the wild

needs to scale to large numbers of sources and complex mappings. We have identified situations in which portions of the search space can be pruned, for example in the context of subsumed mappings, while retaining the systematic exploration of the space provided by dynamic programming,

4. *A method that populates a multi-relation target schema.* Contributions (1) to (3) provide systematic, scalable mapping generation informed by profiling data for single-table targets. To retain the benefits of the single-table approach for targets with multiple tables, including foreign keys, we have proposed an approach that: *i)* creates a universal target relation from a complex target; *ii)* applies single-table Dynamap to that target; and *iii)* maps the data from the universal target relation to the tables of the target. This approach retains the performance benefits of Dynamap for single-table targets, and takes into account target constraints, allowing for the reality that data *in the wild* may not satisfy declared constraints in the target.

5. *An empirical evaluation of the approach:* We have carried out an extensive evaluation that uses a variety of real-world and synthetic data sets to explore the following features of the proposal: *i)* the quality of the results produced by generated mappings; *ii)* scalability of mapping generation; *iii)* accuracy of inferred profiling data; and *iv)* the impact of different pruning strategies. These experiments provide extensive evidence that, building on readily-available profiling data, Dynamap provides dependable results on mapping generation in the wild.

Acknowledgements

We are pleased to acknowledge the support of the Engineering and Physical Sciences Research Council, United Kingdom, through the VADA Programme Grant [EP/M025268/1].

References

[1] R. Fagin, L. M. Haas, M. Hernandez, R. J. Miller, L. Popa, Y. Velegrakis, Clio: Schema mapping creation and data exchange, in: Conceptual Modeling: Foundations and Applications, Springer, 2009, pp.

198–236. doi:10.1007/978-3-642-02463-4_12.
URL http://dx.doi.org/10.1007/978-3-642-02463-4_12

[2] M. Arenas, P. Barceló, L. Libkin, F. Murlak, Foundations of Data Exchange, Cambridge Univ. Press, 2014.

[3] B. Alexe, M. Hernández, L. Popa, W.-C. Tan, Mapmerge: Correlating independent schema mappings, The VLDB Journal 21 (2) (2012) 191–211. doi:10.1007/s00778-012-0264-z.
URL http://dx.doi.org/10.1007/s00778-012-0264-z

[4] L. Chiticariu, W. C. Tan, Debugging schema mappings with routes, in: VLDB, 2006, pp. 79–90.

[5] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, G. Summa, Schema mapping verification: the spicy way, in: EDBT, 2008, pp. 85–96.

[6] P. A. Bernstein, L. M. Haas, Information integration in the enterprise, CACM 51 (9) (2008) 72–79. doi:10.1145/1378727.1378745.
URL http://doi.acm.org/10.1145/1378727.1378745

[7] Z. Abedjan, L. Golab, F. Naumann, Profiling relational data: a survey, VLDB J. 24 (4) (2015) 557–581.

[8] E. Rahm, P. A. Bernstein, A survey of approaches to automatic schema matching, VLDB J. 10 (4) (2001) 334–350. doi:10.1007/s007780100057.
URL https://doi.org/10.1007/s007780100057

[9] T. Papenbrock, Y. Bergmann, M. Finke, J. Zwiener, F. Naumann, Data profiling with metanome, Proc. VLDB Endow. 8 (12) (2015) 1860–1863. doi:10.14778/2824032.2824086.
URL http://dx.doi.org/10.14778/2824032.2824086

[10] P. C. Arocena, B. Glavic, R. Ciucanu, R. J. Miller, The ibench integration metadata generator, PVLDB 9 (3) (2015) 108–119.

[11] L. Mazilu, N. W. Paton, A. A. Fernandes, M. Koehler, Dynamap: Schema mapping generation in the wild, in: SSDBM, New York, USA, 2019, pp. 37–48. doi:10.1145/3335783.3335785.
URL https://doi.org/10.1145/3335783.3335785

[12] C. Beeri, M. Y. Vardi, A proof procedure for data dependencies, J. ACM 31 (4) (1984) 718–741. doi:10.1145/1634.1636.
URL http://doi.acm.org/10.1145/1634.1636

[13] S. Melnik, E. Rahm, P. A. Bernstein, Rondo: A programming platform for generic model management, in: A. Y. Halevy, Z. G. Ives, A. Doan (Eds.), Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM, 2003, pp. 193–204. doi:10.1145/872757.872782.
URL https://doi.org/10.1145/872757.872782

[14] B. Alexe, B. ten Cate, P. G. Kolaitis, W. C. Tan, Characterizing schema mappings via data examples, ACM Trans. Database Syst. 36 (4) (2011) 23:1–23:48. doi:10.1145/2043652.2043656.

[15] A. Bonifati, U. Comignani, E. Coquery, R. Thion, Interactive mapping specification with exemplar tuples, in: ACM SIGMOD, 2017, pp. 667–682.

[16] G. Mecca, P. Papotti, S. Raunich, Core schema mappings: Scalable core computations in data exchange, Inf. Syst. 37 (7) (2012) 677–711.

[17] B. Marnette, G. Mecca, P. Papotti, S. Raunich, D. Santoro, ++spicy: An opensource tool for second-generation schema mapping and data exchange, PVLDB 4 (12) (2011) 1438–1441.
URL http://www.vldb.org/pvldb/vol4/p1438-marnette.pdf

[18] R. Fagin, P. G. Kolaitis, R. J. Miller, L. Popa, Data exchange: semantics and query answering, Theor. Comput. Sci. 336 (1) (2005) 89–124. doi:10.1016/j.tcs.2004.10.033.
URL https://doi.org/10.1016/j.tcs.2004.10.033

[19] R. Fagin, P. G. Kolaitis, L. Popa, Data exchange: Getting to the core, ACM Trans. Database Syst. 30 (1) (2005) 174–210. doi:10.1145/1061318.1061323.
URL http://doi.acm.org/10.1145/1061318.1061323

[20] G. Gottlob, A. Nash, Efficient core computation in data exchange, J. ACM 55 (2) (2008) 9:1–9:49. doi:10.1145/1346330.1346334.
URL http://doi.acm.org/10.1145/1346330.1346334

[21] B. ten Cate, L. Chiticariu, P. G. Kolaitis, W. C. Tan, Laconic schema mappings: Computing the core with SQL queries, PVLDB 2 (1) (2009) 1006–1017.

[22] L. Bellomarini, E. Sallinger, G. Gottlob, The vadalog system: Datalog-based reasoning for knowledge graphs, PVLDB 11 (9) (2018) 975–987. doi:10.14778/3213880.3213888.
URL http://www.vldb.org/pvldb/vol11/p975-bellomarini.pdf

[23] A. D. Sarma, X. Dong, A. Y. Halevy, Bootstrapping pay-as-you-go data integration systems, in: ACM SIGMOD, 2008, pp. 861–874.

[24] H. A. Mahmoud, A. Aboulnaga, Schema clustering and retrieval for multi-domain pay-as-you-go data integration systems, in: ACM SIGMOD, 2010, pp. 411–422.

[25] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, C. Yu, Finding related tables, in: SIGMOD, New York, NY, USA, 2012, pp. 817–828. doi:10.1145/2213836.2213962.
URL http://doi.acm.org/10.1145/2213836.2213962

[26] E. Zhu, F. Nargesian, K. Q. Pu, R. J. Miller, LSH ensemble: Internet-scale domain search, PVLDB 9 (12) (2016) 1185–1196. doi:10.14778/2994509.2994534.
URL http://www.vldb.org/pvldb/vol9/p1185-zhu.pdf

[27] F. Nargesian, E. Zhu, K. Q. Pu, R. J. Miller, Table union search on open data, PVLDB 11 (7) (2018) 813–825. doi:10.14778/3192965.3192973.
URL http://www.vldb.org/pvldb/vol11/p813-nargesian.pdf

[28] A. Bogatu, A. A. A. Fernandes, N. W. Paton, N. Konstantinou, Dataset discovery in data lakes, in: 36th IEEE International Conference on Data Engineering, ICDE, IEEE, 2020, pp. 709–720. doi:10.1109/ICDE48307.2020.00067.
URL https://doi.org/10.1109/ICDE48307.2020.00067

[29] K. Belhajjame, N. W. Paton, S. M. Embury, A. A. A. Fernandes, C. Hedeler, Incrementally improving dataspaces based on user feedback, Inf. Syst. 38 (5) (2013) 656–687. doi:10.1016/j.is.2013.01.006.
URL http://dx.doi.org/10.1016/j.is.2013.01.006

[30] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, S. Xu, Data curation at scale: The data tamer system, in: CIDR, 2013, pp. –.
URL http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper28.pdf

[31] R. C. Fernandez, et al., A demo of the data civilizer system, in: ACM SIGMOD, 2017, pp. 1639–1642.

[32] J. Yang, Y. He, S. Chaudhuri, Autopipeline: Synthesize data pipelines by-target using reinforcement learning and search, CoRR abs/2106.13861 (2021). arXiv:2106.13861.
URL https://arxiv.org/abs/2106.13861

[33] G. H. L. Fletcher, C. M. Wyss, Data mapping as search, in: EDBT'06, 2006, pp. 95–111. doi:10.1007/11687238_9.
URL http://dx.doi.org/10.1007/11687238_9

[34] S. Kruse, T. Papenbrock, F. Naumann, Scaling out the discovery of inclusion dependencies, in: BTW, 2015, pp. 445–454.
URL http://subs.emis.de/LNI/Proceedings/Proceedings241/article24.html

[35] H. Garcia-Molina, J. Widom, J. D. Ullman, Database System Implementation, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[36] A. V. Aho, J. E. Hopcroft, The Design and Analysis of Computer Algorithms, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[37] A. B. Kahn, Topological sorting of large networks, Commun. ACM 5 (11) (1962) 558–562. doi:10.1145/368996.369025.
URL http://doi.acm.org/10.1145/368996.369025

[38] J. Bleiholder, S. Szott, M. Herschel, F. Kaufer, F. Naumann, Subsumption and complementation as data fusion operators, in: EDBT '10, 2010, pp. 513–524. doi:10.1145/1739041.1739103.
URL http://doi.acm.org/10.1145/1739041.1739103

[39] P. C. Arocena, B. Glavic, R. J. Miller, Value invention in data exchange, in: SIGMOD, 2013, pp. 157–168.

[40] B. Marnette, G. Mecca, P. Papotti, Scalable data exchange with functional dependencies, Proc. VLDB Endow. 3 (12) (2010) 105116. doi:10.14778/1920841.1920859.
URL https://doi.org/10.14778/1920841.1920859

[41] F. Coelho, Datafiller 2.0.0 – data generation tool, Available:*https://www.cri.ensmp.fr/people/coelho/datafiller.html* [Accessed:2018-07-16] (2013).

[42] E. Abel, J. A. Keane, N. W. Paton, A. A. A. Fernandes, M. Koehler, N. Konstantinou, J. C. C. Ríos, N. A. Azuan, S. M. Embury, User driven multi-criteria source selection, Inf. Sci. 430 (2018) 179–199. doi:10.1016/j.ins.2017.11.019.
URL https://doi.org/10.1016/j.ins.2017.11.019

[43] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, M. Stonebraker, Aurum: A data discovery system, in: ICDE, 2018, pp. 1001–1012. doi:10.1109/ICDE.2018.00094.

[44] F. Psallidas, B. Ding, K. Chakrabarti, S. Chaudhuri, S4: top-k spreadsheet-style search for query discovery, in: T. K. Sellis, S. B. Davidson, Z. G. Ives (Eds.), Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 2001–2016. doi:10.1145/2723372.2749452.
URL https://doi.org/10.1145/2723372.2749452

[45] A. Bonifati, R. Ciucanu, S. Staworko, Learning join queries from user examples, ACM Trans. Database Syst. 40 (4) (2016) 24:1–24:38. doi:10.1145/2818637.
URL https://doi.org/10.1145/2818637