SCHEMA MAPPING GENERATION FOR AUTONOMOUS DATA SOURCES

A thesis submitted to The University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2019

By
Mihaela Lacramioara MAZILU
School of Computer Science
## Contents

Abstract ................................................. 9  
Declaration .............................................. 10  
Copyright ................................................ 11  
Acknowledgements ....................................... 12  

1 Introduction ............................................. 13  
   1.1 Motivation: Challenges and Opportunities .............. 13  
      1.1.1 Paradigmatic Scenarios ......................... 14  
      1.1.2 Challenges and Opportunities ................. 16  
   1.2 Schema Mapping Generation for Databases .............. 17  
   1.3 Outline and Contributions of the Thesis .............. 19  
      1.3.1 Thesis Contributions ......................... 19  
      1.3.2 Thesis Structure ............................. 21  

2 Related Work ........................................... 24  
   2.1 Schema Mappings for Databases ....................... 24  
      2.1.1 Data Model and Database Schemas ............... 24  
      2.1.2 Schema Mappings ............................... 26  
      2.1.3 Schema Mapping Languages ..................... 28  
      2.1.4 Schema Mapping Generation ..................... 30  
      2.1.5 Other Work on Schema Mapping Generation ......... 38  
   2.2 Schema Mappings for Data Exchange ................... 41  
   2.3 Schema Mappings in the Wild .......................... 43  
   2.4 Discussion ......................................... 47
3 Mapping Generation for a Simple Target

3.1 Preliminaries ............................................. 51
  3.1.1 Metadata, Statistics, and Profile Data .............. 51
  3.1.2 Schema Matching .................................. 53
  3.1.3 Merge Operators .................................. 55
  3.1.4 Merge Operators in Mappings ...................... 57
3.2 DYNAMAP - Overview of the Approach .................. 58
3.3 Profile-informed Merge Operator ...................... 61
3.4 Mapping Generation as a Dynamic Programming Problem .................. 65
3.5 Mapping Fitness .................................... 69
3.6 Profile Data Propagation ............................... 72
  3.6.1 Generalized Profile Data Propagation .......... 72
  3.6.2 Profiling Data Propagation Examples ............ 77
3.7 DYNAMAP Workflow ................................... 86
3.8 Algorithm Evaluation .................................. 88
  3.8.1 Benchmark Experiment - iBench ................. 89
  3.8.2 Real-world Scenario - Schools Domain .......... 92
3.9 Conclusions ........................................ 101

4 Mapping Generation at Scale .......................... 104

4.1 Algorithm Complexity .................................. 105
4.2 Profiling Data Propagation Complexity ................ 106
4.3 Pruning Strategies ................................... 112
  4.3.1 Preliminaries .................................. 112
  4.3.2 Pruning the Search Space ...................... 114
  4.3.3 Pruning the Profiling Data ...................... 119
  4.3.4 Pruning Strategies in DYNAMAP ................. 121
4.4 Integration Scenarios ................................ 122
  4.4.1 Integration Generators ......................... 122
  4.4.2 SYNTEGRATE ................................ 125
4.5 Scalability Evaluation ................................ 126
  4.5.1 Benchmark Experiment - iBench ................. 130
  4.5.2 Real-world Experiment - Real-estate Domain ...... 132
  4.5.3 Real-world Experiment - Schools Domain ........ 135
  4.5.4 Profiling Data Propagation - Accuracy at Scale .. 138
4.5.5 Instance Complementarity at Scale .......................... 140
4.5.6 Schema Complementarity at Scale .......................... 141
4.5.7 Instance & Schema Complementarity at Scale .............. 142
4.5.8 Efficiency of Pruning Strategies ........................... 143
4.6 Conclusions ....................................................... 148

5 Mapping Generation for a Complex Target ........................ 152
5.1 Example Scenario ............................................... 153
5.2 Overview of the Approach ...................................... 154
5.3 Universal Target Composition ................................. 158
5.3.1 Universal Target Graph ...................................... 158
5.3.2 Universal Target Relation ................................. 160
5.4 Universal Target Decomposition ................................. 161
5.4.1 Topological Sort using Kahn’s Algorithm ................ 162
5.4.2 Labelled Nulls Generation ................................. 163
5.4.3 Subsumed Output Tuples ................................. 170
5.5 Mapping Characteristics ...................................... 173
5.5.1 Attribute Completeness ..................................... 175
5.5.2 Source-extracted Values Ratio ............................ 176
5.5.3 Key Consistency ........................................... 177
5.6 Algorithm Evaluation ........................................... 179
5.6.1 Benchmark Experiment - IBench ......................... 182
5.6.2 Real-world Experiment - Real-estate Domain ........... 184
5.6.3 Real-world Experiment - Schools Domain ............... 195
5.7 Conclusions ....................................................... 205

6 Conclusions and Open Issues ..................................... 208
6.1 Concluding Discussion ........................................... 208
6.1.1 Impact of Dynamap ........................................ 213
6.2 Open Technical Challenges .................................... 214
6.2.1 Mapping Fitness ........................................... 214
6.2.2 Characterizing Mappings without Materialization .... 214
6.2.3 Finding Merge Opportunities ............................ 216
6.3 Future Work ...................................................... 216
6.3.1 Adapting to More Data Formats ......................... 216
6.3.2 Mapping Generation Reuse ............................... 217
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.3 Mapping Generation Refinement</td>
<td>218</td>
</tr>
<tr>
<td>6.3.4 Processing Optimization</td>
<td>219</td>
</tr>
<tr>
<td>Appendices</td>
<td>232</td>
</tr>
<tr>
<td>A Helper Functions</td>
<td>232</td>
</tr>
<tr>
<td>A.1 CommonAncestors</td>
<td>232</td>
</tr>
<tr>
<td>A.2 DiffMatches</td>
<td>232</td>
</tr>
<tr>
<td>A.3 FindConnGraphs</td>
<td>233</td>
</tr>
<tr>
<td>A.4 FindMatchesAttr</td>
<td>233</td>
</tr>
<tr>
<td>A.5 FindMatchedKeys</td>
<td>233</td>
</tr>
<tr>
<td>A.6 FindPKey</td>
<td>234</td>
</tr>
<tr>
<td>A.7 FindKeys</td>
<td>234</td>
</tr>
<tr>
<td>A.8 IsFittest</td>
<td>234</td>
</tr>
<tr>
<td>A.9 IsSubsumed</td>
<td>235</td>
</tr>
<tr>
<td>A.10 MaxInd</td>
<td>236</td>
</tr>
<tr>
<td>A.11 SameMatches</td>
<td>236</td>
</tr>
<tr>
<td>B Synthegrate</td>
<td>237</td>
</tr>
<tr>
<td>B.1 Synthegrate Workflow</td>
<td>238</td>
</tr>
</tbody>
</table>

Word Count: 57035
List of Tables

3.1 Cardinality estimations .................................................. 73
3.2 Inclusion dependencies propagation for *lossless* attribute merges . 76
3.3 Inclusion dependencies propagation for *lossy* attribute merges . 77
3.4 Candidate keys propagation ............................................. 77
3.5 Results for iBench primitives .......................................... 91
3.6 Input source files - schools information .............................. 93
3.7 Schools scenario - merges in iteration 2 .............................. 95
3.8 Schools scenario - merges in iteration 3 ............................. 98
3.9 Schools scenario - merges (*SS*₂, *SS*₃) in iteration 4 .......... 100
3.10 Schools scenario - merges (*SS*₁, *SS*₃) in iteration 4 .......... 100

4.1 Web-extracted and open government datasets ....................... 132
4.2 Input source files - schools information ............................. 136
4.3 Error ranges ..................................................................... 139
4.4 Impact of pruning strategies for the search space – number of
generated mappings (existed in the search space at some point
during the mapping generation process) compared to number of
pruned mappings ................................................................. 145
4.5 Impact of pruning strategies for the profiling data – number of
generated inclusion dependencies (inferred at some point during
the mapping generation process) compared to number of pruned
inclusion dependencies .......................................................... 146

5.1 Manchester – example tuples ............................................. 166
5.2 Area Info tuples ............................................................... 167
5.3 UK Realestate tuples ........................................................ 167
5.4 Area Info tuples ............................................................... 170
5.5 UK Realestate tuples ........................................................ 170
List of Figures

1.1 Instance vertical portal example ........................................ 14
1.2 Schema horizontal portal example ..................................... 15

3.1 Mapping generation scenario ........................................... 60
3.2 Lossless merges - attributes overlaps ................................ 79
3.3 Lossy merges - attributes overlaps ................................... 84
3.4 Dynamap – Mapping generation workflow ............................. 87
3.5 Schools scenario ............................................................. 93

4.1 Dynamap – Mapping generation workflow with pruning strategies 122
4.2 Dynamap compared to ++Spicy on complex iBench scenarios with various numbers of constraints ................................. 131
4.3 Performance of Dynamap and ++Spicy on a real-estate scenario . 135
4.4 Performance of Dynamap and ++Spicy on a schools scenario ..... 137
4.5 Runtime of mapping generation for synthetic data scenarios .... 140
4.6 Effectiveness of the pruning strategies for the search space .... 144
4.7 Effectiveness of the pruning strategies for the profile data ..... 146

5.1 Target schema with key and foreign key constraints ............... 153
5.2 Dynamap$^\text{X}$ workflow – approach for target schema with constraints 156
5.3 Topological order example ............................................... 168
5.4 Dynamap compared to ++Spicy on iBench scenarios (rich target schemas) ......................................................... 183
5.5 Real-estate target schema .................................................. 186
5.6 Performance of Dynamap$^\text{X}$ and ++Spicy on real-estate data . 189
5.7 Attribute characteristics on real-estate data .......................... 191
5.8 Schools data - target schema .............................................. 196
5.9 Performance of Dynamap$^\text{X}$ and ++Spicy on a schools scenario 199
5.10 Attribute characteristics on schools data ............................... 202
Abstract

The world is producing digital data at a rapid pace and this led to a new way of seeing data as Big Data. Big Data refers to large numbers of large datasets that typically include complex, various, rapidly-changing data of uncertain quality that (usually) need preprocessing before analysis. Data integration over Big Data gives rise to the challenge of correlating unruly and heterogeneous repositories of data sources.

In this thesis, our focus is on integration techniques for Big Data, more specifically on generating mappings over large repositories of heterogeneous and autonomous datasets. A schema mapping generation algorithm constructs views for populating a target database schema from source schemas. We have designed, developed, and validated techniques for generating schema mappings over autonomous data sources for which scant information is available, and for complex multi-relation, constrained target schemas, at scale. Our proposed algorithm is called $Dynamap^{(X)}$ and has at its core the dynamic programming paradigm for performing the search over the space of mappings. The mappings are built in a bottom-up fashion, where the merge operators are chosen based on profiling information on the sources, i.e., candidate keys and (partial) inclusion dependencies.

We have employed $Dynamap^{(X)}$ in three main types of experiments: (i) with the state-of-the-art integration scenario generator, thus, showing that it can handle scenarios that are expected to be tackled by mapping generation algorithms; (ii) with variations of real-world scenarios that come from different domains with autonomous sources, showing that it can handle integration problems from real datasets; and (iii) with stress-test scenarios showing that our algorithm can handle scenarios where the input comprises hundreds of data sources.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to thank everyone that has helped shape the ideas of this thesis by either sharing their technical insights or by supporting and encouraging me to keep on going.

I had the great opportunity to be guided and supervised by Prof. Norman W. Paton and Dr. Alvaro A.A. Fernandes. I would like to express my deep gratitude to both of them as this thesis would not have been possible without their constant support, ideas, suggestions, and advice. During these four years, I have learned many things from them, among which, how to ask questions, how to find ideas, how to shape them, how to present them, but most importantly, they taught me how to trust myself that I can always find a solution.

I am extremely grateful for the love and support I received from my future husband, Alex, as without him I might have never finished (or started) this journey. I feel lucky we had the chance to share this adventure together. Moreover, I am forever indebted to my family and friends, especially my parents, Costica and Nuta Mazilu, for their unconditional love and support throughout every decision I have made, and my sister, Madalina, who encouraged me to pursue my doctoral studies abroad.

Last, but not least, I would like to thank my colleagues and friends in the VADA Project group for their help and collaboration. Also, I appreciate all the opportunities and the welcoming environment provided by the Information Management Group and the School of Computer Science at The University of Manchester as this helped me complete my degree.
Chapter 1

Introduction

"Take the first step in faith. You don’t have to see the whole staircase, just take the first step."
– Martin Luther King Jr. (1929 - 1968)

1.1 Motivation: Challenges and Opportunities

The world is producing digital data at a rate that is rapidly increasing due to many technological trends, such as user interaction on social media platforms, and machine-generated data. For instance, it was reported in Halevy et al. (2016) that Google has indexed 26 billion datasets and this number keeps growing. This has led to a new way of seeing data as Big Data. Big Data refers to large numbers of large datasets that typically include complex, various, rapidly changing data of uncertain quality that (usually) need pre-processing before analysis. Big Data comes with its challenges which many refer to as the three Vs [Furht and Villanustre (2016); Oussous et al. (2018)]:

1. **Volume** refers to the size of the data, i.e., instances and/or number of sources.

2. **Variety** refers to the formats in which the sources come, e.g., diverse structured representations of the data, as well as their domain.

3. **Velocity** refers to the pace at which information is generated, stored, analyzed, and managed by processes. With every new piece of information, the format and schema of the generated data can be different from previous versions making it all too fast to follow manually.

According to Furht and Villanustre (2016), in addition to the above mentioned
three Vs, there is (at least) one more V, for value, referring to the benefit the user can obtain from Big Data.

In this thesis, broadly speaking, our focus is on integration techniques for Big Data coming from autonomous sources. There are several surveys that present methods for the data integration process [Chen et al. (2014); Ali et al. (2016)]. Data integration has the goal of providing the user with a unified view over local or remote, typically autonomous, data sources. The overall outcome is usually a view containing information about entities which are more coherently modelled as the data is extracted from multiple sources. However, considering the above Vs, data integration over Big Data gives rise to the challenge of correlating data over large numbers of heterogeneous data sources in a constantly-changing, and unruly repository. Nonetheless, data integration is an essential step preceding any data analysis, which often relies on usefully correlated information.

1.1.1 Paradigmatic Scenarios

To motivate the need for schema mapping generation for autonomous data sources, which is the narrower focus of this thesis, we use two example scenarios illustrated in Figures 1.1 and 1.2. These exemplify two of the abstract patterns found in many data integration applications, viz., instance verticality and schema horizontality.

![Figure 1.1: Instance vertical portal example](image)

**Example 1.1.1.** Instance vertical portal. By instance vertical we mean that the outcome of integration is, broadly speaking, the iterated union of the sources.
1.1. MOTIVATION: CHALLENGES AND OPPORTUNITIES

Consider a real-estate aggregator company that needs to supply information for data analysis on house prices to a business partner. The aggregator needs to deliver data that is sourced from different locations (from different suppliers) with unrelated schemas, i.e., schemas that have different names and structures, but broadly store the same kind of information, e.g., postcodes, addresses, prices, etc.

Figure 1.1 depicts an example of the above scenario, i.e., there are three relational schemas, Oxford, London and Manchester Realestate, representing different web-extracted data sources that contain the same kind of information. The data that is required by the business partner is specified by means of a single-table target schema UK Realestate: city name, postcode, street name, and price.

Notice that the aggregator company would need to devise techniques to identify the relevant sources of required data (e.g., ensure that all sources contain the attributes needed), and correlate the (often schema-incompatible) data prior to delivering it as a single dataset to the partner, who need not be aware that the data comes from different sources. Here, roughly speaking, each of the sources contains subsets of the tuples that should appear in the target.

Example 1.1.2. Schema horizontal portal. By schema horizontal we mean that the outcome of integration is, broadly speaking, the iterated join of the sources.

Consider a similar scenario to the one in Example 1.1.1 where a company delivers information to a business partner, but the information needed by the
partner is contained in different data sources that do not contain the same kind of information, i.e., the sources are schema complementary w.r.t. the required data.

Figure 1.2 depicts an example of the above scenario, i.e., different source schemas contain parts of information that is required by the business partner.

Notice that the aggregator company would need again to devise techniques to identify the relevant sources of data, merge the data using appropriate attributes prior to delivering it as a single dataset to the partner, who again need not be aware that the data comes from different sources. Here, roughly speaking, each of the sources contains subsets without the partner being aware of the data behind the view that is served to them.

Furthermore, to complicate matters, we can assume that the source schemas used by the company undergo periodic changes due to the dynamic nature of its business, e.g., it may add attributes such as managing agency, construction year, or currency that have become of interest for the business partner. Given these changes in required data, an expert user in charge of delivering the data would need to review all data sources and go through the same process of generating the dataset desired by the partner.

In both examples, an expert must find the relevant attributes in the sources (e.g., in Example 1.1.2, UKDeprivation.Crimerank information is needed in the target schema as data values in UKRealestate.CrimeRank). Also, the expert must find patterns (e.g., relationships) in the source data that enable the appropriate merging of the relevant datasets. Then, taking the identified source attributes into account as well as the inferred relationships, the expert must design queries that, when evaluated, transform the data in the sources into data that can populate the target (e.g., in Example 1.1.2, tables ManchesterRealestate and UKDeprivation need to merge to correlate the realestate entries with the corresponding crimerank information, and, if there is value overlap, a sensible way of merging would be to join them by using the postcode attributes in the two tables).

1.1.2 Challenges and Opportunities

Examples 1.1.1 and 1.1.2 hint at the challenges tackled by experts in charge of delivering an integrated view on multiple autonomous, heterogeneous sources. These can be generalized as follows:
1.2. SCHEMA MAPPING GENERATION FOR DATABASES

1. How to manage a (potentially large) repository of data sources and identify the relevant sources for the required data?

2. How to infer relationships between the (initially unrelated) data sources so as to correlate the data, especially for scenarios such as in Example 1.1.2, where it is plausible for the merged sources to have completely different schemas and domains?

3. How to find, given the inferred relationships, a set of executable queries that, when run over the pool of relevant sources, generate an integrated view that correlates data as specified in a target schema?

4. How to deal with the problem of constantly-changing data and schemas that cause one to have to review and redesign the view to fit the new target?

In a setting with a large volume of data sources, which are constantly changing in terms of either stored information or schema format, manually creating the required view in the light of all the above challenges becomes unfeasible. This shows the growing relevance of a solution to the problem of automating the process of creating views in the format of a target schema given large heterogeneous source repositories in the context of volume, variety, and velocity of Big Data. In this thesis, we contribute to this solution. We address data integration challenges in the context of the three V’s by proposing a set of techniques to correlate repositories of independent and diverse data thereby adding value to the available data.

1.2 Schema Mapping Generation for Databases

**Schema mapping.** Schema mapping is an essential operation in any data integration process whereby, when executed, the mapping transforms data under the source schemas into data conforming to a target schema [Lenzerini (2002)]. More specifically, a schema mapping is an executable transformation that specifies the manner in which the tuples from the source schemas should be used to generate tuples in the target.

**Mapping generation.** Schema mapping generation is the data management task that, given a set of sources and a target schema, generates schema mappings. The generated mappings are in a logical format that expresses the relationship between the source and target schemas. Thus, mappings are commonly expressed
as *source-to-target tuple generating dependencies* (*s-t tgds* – described in Section 2.1.3) which can, in turn, become executable queries expressed in query languages (e.g., SQL), depending on the underlying storage systems [Fagin et al. (2005)].

**Available knowledge for mapping generation.** Previous schema mapping generation approaches take as input at least one source schema to be mapped to one target schema, schema matches (also known as correspondences) and (possibly) schema metadata (e.g., schema constraints such as primary keys and foreign key relationships) to create mappings w.r.t. the chosen target format [Miller et al. (2000); Popa et al. (2002); Mecca et al. (2009)]. Prior work, such as the Clio project [Fagin et al. (2009)], which has been extremely influential in this area, consider the input as containing one source schema and one target schema, where the source schema is well-behaved, by which is meant that it has been designed according to relational theory (in the typical cases) and that all design constraints are explicitly known. Thus, it is often the case that the creation of explicit join paths is done through the use of semantic constraints such as key or foreign key constraints. In the absence of such schema metadata, merge opportunities can be overlooked.

**Example 1.2.1.** Consider again Examples 1.1.1 and 1.1.2, in Figures 1.1 and 1.2. For the purpose of creating the desired views, an expert creates a set of *schema matches* between the sources and the desired target potentially with tool support, so as to specify the relevant data in the sources (e.g., in Example 1.1.2, *UKDeprivation.CrimeRank* matches *UKRealestate.CrimeRank*, and *UKLQI.IncomeRank* from *UK Life Quality Indices* table matches *UKRealestate.IncomeRank*, etc). Then, these matches are used to design the appropriate *schema mappings* (e.g., in Example 1.1.2, sources *UKDeprivation* and *UKLQI* need to be merged following a (set of) join path(s) that are either inferred or explicitly declared). However, in neither of the Examples 1.1.1 or 1.1.2 there are explicit schema constraints, although if there is a value overlap at attribute level, two tables (e.g., the two aforementioned tables, *UKDeprivation* and *UKLQI*) might be merged by performing a relational equijoin on the *county* attributes in each of the two tables.

**Schema mapping generation use cases.** The use cases of schema mapping generation are, generally, in data management processes such as data integration [Lenzerini (2002)], data exchange [Kolaitis (2005); Fagin et al. (2009)] (a.k.a.
1.3. OUTLINE AND CONTRIBUTIONS OF THE THESIS

data translation [Shu et al. (1977)], peer-to-peer data sharing [McBrien and Poulavassilis (2003b); Fuxman et al. (2006)], schema integration [Batini et al. (1986)], and schema evolution [Yu and Popa (2005); Curino et al. (2008)]. The importance of data management tasks, especially data integration, was recognized decades ago in works such as the ones carried out by Shu et al. (1977), Casanova and Vidal (1983), and Batini et al. (1986). However, much has evolved in the past years because the constant increase in digital data gave rise to new challenges which were addressed, e.g., by Halevy et al. (2006); Bernstein and Haas (2008); Dong et al. (2009) and Golshan et al. (2017) who focus on issues such as the need to integrate data without having to materialize all sources in a single place, new data management requirements in organizations, manipulating uncertain data, or sharing information between data services. Moreover, Dong and Srivastava (2013) describe the integration challenges in terms of Big Data Vs, highlighting the need to merge disparate, large, and heterogeneous sources. Although these works describe both old and new challenges, they all rely on the same common step of bringing sources together in a unified view through schema mappings. The distinction comes from the fact that the mapping generation problem changes its context according to the constant evolution of the data landscape.

1.3 Outline and Contributions of the Thesis

In this section, we state our main objectives and sketch the overall approach to the schema mapping generation problem explored in the dissertation, i.e., *schema mapping generation for autonomous data sources*. In our work, we set three main objectives, and each of them is described together with proposals for tackling it that constitute our research contributions.

The overall aim of this thesis is to design, develop, and validate techniques for generating schema mappings over autonomous data sources for which scant information is available, and for complex multi-relation, constrained target schemas, at scale.

1.3.1 Thesis Contributions

This thesis has three main objectives:

**Objective 1:** Mapping generation for a single-relation target schema – to
merge sources that do not have any explicitly declared relationships (i.e., declared foreign keys that would facilitate the creation of the join paths) w.r.t. a target schema with one target relation and no constraints.

**Contribution 1.1:** We have devised a decision procedure that combines pairs of candidate mappings, informed by intra-source and inter-source information, viz. relational metadata and profile data [Abedjan et al. (2015)], respectively.

**Contribution 1.2:** We have designed, implemented and evaluated a *dynamic programming* algorithm that explores the space of candidate mappings, identifying opportunities for combining source relations building on *Contribution 1.1.*

**Contribution 1.3:** We have designed and implemented a method for inferring profile data for candidate mappings without materializing their results. The proposal in *Contribution 1.1* needs such information at each step of the dynamic programming iteration proposed in *Contribution 1.2*. Thus, inferring and propagating profile data without materialization becomes paramount.

**Objective 2:** *Mapping generation at scale* - to generate mappings over a large set of input sources.

**Contribution 2.1:** We have designed, implemented and evaluated several pruning strategies for a search space of mappings such that they filter the candidate mappings keeping only those that promise to yield desirable outcomes.

**Objective 3:** *Mapping generation for a multi-relation target schema with constraints* – to extend *Objectives 1* and *2* in order to encompass target schemas with multiple relations that have schema constraints and thereby create and select mappings that violate as few target constraints as possible.

**Contribution 3.1:** We have designed, implemented and evaluated an algorithm that populates a multi-relation target schema where constraints such as candidate keys and foreign keys are satisfied to the greatest possible extent when populating the corresponding attributes.

**Contribution 3.2:** We have designed, implemented and evaluated a mechanism for characterizing the generated mappings based on the degree of target constraints violation. This enables other data management modules (such as mapping selection) to rank the mappings in terms of their suitability for populating a given target.

The contributions above give rise to a comprehensive mapping generation system, which we have evaluated empirically. In doing so, we have (i) explored its
applicability on simple and complex benchmark scenarios generated by the state-of-the-art mapping generation benchmark (iBench); (ii) explored its performance on open-government and web-extracted data from two real-world domains; and (iii) explored its scalability on generated scenarios involving large numbers of sources populated with synthetic data.

1.3.2 Thesis Structure

The structure of the thesis closely follows the above stated objectives and contributions:

Chapter 2 describes the background and the related work on schema mapping generation which are necessary for understanding the research context under which this work has been developed and the contributions of this thesis.

In Chapter 3, we describe our proposal for mapping generation given many source schemata and a single-table target schema without schema constraints. This chapter contains (i) the description of a decision procedure that combines pairs of candidate mappings, informed by profiling data, (ii) the description of a dynamic programming algorithm that explores the space of candidate mappings to incrementally identify opportunities for combining subsets of source relations on the basis of profiling data, (iii) a description of an approach to infer the profiling data for the result of a relational operator application from the profiling data of its operands that removes the need to materialize intermediate mappings, and (iv) an evaluation of the resulting algorithm, called Dynamap, against a set of primitive scenarios [Arocena et al. (2015)] built using the state-of-the-art data integration benchmark iBench, and against a scenario stemming from a larger real-world domain. In these evaluation scenarios, we measure the quality of the data generated by the output mappings.

In Chapter 4, we describe an approach to tackling the problem of unifying a large set of large data sources, i.e., mapping generation at scale. We first analyze the complexity of the mapping generation algorithm described in Chapter 3 which leads to a challenge of run-time efficiency. In order to explore a large space of candidate mappings using dynamic programming, we describe a set of pruning strategies which aim to keep control over the search space and, as a result, on the growth of the running time. For evaluating the effectiveness of the pruning strategies, we describe an integration scenarios generator, called Synthegrate, which we use to create large scale scenarios against which
we can stress-test the mapping generation algorithm. We conduct an empirical evaluation that explores different aspects of the pruning strategies:

- **Evaluation against scenarios generated using iBench.** We show that, even though the pruning strategies discard candidate mappings, Dynamap can still tackle complex scenarios comprising merged integration scenarios.

- **Evaluation against real-world scenarios.** We use two real-world domains (one of which is the extended scenario of the one in Chapter 3) to show that there are real-world scenarios for which, compared to a ground-truth mapping, the generated output mapping generates satisfying results.

- **Accuracy of profiling data propagation.** We again use the real-world scenarios to measure the accuracy of the propagation method for profiling data. As the propagation results are estimations of the real values, we measure the accuracy of the results by comparing the estimations against the accurately-determined results of a profiling tool that was run on the forced materialization of the intermediate mappings.

- **Efficiency of the pruning strategies on large scale scenarios.** We use SYN-TEGRATE to generate various large scale scenarios and stress-test the algorithm for running time.

- **Effectiveness of each pruning strategy.** We show the effectiveness of the pruning strategies by measuring the run-time with each pruning strategy active so as to quantify how each strategy impacts on the overall run-time.

In Chapter 5, we describe a proposal, called $Dynamap^{(e)X(tended)}$, for schema mapping generation between many source schemata and a multi-table target schema with constraints. The main contribution of this chapter is a method to alter the mappings generated by the algorithm described in Chapters 3 and 4 so that they generate data for each of the desired target tables taking into account the target constraints.

To evaluate Dynamap, we conduct a similar evaluation to the ones in Chapters 3 and 4, where we analyze the behaviour of the algorithm on integration scenarios generated with iBench, and, on real-world scenarios on target schemas with constraints, measuring the quality of the output data with reference to the ground-truth.
In Chapter 6, we briefly discuss how our contributions fit into the research area of mapping generation and we outline possible future directions for schema mapping generation over autonomous data sources which have not been addressed and seem to be suitable next steps stemming from the contributions and evaluation reported here.
Chapter 2

Related Work

"Progress lies not in enhancing what is, but in advancing toward what will be."

– Khalil Gibran (1883 - 1931)

Schema mapping generation has been the subject of significant research and development effort. In this chapter, we review work on mapping generation for databases, its context in data exchange, and schema mapping generation in the wild, i.e., over autonomous sources.

2.1 Schema Mappings for Databases

In comparing related mapping generation proposals, it is also important to consider the data model used to describe the source and the target schemas, and the language in which the mappings are expressed, thus, we briefly discuss these before discussing the related work on mapping generation.

2.1.1 Data Model and Database Schemas

A data model is used to represent a formal description of information. This description comprises the structure of the data, a.k.a., the conceptual model; possible operations on the data, i.e., the types of queries for retrieving or modifying the data and that can be run over the structure of the data; and the constraints on the data, i.e., the description of the limitations imposed on the data to be stored. The two most commonly-used data models for database systems are [Garcia-Molina et al. (2008)] the relational model and the semi-structured data model.
Prior work on mapping generation has been on both relational and semi-structured data models. The work presented in this thesis focuses on the relational model, however, it could be extended to the semi-structured data model, as it will be presented as an open technical challenge in Chapter 6. We describe below the relational data model.

### 2.1.1.1 The Relational Model

**Data Structure.** The relational model is based on relations (i.e. tables) which are two-dimensional data structures that have field headers representing attributes (table columns). Under each attribute one can find values, and a row of attribute values represents a tuple. A relational schema comprises a set of relations. The values that a relational database stores are either constants or nulls, where the latter can be labelled nulls. The notion of labelled nulls (a.k.a. skolems) has been described in works such as Hull and Yoshikawa (1990), where Skolem functions are discussed as a means for the invention of unique values. The usefulness of these invented values for generating solutions in integration scenarios has been recognized and described among others Mecca et al. (2009).

The structure of a relational database can be formalized as [Garcia-Molina et al. (1999)]:

- Attribute names (columns names) are represented by $A = \{A_0, A_1, \ldots, A_m\}$. Under the label of an attribute $A_j \in A$, there is a set of values which can be either a constant or a (labelled) null. The number of distinct values in an attribute $A_j \in A$ is denoted $V(A_j)$.

- A set of relation names (table names) is represented by a set $R = \{R_0, R_1, \ldots, R_N\}$, where each relation $R_i \in R, i \in \{0..N\}$, contains a set of attributes $A$, defining its schema and denoted by $R_i(A_0, A_1, \ldots, A_{m_{R_i}})$. We say $m_{R_i}$ is the arity, i.e., the number of attributes, of relation $R_i$.

- An instance of a relation $R_i$ is a set of tuples (rows), where a tuple is of the form $R_i(A_0 : v_0, A_1 : v_1, \ldots, A_{m_{R_i}} : v_{m_{R_i}})$, where each value $v_i \in \{v_0, v_1, \ldots, v_{m_{R_i}}\}$ can be either a constant or a (labelled) null. We say that the total number of tuples of relation $R_i$ is its cardinality denoted by $|R_i|$.

- A set of database schemas is represented by a set $S = \{S_0, S_1, \ldots, S_k\}$, where each schema $S_i \in S, i \in \{0..k\}$, comprises a set of relations $S_i =$
\{R_0, R_1, ..., R_{N_S}\}.

- Given a database $D$, $D$ can contain one or more schemas, thus, $D = \{S_0, S_1, S_2\}$.

**Operators.** The operations that can be typically performed under the relational model are:

1. Binary set operators on relations: *union*, *intersection* and *difference*;
2. Unary operators that (partially) retrieve data in the relations: *selection*, which selects some of the rows of a relation, and *projection*, which selects some of the columns of a relation;
3. Binary operators that merge the data in two relations: *join*, which merges the tuples in two relations based on the values in a set of condition attributes, and *cartesian product* which merges, pairwise, all tuples in both argument relations;
4. Unary operators that change the relation schema by renaming the attributes and/or the relation.

These operators are usually formalized in *relational algebra*, which we discuss in Section 3.1. In Section 3.3, we discuss under which settings we use a set thereof to merge input relations to create schema mappings.

**Constraints.** The relational constraints describe the type of data and the values that can be stored by a relational model. For schema mapping generation, the constraints that are most relevant are *key* and *foreign key* constraints that are set on individual and on pairs of attributes, respectively. In Section 2.1.4, we discuss related work that uses schema constraints in mapping generation.

### 2.1.2 Schema Mappings

The creation of schema mappings is an essential step in processes such as data integration, where the data residing in multiple sources needs to be merged and transformed so as to populate a desired target. Schema mappings are executable transformations that specify the manner in which the tuples from the source schemas are used to generate tuples in the target. The chosen target is usually referred to as a *global schema*, or a *mediated schema*, and it represents the unified
virtual view that will contain the merged source information. In Lenzerini (2002),
there are described and compared three proposed approaches to model the rela-
tionship between the target, i.e., the global schema that would contain the virtual
data, and the sources, which contain the original data. One of the approaches is
global-as-a-view, where the target is expressed in terms of data sources; the second
approach is named local-as-a-view, where each source is defined as a view over
the global schema, with the global schema being independently defined from the
sources, and the third is global-and-local-as-a-view, which is a combination of both
preceding approaches. These models are used in tasks such as query answering
[Halevy (2000)] so that it can be determined how a query executed on a data
integration system delivers the data. We now briefly discuss the three types of
schema mappings.

Global-as-a-view (GAV). In the GAV model, each global relation has an
associated mapping which specifies how to retrieve the data from the sources to
the target. This model is suitable when the sources are less likely to change, and
a flexible global schema is desirable, thus, every change that the global schema
might suffer needs to be reflected in modifications to the mappings between the
sources and the new global schema. However, if the pool of sources is enriched by
adding new elements, then the already defined mappings need to be refined such
that the new sources are taken into consideration if they are relevant for the view
that the global schema encompasses.

Local-as-a-view (LAV). In this context, the global schema is defined inde-
dependently from the sources, but the sources are specified in terms of the global
schema and its concepts. Such an approach is desirable when the global schema
is unlikely to change, while the sources can be changed as often as needed by
either altering their schemas or removing sources completely. Such source schema
changes would not alter the global schema, however, the mappings between the
sources and the global schema would need to be changed according to the source
modifications.

Global-and-local-as-a-view (GLAV). In more recent works such as [Dong
and Srivastava (2013)], there has also been a focus on a combination of GAV and
LAV, namely global-and-local-as-a-view (GLAV) mappings. This model specifies
both the mediated data and the local data as views of data of a virtual schema.
The GLAV mappings are usually named tuple-generating dependencies which we
discuss in more detail in Section 2.1.3.
Other works, such as the one undertaken by McBrien and Poulavassilis (2003a), have built on the above approaches and propose both-as-a-view (BAV) where, similar to GLAV, they combine global-as-a-view and local-as-a-view such that, with their proposed method, it is possible to define a view of the global schema using the sources, and, at the same time, it is possible to define a view of the local schemas using the global one, i.e., they treat both global and the local schemas as sources. Additionally, they introduce a method for updating the mediated schema based on the integration of the newly added sources, and they do this by creating a mapping that alters the attributes and the relations (add, delete, or rename) in the mediated schema. With the BAV approach, schema evolution on either the sources’ part or the global schema part are supported, providing a framework for schema transformation and integration.

2.1.3 Schema Mapping Languages

A mapping specification language needs to accurately reflect the transformation from one (set of) schema(s) to another. We now briefly discuss on a formalism to express transformations between a source schema and a target schema, e.g., that encompasses GAV, LAV, and/or GLAV.

2.1.3.1 Tuple-generating Dependencies (tgds)

Tuple-generating dependencies (tgds) and equality-generating dependencies (egds) are two types of database dependencies with which one can express either relationships between relational database components, or constraints on them [Beeri and Vardi (1984)]. They were first used for database design, but, in the past decades, tgds and egds have been used to express schema mappings and for data exchange [Fagin et al. (2005)].

Given a relation $R(A_0, A_1, ..., A_m)$, where $A \in \{A_0, A_1, ..., A_m\}$ are the attributes of $R$, a tgd is a first-order formula of the form:

$$\forall A(\phi(A) \rightarrow \exists Y\psi(A,Y))$$

where $A$, and $Y$ are sets of attributes, and :

- $\phi(A)$ is a conjunction of atomic formulas, each formula being over attributes in $A$,

- $\psi(A,Y)$ is a conjunction of atomic formulas, each formula having variables from $A$ and $Y$,
2.1. SCHEMA MAPPINGS FOR DATABASES

- all variables $A_i \in A$ appear in $\phi(A)$, but not necessarily in $\psi(A,Y)$.

**Source-to-Target tuple-generating dependencies.** A commonly used formalism to express mappings, i.e., transformations between a source and a target, is *source-to-target tuple-generating dependencies* ($s$-$t$ tgds). These are expressions that describe which tuples in the source should also appear in the target. Thus, in the context of using tgds to express schema mappings:

- $\phi(A)$ is a conjunction of atomic formulas over the source schema(s), where $A$ is a set of source attributes,

- $\psi(A,Y)$ is a conjunction of atomic formulas over the target schema, where $A$ is a set of target attributes bound to the source attributes, and $Y$ a set of unbound target attributes.

**Example 2.1.1.** For the example in Figure 1.2, the following tgd would transform tuples from the format of *UK Deprivation* source to the target format of *UK Realestate*:

$$\forall lc, cr, cd, co, pc : UKD(lc, cr, cd, co, pc) \rightarrow \exists I, S, P : UKR(I, cr, pc, S, P)$$

This mapping expresses that for all tuples in $UKD$, copy the crime-rank ($cr$), and postcode ($pc$) values into the target ($UKR$), with income ($I$), street ($S$) and price ($P$) left as nulls.

**Source tgds and Target tgds.** These expressions usually represent the constraints in, resp., the source and the target when there are, e.g., foreign key constraints. In the context of schema mapping generation, they are part of the input metadata from the source and/or target schemas.

**Example 2.1.2.** In Figure 1.2, we mentioned that for merging two of the source relations, one might need to infer some relationships between them, such as foreign keys. Let us assume that we infer a foreign key relationship between *UK Deprivation* and *Manchester Realestate* (both are sources) as we can assume that the postcode values in the open-government data source will fully contain the postcode values in the realestate source. The tgd that expresses this foreign key constraint is:

$$\forall le, a, bcy, c, pc, s, bn, pr : MR(le, a, bcy, c, pc, s, bn, pr) \rightarrow$$

$$\exists LC, CR, CD, CO : UKD(LC, CR, CD, CO, pc)$$
This expression represents a foreign key constraint: for every postcode (pc) value in MR there exists one postcode (pc) value in UKD which is equal to it, i.e., all the values in MR.pc are contained in the values of UKD.pc. This is the full containment condition that needs to be satisfied between two attributes that share a foreign key constraint.

2.1.3.2 Equality-generating Dependencies (egds)

Equality-generating dependencies are used to represent key constraints, which, in the context of schema mapping generation, are part of the input metadata on the source and/or target schemas.

Example 2.1.3. In Figure 1.2, again, suppose one wants to express that the postcode attribute in UK Deprivation is a candidate key, then the egd that expresses this key constraint is:

\[ \forall lc, lc', cr, cr', cd, cd', co, co', pc : \]
\[ UKD(lc', cr', cd', co', pc) \land UKD(lc, cr, cd, co, pc) \rightarrow \]
\[ (lc = lc') \land (cr = cr') \land (cd = cd') \land (co = co') \]

The egd above expresses that if there are two tuples in relation UKD that have the same values on postcode (pc), then all the other corresponding values in the two tuples are the same, essentially, stating that the postcode uniquely identifies every tuple in relation UKD.

2.1.4 Schema Mapping Generation

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, with the constant growth in available datasets that need integration [Bernstein and Haas (2008)], it has become increasingly relevant to automate schema mapping generation. Schema mapping generation has been the subject of significant research and development effort. In this section, we review work on mapping generation for databases.

Definition. A schema mapping generator has the signature \( \text{MapGen}(P) \rightarrow M \), where \( M \) is a set of generated mappings and \( P \) is a collection of parameters. \( P \) may include at least the following: \( S \), a (potentially singleton) set of source schemas; \( T \), the target schema; \( MD_S \), metadata about the sources; \( MD_T \), metadata about the target; and \( MD_{S \rightarrow T} \), metadata that relates \( S \) to \( T \). The mappings in \( M \)
are usually expressed in the generic form of \( s \rightarrow t \ tgds \), as this abstracts over the conceptual model underlying the database system, but, to execute them over the data, they need to be translated into an executable query language, e.g., SQL.

In relation to mapping generation for databases, probably the most influential proposal is Clio first described by Miller et al. (2000), where \( MD_S \) and \( MD_T \) include not only type information, but, crucially, also foreign key constraints; \( MD_{S \rightarrow T} \) consists of matches between elements of \( S \) and \( T \), which are called value correspondences. The described algorithm generates separate queries for each target relation, where a query is a single select-from-where-group-by clause, by using subsets of value correspondences that match the target relation. Each target attribute is matched at most once by each candidate subset of value correspondences, and the candidate set is said to be complete if all the value correspondences for every attribute in the target are included. If the value correspondences in a subset involve multiple source relations then they must satisfy the condition of taking part in (at least) one join path, which is found using foreign key constraints between the sources, otherwise, the set is discarded. The algorithm continues by selecting the minimum number of candidate sets that cover all input value correspondences (even if the same correspondence appears in several sets), thereby reducing the possibility of generating mappings that output redundant data. Then, for each candidate set, the algorithm creates the corresponding query using value correspondences that involve a (sub)set of source relations merged through join paths detected through foreign key constraints. At the end, the union all operator is applied over all the created queries.

Importantly, Clio was designed 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 there is a single schema for the source, with specified keys and foreign keys.

**Schema mappings with target constraints.** The initial Clio algorithm produced reasonable results in the sense that data from the source was translated into the format of a chosen target. However, it failed to address the issue of satisfying target constraints. Clio does not address this problem as it creates mappings regardless of the fact that not all the target attributes are covered by value correspondences. Key and foreign key constraints may be violated, e.g., if there is a lack of value correspondences on the target key attribute(s). Given
that, in general, sources can be expected to be heterogeneous, if not disjoint, i.e.,
having different origins, it is only natural that the target constraints might not
always be satisfied given the data.

In order to tackle the problem of generating mappings that address this issue,
the notion of semantic translations was introduced in Popa et al. (2002). Semantic
translation is the process of generating interpretations of the correspondences
such that they also satisfy the schema constraints, and whose results are named
logical mappings. The work described in Popa et al. (2002) is built upon the Clio
algorithm to which it adds a set of new features such that it is able to avoid data
inconsistencies. The described algorithm builds the logical mappings by using
associations between the source relations and between the target relations. For
example, considering a pair of value correspondences, \( v_1 \) and \( v_2 \), which match
two different target relations, the algorithm will first detect if the source relations
involved in the two correspondences can be associated through a (nested) refer-
tential constraint, and then it does the same check for the target relations as well.
A (nested) referential constraint can be a foreign key, for the relational model, or
provided through the nesting structure of the schema, for semi-structured data
model, i.e., XML. If after these checks it results that there is a link between the
data sources, and between the target relations, then, instead of generating separate
mappings for each target relation (as Clio did, in Miller et al. (2000)), their
mapping is built by taking into consideration both correspondences at the same
time. The associations are detected using two methods: the attributes involved
by the value correspondences are part of the same relation (structural association),
or the attributes belong to different relations, but the relations are linked through
a foreign key relationship, thus, it can be concluded that the attributes could be
semantically grouped together (logical associations). The logical associations are
considered to be maximal if there are no more attributes that can be added to
that semantic group. The logical associations are built using the chase method
[Maier et al. (1979)], where a chase step is an augmentation of an association by
applying to it a schema constraint.

Example 2.1.4. To describe better how the logical associations are built, consider
the example in Figure 1.2, and let the following be foreign key constraints between
the sources:

\[
FK_1: \forall e, a, bcy, c, pc, s, bn, pr : MR(e, a, bcy, c, pc, s, bn, pr) \rightarrow \\
\exists LC, CR, CD, CO : UKD(LC, CR, CD, CO, pc)
\]
2.1. SCHEMA MAPPINGS FOR DATABASES

FK₂: \( \forall lc, cr, cd, co, pc : UKD(lc, cr, cd, co, pc) \rightarrow \exists IR, EA, HR : UKQI(co, IR, EA, HR) \)

FK₁ expresses that for each postcode in ManchesterRealestate (dependent) there is a postcode value in UKDeprivation (referenced).

FK₂ expresses that for each county value in UKDeprivation (dependent) there is a county value in UKQualityIndices (referenced).

Chase step 0: Applying the chase method from the structural association \( A₁ \) implied by relation ManchesterRealestate:

\[ A₁ : MR(le, a, bcy, c, pc, s, bn, pr) \]

Chase step 1: Apply FK₁ to \( A₁ \) to obtain the logical association \( A₂ \):

\[ A₂ : MR(le, a, bcy, c, pc, s, bn, pr), UKD(lc, cr, cd, co, pc), MR.pc = UKD.pc \]

Chase step 2: Apply FK₂ to \( A₂ \) to obtain the logical association \( A₃ \):

\[ A₃ : MR(le, a, bcy, c, pc, s, bn, pr), UKD(lc, cr, cd, co, pc), UKQI(co, ir, ea, hr), MR.pc = UKD.pc \land UKD.co = UKQI.co \]

\( A₃ \) is the maximal logical association as there are no more constraints to apply to it. Indeed, there are no more source relations either, but if there were, the only thing that would matter would be the existence of other foreign keys that could transform the created associations.

In a mapping generation scenario, the associations are considered separately on the source and the target schemas. An association pair is said to cover a correspondence if the source and target relations in the correspondence are part of both the source and the target associations. Pairs of source and target associations are translated into mappings expressed as \( \text{tgds} \), where the target side contains the conditions imposed by the value correspondences as well.

**Example 2.1.5.** Continuing Example 2.1.4, where \( A₃ \) is the source association:

\[ A₃ : MR(le, a, bcy, c, pc, s, bn, pr), UKD(lc, cr, cd, co, pc), UKQI(co, ir, ea, hr), MR.pc = UKD.pc \land UKD.co = UKQI.co, \]

and given the target association:

\[ T₁ : UKR(ir, cr, pc, st, pr). \]

(for simplicity, we only consider a subset of the matches in Figure 1.2):

\[ v₁ : MR.pr \rightarrow UKR.pr \]

\[ v₂ : UKD.cr \rightarrow UKR.cr \]

\[ v₃ : UKQI.ir \rightarrow UKR.ir \]

Here, \( v₁, v₂, \) and \( v₃ \) are matches that state which source attributes match which target attributes: attribute Manchester.price matches UK Realestate.price,
UK Deprivation.crimerank matches the crimerank in UKR and, similarly, for income rank in UK quality of life matching the income rank in the UK Realestate target table.

Given that $A_3$ and $T_1$ associations include all source and target relations involved in $v_1$, $v_2$, and $v_3$, the algorithm detects that it needs to take into consideration all three correspondences when generating the mapping. The mapping corresponding to the three sources, i.e., MR, UKQI, and UKD, and the chosen target, i.e., UKR, based on the above associations and their correspondences, is the following tgd:

\[
\text{tgdm}_{\text{map}}: MR(\text{lc}, a, \text{bcy}, c, \text{pc}, \text{st}, \text{bn}, \text{pr}), UKD(\text{lc}, \text{cr}, \text{cd}, \text{co}, \text{pc}), UKQI(\text{co}, \text{ir}, \text{ea}, \text{hr}),
\]
\[
\text{MR.pc} = \text{UKD.pc} \land \text{UKD.co} = \text{UKQI.co} \rightarrow 
\]
\[
\text{UKR(ir, cr, pc, st, pr)}, 
\]
\[
\text{UKR.pr} = \text{MR.pr} \land \text{UKR.cr} = \text{UKD.cr} \land \text{UKR.ir} = \text{UKQI.ir}
\]

The method presented in Popa et al. (2002) was taken further by the work in Fagin et al. (2009), where given a set of value correspondences, Clio can generate nested tgds. However, the work in Fagin et al. (2009) is related to data exchange, which we discuss in Section 2.2.

Another mapping generation system is (+)Spicy which is described in several works such as Bonifati et al. (2008) (Spicy), Mecca et al. (2009) (+Spicy), and Marnette et al. (2011)(++Spicy). The work on the Spicy project shows an end-to-end data integration solution by pipe-lining integration components: first running the matching component, inputting the matches (a.k.a. value correspondences) into a process for mappings generation, and then improving the generated mappings by adding a verification step [Bonifati et al. (2008)]. (+)Spicy mapping generation builds on the Clio algorithm, but adds to it further enhancements, among which the following: (i) it rewrites s-t tgds to reduce redundancy generated by subsumed mappings; (ii) it makes use of target constraints, more specifically of key constraints that are expressed as target equality-generating dependencies; and (iii) it addresses the problem of generating mappings for a target with nested components, e.g., XML.

+Spicy [Mecca et al. (2009)] described a rewriting algorithm that takes as input a set of s-t tgds (possible mappings) and rewrites them such that, when materialized, the data redundancy is reduced, where Clio produced mappings that can generate redundant data (named canonical solutions in the context of data exchange which we discuss in Section 2.2). The work in +Spicy relied on the fact
that it can be recognized when two *tgds* may produce redundant data, e.g., possibly one subsuming the other. They studied various scenarios such as *subsumption*, where one mapping appears to produce (at least) the same information as another mapping, self-joins, and *coverages* w.r.t. the target, i.e., a coverage for the right-hand side of a *tgd* is a set of atoms from other *tgds* that might represent alternative ways of satisfying the same target constraint. According to their investigation the most common rewriting is done for *subsumption* cases. They consider that a mapping *subsumes* another if it can be detected that it can generate the same or more information than the subsumed mapping. The method they use to check if one mapping $m'$ subsumes another mapping $m$ is by checking the right-hand side of their *tgds* (the conclusion): if both conclusions are for the same target relation(s), and at least one unbound variable from the conclusion of one mapping $m$ can be mapped to a bound variable in the conclusion of the other mapping $m'$, then it is said that $m'$ subsumes $m$. When it is concluded that one mapping $m'$ subsumes another $m$, the *tgds* are rewritten such that the mappings produce less redundant tuples. For rewriting the *tgds*, they used the following intuition: when materializing the mappings, the first step would be to produce the tuples of the subsuming mapping, and then materialize from the subsumed mapping only the tuples that bring new information to the target.

**Example 2.1.6.** Given the example in Figure 1.2, assume we have the following two mappings expressed as *tgds*:

$m_1: \forall lc, cr, cd, co, pc : UKD(lc, cr, cd, co, pc) \rightarrow \exists I, S, P : UKR(I, cr, pc, S, P)$

$m_2: \forall lc, cr, cd, co, pc, le, a, bcy, c, s, bn, p :$

$UKD(lc, cr, cd, co, pc) \land MR(le, a, bcy, c, pc, s, bn, pr) \rightarrow \exists I : UKR(I, cr, pc, s, pr)$

Both mappings are compared by looking at the right-hand side of the *tgds*. It can be observed that $UKR(I, cr, pc, S, P)$ from $m_1$ can be mapped into the conclusion of $m_2$, $UKR(I, cr, pc, s, pr)$, by mapping: $I \rightarrow I$, $S \rightarrow s$, and $P \rightarrow pr$.

Under the strategy above, the mappings are rewritten into $m_1'$ and $m_2'$:

$m_1': \forall lc, cr, cd, co, pc, le, a, bcy, c, s, bn, pr :$

$UKD(lc, cr, cd, co, pc) \land MR(le, a, bcy, c, pc, s, bn, pr) \rightarrow \exists I : UKR(I, cr, pc, s, pr)$

$m_2'$ is unchanged as it is the subsuming mapping, i.e., more informative.

$m_1': \forall lc, cr, cd, co, pc, le, a, bcy, c, s, bn, pr :$

$UKD(lc, cr, cd, co, pc) \land$

$\neg(UKD(lc, cr, cd, co, pc) \land MR(le, a, bcy, c, pc, s, bn, pr)) \rightarrow$

$\exists I, S, P : UKR(I, cr, pc, S, P)$
\( m'_1 \) has changed (compared to \( m_1 \)) so that, when materialized, only the tuples that have not yet been materialized by \( m'_2 \) will be produced, thus, avoiding redundancy.

Similarly to the approach of +Spicy, and with the same aim of removing redundancy, the method described in ten Cate et al. (2009) on laconic mappings generation proposes another way of rewriting \( s-t \) tgds. The main method for transforming a set of \( s-t \) tgds into a laconic mapping comprises four steps: the first step is to build a finite list of possible tuple patterns. This is done by analyzing the conclusions (the right-hand side) of the \( tgd \). In the second step, each pattern will have an associated precondition. A precondition is a first-order formula on the sources which is able to detect if the solution will contain the pattern. In order to do this, they use the notion of certain answers, i.e., a tuple is certain for a target query \( q \) w.r.t. to a mapping \( m \) if it is returned regardless of the chosen source instance \( I \). There are methods for detecting certain answers, e.g., using universal solutions (which we discuss in Section 2.2) and using query rewriting. Their approach is based on the latter: given a query \( q_T \) over the target schema \( T \), \( q_T \) is rewritten to a query \( q_S \) over the source schema \( S \) such that \( q_S(I) \) computes the certain answers for the initial query. The approach to rewriting the queries is based on MiniCon algorithm proposed by Pottinger and Halevy (2001). The third step generates a set of side conditions expressed as a Boolean combination of formulas of the form \( x_i < x_j \). The purpose of the side conditions is to prevent the creation of multiple versions of the same pattern in the solution, i.e., it handles self-join scenarios where the same relation symbol appears more than once in the right-hand side of a \( tgd \). In the last step, the laconic mapping is built: the right-hand side is a pattern, and its left-hand side comprises the pre-condition and the side condition for its pattern.

The work on +Spicy was advanced in ++Spicy, its successor. ++Spicy was described in Marnette et al. (2011) and introduces another important feature of mapping generation: it uses target egds, as well, and similarly to +Spicy, detecting the possible redundancies between mappings before materializing them (unlike other works which first materialize and then remove redundancy, e.g., DEMo [Pichler and Savenkov (2009)]). ++Spicy builds on +Spicy where the \( tgd \), i.e., foreign keys, were chased. It adds a new feature which detects overlaps between the \( tgd \) conclusions, i.e., \( \psi(x, y) \), and then chases them. Chasing the egds has the effect of reducing nulls in two ways: the first is to replace nulls with
2.1. SCHEMA MAPPINGS FOR DATABASES

constants, where it uses overlaps to do the replacements, and the second is to detect that two nulls are one and the same, where it proposes a method for doing skolemization, and thereby, create labelled null values that may be detected as equal. An overlap is defined as follows. Consider the conclusions of two tgds with the atoms $R(t_1, t_2, \ldots, t_n)$ and $R(t'_1, t'_2, \ldots, t'_n)$, and a functional dependency $R[a_1, \ldots, a_i] \rightarrow k$, where $k$ is a key attribute in $R$ and $\{a_1, \ldots, a_i\}$ are a set of attributes of $R$. An overlap is detected if, for each $j \in \{1 \ldots i\}$, $t_{aj}$ and $t'_{aj}$ are both universal or they represent the same existential variable in $y(\psi(x, y))$. This idea is used to rewrite the tgds involving the overlap such that they do the chase on the tgds (as described in Popa et al. (2002)) and then they chase the egds. The chase on the egds is done by equating the corresponding universal and existential variables (as explained above).

Example 2.1.7. Assume that for the example in Figure 1.2, we have the following mappings expressed as s-t tgds:

$m_1 : \forall lc, cr, cd, co, pc : UKD(lc, cr, cd, co, pc) \rightarrow \exists I, S, P : UKR(I, cr, pc, S, P)$

$m_2 : \forall le, a, bcy, c, pc, s, bn, p : MR(le, a, bcy, c, pc, s, bn, p) \rightarrow \exists I, CR : UKR(I, CR, pc, s, p)$

Where the two relations, $UKD$ and $MR$ have a foreign key constraint expressed:

$FK_1 : \forall le, a, bcy, c, pc, s, bn, pr : MR(le, a, bcy, c, pc, s, bn, pr) \rightarrow \exists LC, CR, CD, CO : UKD(LC, CR, CD, CO, pc)$ (see Example 2.1.4)

In the target, $UKR$, the key constraint on $UKR, pc$ is expressed as an egd:

$egd : \forall i, cr, pc, s, p, i', cr', s', p' : UKR(i, cr, pc, s, p) \wedge UKR(i', cr', pc, s', p') \rightarrow (i = i') \wedge (cr = cr') \wedge (s = s') \wedge (p = p')$

The resulting tgd after the chase on the tgds (which is done following the method described in Popa et al. (2002) and explained in Example 2.1.4) and, then, after applying again the chase with the above egd is:

$o : MR(le, a, bcy, c, pc_1, s, bn, pr) \wedge UKD(le, cr, cd, co, pc_2) \wedge pc_1 = pc_2 \rightarrow \exists I : UKR(I, cr, pc_1, s, p)$

However, it can be observed that the overlap $o$ removes some tuples from the output as it only produces the join of the two source relations. Thus, ++Spicy adds another step which produces the tuples outside the merge by applying negation to the original mappings. Applying negation on mapping $m_1$ transforms it into:

$m'_1 : UKD(le, cr_1, cd, co, pc_1) \wedge$
\((\neg MR(le, a, bcy, c, pc_2, s, bn, pr) \land pc_1 = pc_2) \rightarrow \exists I, S, P : UKR(I, cr_1, pc_1, S, P)\)

The same applies for transforming \(m_2\) as well.

### 2.1.5 Other Work on Schema Mapping Generation

Considering the usual settings for mapping generation, i.e., \(\text{MapGen}(S, T, MD_S, MD_T, MD_{S \rightarrow T}, \rightarrow M)\), where \(M\) is a set of generated mappings; \(S\), a (potentially singleton) set of source schemas; \(T\), the target schema; \(MD_S\), metadata about the sources; \(MD_T\), metadata about the target; and \(MD_{S \rightarrow T}\), metadata that relates \(S\) to \(T\), other researchers have proposed mapping generation approaches relying on different kinds of evidence, e.g., instance-level data, user feedback, existing mapping metadata, etc. In our setting, i.e., over autonomous sources, the schema metadata may be scant (or completely missing). Thus, in our approach, we need to look for different ways to make informed decisions to merge the sources that are not obvious candidates for merging by virtue of explicitly declared join paths. In this section we outline some of the research directions that have been developed on issues related to schema mapping generation.

**Using instance-level information.** Techniques have been developed for the case where instance-level data is included as evidence, where these instances are typically provided by expert users. For example, in the work of Fletcher and Wyss (2006) they describe **Tupelo**, a mapping discovery algorithm that performs a search within the transformation space of example instances based on a set of mapping operators. The mapping operators extend the relational algebra with dynamic structural transformations, which dynamically promote data to attributes and relation names, merge, or demote metadata to data values. Essentially, these are operators which help create more complex mappings that create 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.

In Gottlob and Senellart (2010), they do not propose a mapping generation algorithm, but they describe a method for finding optimal schema mappings based on the structure and occurrences of constants in the instances. Their main contribution is the introduction of notion of a **cost function** for schema mappings. The proposed cost function has different criteria such as **validity**, **explanation**, **zero-repair**, etc. The work provides an analysis of the complexity of computing a numerical value for each type of criteria of the defined characteristics. The
computation of these characteristics is based on the source and target instance data. They measure the number of repairs that are needed in order to correctly transform the data from the source to the target format.

The work of Gottlob and Senellart (2010) has been extended by Cate et al. (2017), where they allow the use of any finite number of ground-truth data examples (instead of just one), and they, also, consider more schema languages, and for each language they consider a repair language which expresses equalities, inequalities, and ground facts (as in Gottlob and Senellart (2010)).

Another work that relies on instance data is the one described in Alexe et al. (2011a,b). This approach uses two kinds of evidence: instance-level data and user feedback for refinement. Their algorithm considers a set of mappings expressed as source-to-target tgdts (GLAV) and a set of data examples. They characterize these mappings in terms of a finite set of positive and negative data examples and whether the mapping can generate the given data examples or not. The mapping refinement process is guided by an expert user that iteratively gives pairs of source and target instance examples as input to the system, repeating the characterization step until they are satisfied with the generated mapping.

In a similar research direction, Bonifati et al. (2017) present the mapping generation process as being steered using both instance data and user feedback. The focus of their proposal is on bootstrapping a set of example tuples from the source and the target, asking for feedback from the users and then generalizing the input mapping to a new mapping that more closely meets the needs of users based on their feedback.

Mappings reuse. Further work has sought to support the refining and reuse of mappings. The notion of debugging schema mappings has been described in Chiticariu and Tan (2006). This work focuses on developing a language that helps the user understand a schema mapping. The user is able to select target/source data and the debugging algorithms find a route to it. A route is the description of the relationship between the source and target data within the schema mapping. The algorithm they propose uses tgdts and egds for expressing schema mappings and Clio for mapping generation. However, the mapping descriptions, i.e., routes, are not tied to the mapping generation algorithm.

Another research direction that has received attention is that of reusing mappings through the application of two mapping operators: composition and inversion. Mapping composition means that two mappings can be combined such
that the result of their successive execution has the same result as the execution of their composition mapping. A first proposal in this direction was made by Madhavan and Halevy (2003). The focus was on creating a composition operator which has as a result a mapping such that a query $q$ w.r.t. the composition mapping has the same certain answers as successively applying the two initial mappings that created the composition. Other proposals in this direction are described in Fagin et al. (2005) and Nash et al. (2007). Inversion was tackled in works such as Arenas et al. (2009) and its purpose is to recover initial source data once the data has been transferred from the source to the target. These operators, i.e., composition and inverse, prove to be essential in applications where metadata management is needed [Bernstein (2003)] or in schema evolution. Using the composition operator, in the area of mapping generation, MapMerge was developed and described in Alexe et al. (2012). The main contribution of the work is that once the mappings have been generated, they can be reused to create more meaningful mappings through composition. Their proposed algorithm is based on a Divide-and-Conquer strategy that breaks down the input mappings which are then correlated (used in composition) to the small mappings to create larger and more complex ones. The method they use to detect whether two mappings can be correlated or not is by using associations detected through the usage of defined foreign key constraints.

As the reuse of mappings becomes a relevant research topic itself, the need to formalize the process of reuse became a topic for investigation, and advances in this direction have been made in works such as Wisnesky et al. (2010) and, in more recent work, by Atzeni et al. (2019). In Atzeni et al. (2019), they characterize schema mapping reuse and explain how schema metadata can describe input schema mappings and use the extracted metadata knowledge to infer new mappings, called meta-mappings, i.e., an abstraction over previously defined mappings. The inference approach extends previous efforts for the definition of schema mappings by example, as the algorithm can take as input a source and a target and searches in a pool of meta-mappings for the mappings that can fit the input data (according to a defined fitness function). This work is particularly relevant when there is a need to generate mappings fast and often, so that reusing already computed meta-mappings could prove to be an asset.
2.2 Schema Mappings for Data Exchange

In data exchange, mappings are used with the aim of creating target tuples that represent the source data as accurately as possible. The problem of schema mapping generation has given rise to investigations into a data exchange setting because, given a set of generated mappings, materializing each of them could result in a significant amount of redundant or missing data (due to poorly correlated tuples).

**Data exchange setting.** Fagin et al. (2005) defines the problem of data exchange as follows. Given a source schema $S$, a target schema $T$, a set of mappings $M$ that relate $S$ to $T$, and a set of target constraints $MD_T$, the data exchange problem is the following: given a finite source instance $I$, find a finite target instance $J$ such that $(I, J)$ satisfies $M$ and $J$ satisfies $MD_T$. Such an instance $J$ is called a solution in data exchange.

**Solutions.** The set of mappings $M$ can represent several ways of transferring the data from the source to the target. However, some data transformations can be better than others in the sense that some can generate better quality tuples with less redundancy and better data correlation. In the mapping generation literature, the mapping solutions are required to have the following quality requirements [Bellahsene et al. (2011)]:

- The instance data transferred in the target, $J$, comes only from the instance source data, $I$, considering the given source to target correspondences ($MD_{S\rightarrow T}$), and the (possible) target constraints ($MD_T$). These are called universal solutions.
- Given that the universal solutions can contain mappings that generate redundant data, it is desirable to select the ones that produce solutions of the smallest size. These solutions are called core universal solutions.

The above two types of solutions are formalized using the notion of homomorphism. Essentially, a homomorphism is a mapping which expresses that one data instance can be transferred into a subset of another instance:

Given two instances $I$, and $J$, over a schema $S$, a homomorphism is defined as $h : J \rightarrow I$, $h(c) = c$, where $c \in \text{dom}(J)$ (the domain of values of $J$).

The above definition means that any given constant in $J$ can be mapped to itself in $I$, and each tuple $t = R(A_1 : v_1, ..., A_k : v_k), t \in J$ can be mapped to a tuple $t' \in I$, thus, we can say that $t'$ contains at least the same amount of
information as $t$. However, if $t$ contains more information than $t'$, i.e., there are values in $t$ that cannot be found in $t'$, then we cannot define a homomorphism between the two.

Assuming a data exchange setting where $I$ is an instance of the source schema $S$, $T$ is a target schema, $MD_{S\rightarrow T}$ are s-t tgds, and $MD_T$ a set of tgds and egds, the solutions can be defined as following [Mecca et al. (2009)]:

**Universal solution.** A solution $J$ is a universal solution for $I$ if $J$ satisfies $MD_{S\rightarrow T}$ and $MD_T$, and if for every other solution $J'$, there is a homomorphism $h: J \rightarrow J'$.

**Core universal solution.** A core solution is the smallest possible universal solution $C$, where $C$ is a subinstance of $J$ such that there is a homomorphism from $J$ to $C$, but there is no homomorphism from $J$ to a proper subinstance of $C$.

**Mapping generation for Data Exchange.** Given a set of mappings, data exchange provides techniques for evaluating these mappings in ways that minimize redundancy in the target [Fagin et al. (2005)], where the redundant data comes 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. Fagin et al. (2005); Gottlob and Nash (2008)), or involve transformations to the mappings (e.g., Mecca et al. (2012); ten Cate et al. (2009)). 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 thesis, the generation algorithm is cast in terms of algebraic operators, these can be translated to tgds for evaluation using algorithms that make use of the chase procedure. Data exchange has been investigated for different mapping languages, including those with target constraints [Fagin et al. (2005)]. In this thesis, reflecting the fact that we act over autonomous sources that can be expected to manifest inconsistencies, we do not rely on explicit schema constraints, and therefore, we do not aim to compute core mappings in the sense they have been studied so far.

Given the solutions in the context of data exchange, the evolution of schema mapping generation systems is described in Marnette et al. (2010) as being separated into several generations: Clio [Miller et al. (2000)] and early versions of Spicy [Bonifati et al. (2008)] can be seen as the paradigmatic first-generation mapping generation system. The first generation is characterized by the fact that they are capable of only generating (canonical) universal solutions, but they fail
to remove redundancy when materialized, while ++Spicy [Marnette et al. (2010)]
is referred to as being the first schema mapping generation system that is part
of the second generation given that it addresses the problem of generating core
solutions for materialization by rewriting the initial s-t tgds such that they remove
redundancy in the output and they generate labelled nulls for correlating tuples.
Laconic schema mappings would be part of the second generation as well as they
are able to generate core solutions through a different method (as explained in
Section 2.1.4).

2.3 Schema Mappings in the Wild

We call schema mapping generation in the wild the process of generating mappings
over autonomous sources that are not expected to have well-defined schemas, e.g.,
one cannot rely on the existence of explicit relationships between the sources,
and the sources can come from a plethora of domains. Thus, in the wild is the
condition where there are no declared relationships to inform the integration.

Previous research, as in Section 2.1.4, addresses the problem of generating
mappings that yield quality results in a well-defined mapping generation scenario,
i.e., where both the source and target are represented as well-behaved schemas,
and where the schema constraints are expected to be explicitly defined, i.e., in the
form of source and target metadata as tgds and egds. However, with the increase
in the availability of diverse data sets (e.g., through open data or in data lakes),
some assumptions made by prior work do not tackle the changes in the mapping
generation problem. Golshan et al. (2017) briefly describe assumptions that used
to be made in data integration scenarios that have changed in the context of
Big Data. Most of the assumptions can be transferred to the schema mapping
generation problem as well:

A1: The global schemas in data integration scenarios have reasonable sizes.

- The change in this assumption is that the global schema can be richer and
more complex in new integration scenarios where data from a large pool of
sources needs to be formatted. This challenge applies to mapping generation:
the more complex the global schema, the more complex the mapping needs
to be so as to transform as much data as possible from the sources into the
format of the target.
CHAPTER 2. RELATED WORK

- Given the problem of a complex target, it may not be feasible to make the assumption that there is just one correct mapping. Thus, there has been a focus on devising techniques for mapping refinement and mapping selection. Some papers describe techniques that rely on user feedback or/and data examples to refine the mappings (e.g., Alexe et al. (2011b); Cate et al. (2017)), others choose mappings based on a set of characteristics (e.g., Gottlob and Senellart (2010); Alexe et al. (2011a); Kimmig et al. (2017); Abel et al. (2018)). For instance, in Alexe et al. (2011b), the aim is to derive schema mappings from a set of examples such that the mappings fit the given examples, while, in Alexe et al. (2011a), the aim is to detect if a given schema mapping is characterized by a set of examples. In Gottlob and Senellart (2010), they use tuple examples without nulls, and search for mappings that can produce that target instance using a source instance I. Given that such mappings may not exist, they propose a technique for repairs on a given mapping (the repairs become an indicator of cost for the mappings). Also, in the context of web data, metadata in the form of query logs could be used to choose schema mappings that are related to frequently executed queries [Elmeleegy et al. (2011)].

A2: The data sources have well-defined schemas and the data is stored in either a structured or semi-structured format.

- A change in this assumption is that the source data is no longer expected to come in the same (structured) format, in a well-behaved schema, and not even from the same domain (which, usually, facilitated the integration process). In mapping generation, the availability of knowledge about the structure of the schemas and the existence of well-defined schema constraints has been essential as, usually, these are used to determine the merge paths between the sources (as described in Section 2.1.4). This assumption no longer holds as schema mapping generation can no longer rely on input metadata that was usually assumed to exist.

- This challenge has been addressed by recent work, such as Das Sarma et al. (2012); Zhu et al. (2016); Castro Fernandez et al. (2018), and Nargesian et al. (2018). Their focus is on tackling the problem of finding relationships between the sources in terms of possible ways they could be merged. A common approach that they have is to detect if attribute values coming
from different sources are part of the same domain. Based on these, they determine whether the sources are schema complementary, i.e., joinable, or entity complementary, i.e., unionable. Their research addresses the issue of no longer relying on relationships between the sources that could be found in well-behaved schemas, e.g., foreign keys. Their work is on a problem we share, but it focuses on inferring relationships between the sources w.r.t to a target, while our focus is on using (potentially inferred) relationships to build mappings between multiple sources w.r.t. to a target.

A3: There is a need to integrate all the sources at hand.

- The integration paradigm has for long assumed that the pool of input sources has a manageable size and scope, and thus that it is feasible to create a coherent global schema from them (see A1). This assumption no longer holds as, nowadays, the pool of sources can increase rapidly [Bernstein and Haas (2008)], and it is not feasible to assume all sources will be integrated in the same integration solution. For mapping generation, this challenge translates into being able to create mappings which involve only relevant sources w.r.t. the chosen target, otherwise, there is a risk of including data which can be considered as noise or redundant when transferred to the target format.

- Dong and Srivastava (2013) outline the challenges brought by data integration on Big Data, and one of them is the scale of the data and the constant evolution of source schemas, which creates difficulties for maintaining schema mappings, recognizing the need to investigate other approaches, producing best-effort integration. Best-effort integration over a large repository of sources can be tackled with different approaches such as creating probabilistic schema mappings which are built between the source schemas and a (possible) probabilistic mediated schema [Sarma et al. (2008); Dong et al. (2009)]. For instance, the work presented in Sarma et al. (2008) describes an approach to generating mappings for a probabilistic mediated schema that is automatically created from the input sources. They do not assume a defined target schema, and they focus on aligning source attributes with counterparts in a mediated schema, and not so much on the generation of mappings that combine data from different sources. In Dong et al. (2009), they tackle the challenge of an input of diverse sources
with (potentially) erroneous data, thus, they try to find approximate mapping solutions that take into consideration the challenges brought by such input data. Also, Mahmoud and Aboulnaga (2010a) tackle the problem of integrating numerous (web) sources by clustering single table sources, and then mapping keyword queries to the domains represented by the clusters. Another approach to create best-effort mappings could be through using a pay-as-you-go approach, where the user provides feedback on the mappings [Talukdar et al. (2008); Belhajjame et al. (2013)]. With this approach, the user annotates tuples as true positives, false negatives, and false positives according to their requirements.

A4: The data in the input data sources is mostly correct and consistent.

- This assumption no longer holds in integration scenarios as it is no longer the case that the sources have just one origin, e.g., one enterprise. Nowadays, integration scenarios involve several domains (e.g., real-estate with open-government data, or medical details with student records), thereby raising the challenge of integrating independent and inconsistent sources. In the context of mapping generation, this change is relevant because finding merge paths between (initially) independent sources becomes a challenge in itself, as mapping generation can no longer rely on explicit constraints to find possible merge paths, and as a result, these need to be inferred across the (possibly different) domain(s).

- The challenges brought by inconsistent data were addressed by research that aims to (i) find approximate mapping solutions that are between (potentially) inconsistent sources and a target [Dong et al. (2009)]; or (ii) associate and correlate source data by merging them through entity resolution and data fusion [Stonebraker et al. (2013); Fernandez et al. (2017)]. Data Tamer [Stonebraker et al. (2013)] describes an integration process where the sources are processed with machine learning algorithms to identify and group attributes into tables, transform input data and perform deduplication. Data Civilizer [Fernandez et al. (2017)] is a system that addresses a similar problem to ours, i.e., integration on less consistent data, but their approach is different because they rely less on mapping generation and more on entity resolution complemented by data fusion. There are scant details on the mapping generation component in Data Civilizer, but they deploy user feedback
on alternative join paths where these paths are associated with quality metrics. As such, our understanding is that Data Tamer and Data Civilizer do not aim to address the problem of generating mappings over many sources. Moreover, in the context of heterogeneous sources, the work of Kimmig et al. (2017) focuses on mapping selection that relies on evidence that is not always reliable and consistent. Their work proposes a selection technique that considers data examples and user feedback that are used to rule out or to promote mappings that are inconsistent or consistent, respectively, to the input examples.

Overall, the work mentioned aims to address challenges brought by data integration between autonomous, heterogeneous sources and (potentially complex) various targets. It shows that for integrating sources that come in large scale, are autonomous and constantly changing, one cannot rely on the assumptions for mapping generation that hold in a well-behaved setting, i.e., where all sources at hand are well-behaved and can be integrated in a solution for a target (as described in Section 2.1.4). Schema mapping generation over autonomous sources is an important component for integration of Big Data. The next chapters report our contributions to it.

2.4 Discussion

A significant body of work on mapping generation can trace its technical ancestry to Clio [Miller et al. (2000)], though typically retaining the assumptions that the source metadata, i.e., $MD_S$, includes declared foreign keys [Popa et al. (2002); Mecca et al. (2009)] and keys [Marnette et al. (2010)], and that mapping generation is being performed to support an expert in the construction of a high-quality integration. Further work on the refinement of mappings generated between one source schema and one target schema has been done by using different kinds of evidence, such as user feedback, instance-level information, but it was mostly done in the context of refining the mappings generated by algorithms such as Clio or (++)Spicy. Contrasting mapping generation in the wild, we can say that Clio and the works built on it 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, while *mapping generation in the wild* must contend with arbitrary numbers of source schemas and less reliable information to inform mapping generation. As a result, the focus for mapping generation research has shifted toward managing the resulting uncertainty and this challenge has been recognized in several works that have advanced the research in the direction of generating best-effort integrations.

In this thesis, we address the problem of *schema mapping generation over autonomous sources* where a mapping generation system can no longer rely on explicit schema constraints between the sources, thus, our work occupies a position between the results discussed as *mapping generation for databases* (Section 2.1.4) and *mapping generation in the wild* (Section 2.3), addressing challenges expressed by the latter while still handling those of the former. We propose an algorithm that infers relationships between heterogeneous, independent sources, where the sources are expected to contain data structured using the relational model.

The work described in this thesis 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. On the other hand, $|S|$ can be greater than 1 as we assume the data can come from various domains, i.e., different schemas; and 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 [Abedjan et al. (2015)].

In contrast with most previous work related to *mapping generation in the wild*, we combine tables in diverse ways 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 [Abedjan et al. (2015)]. The cost of computing such metadata and of exploring the space of mappings likely precludes the use of our approach at web scale. We anticipate, however, that many applications exist where data can usefully be combined from multiple sources, even where the numbers of tables involved may not run into millions.

In Section 2.3, we mentioned several techniques that tackle the challenges brought by data integration over Big Data, where *mapping generation over autonomous sources* is one of them. Methods for schema refining or mapping selection are proposed to be performed after mapping generation, as one can no longer assume that the mapping generation can yield perfect mappings without any help.
Such works can follow Dynamap as a post-processing step as the output mappings might need refinement [Cate et al. (2017)] or filtering according to a (potential) user requirements [Kimmig et al. (2017); Abel et al. (2018)]. Also, the problem caused by the lack of explicit relationships between the sources in a repository has been a research direction in several projects, such as Zhu et al. (2016); Castro Fernandez et al. (2018), and Nargesian et al. (2018). They propose methods for finding related sources w.r.t. a target based on the detected domains of the attribute values. The solutions they propose can complement our work in the sense that their methods could be integrated in a mapping generation technique with the purpose of providing information about the possible merge opportunities between the sources. Moreover, Data Tamer [Stonebraker et al. (2013)] and Data Civilizer [Fernandez et al. (2017)] have not sought to address the problem of generating mappings over many sources head-on, as in this thesis, but they rely on entity resolution and data fusion to merge the data, thus, proposing a different research direction to a shared problem.

Given that the landscape of the integration problem has shifted to open data [Miller (2018)], i.e., over heterogeneous, independent sources that do not have explicitly declared relationships between them, the works mentioned in Section 2.3 show that the integration process under such settings brings various challenges that have been the focus of recent works. Some of these challenges remain to be addressed, and *mapping generation over autonomous sources* is one of them.
Chapter 3

Mapping Generation for a Simple Target

"The secret of getting ahead is getting started. The secret of getting started is breaking your complex overwhelming tasks into small manageable tasks, and then starting on the first one."
– Mark Twain (1835 - 1910)

This chapter describes 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.

As explained in Section 2.1.4, mapping generation algorithms can take as input different types of evidence, e.g., user feedback, instance values, etc., to compute mappings in an informed manner. However, the evidence that has been taken into account so far (e.g., Clio [Fagin et al. (2009)], or ++Spicy [Marnette et al. (2010)]) with a view to generating and refining mappings are only found in well-behaved schemas. Little work has been done towards generating mappings where source schema constraints are missing. Because of this, merge opportunities can be overlooked by state-of-the-art algorithms. We describe a new technique for generating mappings that is appropriate to a setting where sources are autonomous and independent and therefore lack explicit relationships between them. Our proposal for relationship inference is based on identifying merge opportunities using profile data, more precisely candidate keys and (partial) inclusion dependencies, and schema matching tools. We then rank the mappings based on their fitness, which requires metadata, such as the estimated size of the view that would result from
mapping evaluation and the estimated number of distinct values and nulls the resulting attributes would contain.

In terms of the objectives outlined in Section 1.3, this chapter aims to meet Objective 1, viz. to devise a method for generating mappings (which we refer to as Dynamap) between source schemas and a target schema, i.e., merging the sources w.r.t. to the target schema, where the sources do not have any explicitly declared relationships (i.e., declared foreign keys that would facilitate the creation of join paths). Objective 1 is met following three research contributions: Contribution 1.1 proposes a method for merging autonomous sources based on inferred relationships using profile data, Contribution 1.2 proposes a mapping generation search algorithm based on the dynamic programming paradigm; and Contribution 1.3 describes a method for propagating profile data to intermediate mappings that result from merging other mappings in the same search space.

3.1 Preliminaries

This section describes the terminology used throughout the rest of the thesis for relation and mapping metadata.

3.1.1 Metadata, Statistics, and Profile Data

We now define the notions and notations from data profiling (i.e., candidate keys and (partial) inclusion dependencies) and mapping metadata (i.e., estimated size of the view defined by the mapping, estimated number of distinct values and nulls in the attributes of the mapping-defined view).

Database metadata and statistics. We assume a preprocessing step generates statistics for each source relation, as follows:

**Cardinality.** The cardinality (or the size) of a relation $R$ is the number of tuples in $R$, denoted by $|R|$. The cardinality of a projection $X$ of $R$, where $X$ is a (set of) attributes, is denoted by $|R.X|$.

**Number of distinct values.** The number of distinct values in the extent of an attribute $X$ in relation $R$ is denoted by $V(R.X)$ (sometimes we omit the relation name when it is clear from context and write $V(X)$ for $V(R.X)$).

**Number of nulls.** The number of nulls of an attribute $X$ in relation $R$ is denoted by $\text{nulls}(R.X)$ (again, sometimes we omit the relation name when it is clear from the context and write $\text{nulls}(X)$ for $\text{nulls}(R.X)$).
Profiling data. In the same preprocessing step as gathering database information, profiling data for each source is generated. This includes candidate keys and (partial) inclusion dependencies, such as produced by the Metanome data profiler [Papenbrock et al. (2015)], which has modules such as HyUCC [Papenbrock and Naumann (2017)] (for discovering candidate keys) and SINDY [Kruse et al. (2015)] (for discovering (partial) inclusion dependencies).

HyUCC is a candidate key discovery algorithm which uses and combines row and column-efficient methods to analyze datasets. It creates subsets of attributes from the same relational table determining which are minimal unique column combinations (minimal UCC). A UCC is a set of attributes whose projection contains no duplicate tuples, a minimal UCC means a set of attributes where elimination of any if its attributes invalidates the uniqueness of tuples for the remaining column combination. The UCCs are detected through a bottom-up generation of the subsets of attributes (starting with singleton UCCs, then with pairs of attributes and so on). The output contains all resulting minimal UCC which are detected in the bottom-up parse of the relational attributes.

SINDY is a (partial) inclusion dependencies detection algorithm that builds upon a map-reduce-style framework, which is meant to accelerate the identification of inclusion dependency candidates as this enables SINDY to run on a cluster of machines. Instead of checking all pairs of columns, SINDY performs the identification of inclusion dependencies through the full outer join of the columns in the database [Kruse et al. (2015)]:

Let \( O = A \Join B \Join \ldots \) be the full outer join of all columns in a database, where \( A, B \) etc. are columns. SINDY checks if \( A \subseteq B \) by looking at the tuples in the outer join where \( t \in O : (t[A] \neq \text{NULL} \text{ then } t[B] \neq \text{NULL}) \). Based on this verification, SINDY detects the pairs of attributes that share inclusion dependencies.

We chose HyUCC and SINDY as they are capable of outputting the type of profiling data input that we require for our mapping generation technique, i.e., candidate keys and (partial) inclusion dependencies, respectively, while also proving that they scale on large input datasets [Papenbrock and Naumann (2017); Kruse et al. (2015)], which is the context under which we build our work.

(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
3.1. PRELIMINARIES

$R$ and $S$, i.e., the ratio of distinct values from $R$ that are included in the distinct values from $S$. Note that $\theta_{R,S} \neq \theta_{S,R}$ as they represent different ratios.

The overlap $\theta_{R,S}$ of an inclusion dependency $I_{R,S} = R \subseteq_{\theta_{R,S}} S$ is defined as follows:

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 $\theta_{R,S} = 1$, and, based on profiling evidence, we say that $R \subseteq S$ 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 $\theta_{R,S} = \frac{V(R \cap S)}{V(R)}$ and $0 < \theta_{R,S} < 1$, in which case, based on profiling evidence, we say that there is a partial inclusion dependency between $R$ and $S$.

**Candidate keys.** A candidate key for a relation $R$ is an attribute (or a combination thereof) that has distinct values for every tuple in $R$ based on profiling evidence.

### 3.1.2 Schema Matching

Schema matching is a closely related research topic to mapping generation, as matching is needed to generate the source-to-target relationships ($s$-$t$ tyds) that express semantic correspondences between the source attributes and the target attributes. The close relationship between matching and mapping generation is due to the fact that matches determine which attribute values may be relevant to populating the target, while mapping generation determines how these attributes are combined to achieve that.

**Background.** Work on schema matching has been described in several surveys, e.g., Bernstein et al. (2011); Shvaiko and Euzenat (2005); Rahm and Bernstein (2001). We now briefly discuss some aspects of that work insofar as to offer a glimpse of the complexity of the overall problem.

Most matching algorithms use either schema-level matchers (i.e., based on the similarity of the schema structure as manifested in attributes and table names) or instance-level matchers (i.e., based on comparison of instances in one dataset with the instances in another) or a hybrid between them [Bellahsene et al. (2011)]. Recent work has focused on combining these techniques so as to scale to more and larger schemas. Papers that propose solutions for schema matching at scale have taken a variety of approaches. Some propose an incremental process (e.g., adding one schema at a time) [Bernstein et al. (2006)], some a holistic process
Both incremental and holistic processes often use clustering techniques on schemas or attributes or both. These aim to reduce the number of pairwise comparisons needed. The pairwise comparisons involve computing some similarity measure (e.g., edit distance, cosine distance, data type comparisons or Jaccard distance) between the arguments (e.g., schema names, and/or instance tuples). By resorting to one or more of these operations and aggregating their measurements, matchers output the similarity score between two schemas. If the score exceeds a given threshold, then conceptual equivalence, i.e., a match, is thereby postulated. When the clustering process is successful, the matching process becomes less compute-intensive and, hence, less time consuming [Mahmoud and Aboulnaga (2010b); Algergawy et al. (2011); Batini et al. (2015)]. However, reliance on clustering comes with its own challenges. The more stringent problems in this regard are (i) choosing criteria with which to decide when the derivation of new clusters should stop so that each cluster has maximum dissimilarity with the other clusters while the elements within each cluster have maximum similarity with each other, and (ii) deciding which schema belongs to which cluster. These problems have been tackled but it has been demonstrated that it is hard to achieve high accuracy when the schemas may well belong to multiple clusters [Huang et al. (2014)]. This occurs when each cluster represents a domain and the schemas comprise elements from multiple domains. Furthermore, if the clustering is not done with high accuracy, then the matching process might be affected by the fact that two similar schemas may belong to different clusters and these would never be compared to each other.

Another technique used in the incremental approach is schema fragmentation, where each schema is broken down into smaller parts, e.g., small tree structures. Such an approach is found in Aumüller et al. (2005) where the COMA++ matching tool is introduced. This tool matches each fragment from the source schema to each fragment from the target schema. The user is prompted with a set of candidate correspondences that can be modified using a GUI in order to increase accuracy. The approach decomposes a large schema matching problem into several smaller ones and reuses previous match results at the level of schema fragment. By dividing the schemata into smaller parts, COMA++ proves more suitable for matching large schemas than most competitors. However, the fact that COMA++ needs user feedback on matchings makes it unfeasible for use at scale. There were attempts at minimizing the human effort when finding the matches, but it comes
with assumptions (e.g., constraint specification for each schema) that are not always valid [Quoc Viet Nguyen et al. (2013)].

**Matches.** Previous research shows that schema matching results are not expected to be perfect, as achieving flawless results has been proven to be a difficult task due to the high degree of semantic heterogeneity, domain specificity, and different structures in the data [Bernstein et al. (2011)]. In this thesis, our focus is on schema mapping generation and, given that the correctness of matches is paramount for mapping generation, we assume that the quality of the matches is such as would result from creation by expert users who can discern correct semantic correspondences. Nevertheless, for mapping generation in the wild one cannot expect that matches will be perfect as these are (usually) provided by a matching algorithm. Thus, extending the mapping generation process by allowing the integration of partially correct matches is an open topic which we mention in Chapter 6 as an open technical challenge.

In this thesis we consider matches of the following simple form:

\[ \text{match}_{ST} : S.a \rightarrow T.a', \]  

where \( S \) is an input source relation, \( a \in \text{schema}(S) \) is an attribute in \( S \); \( T \) is a target relation and \( a' \in \text{schema}(T) \) is an attribute in \( T \).

### 3.1.3 Merge Operators

The candidate mappings generated by our mapping generation algorithm, Dynamap, are expressed in relational algebra. We describe below the relational algebra notations we use throughout the thesis. For completeness, for each operator, we add the equivalent tgd expressions as they abstract over the query language in the underlying data model. We assume that the relational operations are known as defined in Garcia-Molina et al. (2008).

Let \( R(A_0, A_1, \ldots, A_{m_R}) \) and \( R'(A'_0, A'_1, \ldots, A'_{m_{R'}}) \) be two relations with their corresponding attributes, and \( R'' \) the result of the following operators applied on them:

**Projection Operator**

Given \( P = \{A_1, \ldots, A_p\}, 1 \leq p \leq m_R \), a subset of attributes in relation \( R \) with arity \( m_R \), \( P \subseteq \text{schema}(R) \), \( R[P] \) denotes the projection of the selected \( p \) attributes of relation \( R \) in \( P \). For simplicity, we will use the notation \( R[P] \) interchangeably with the notation \( R.P \), however, the meaning should be obvious from the context.
Relational Algebra expression: \( R'' \leftarrow \pi_P(R) \)

Tuple-generating dependency expression:
\( R''(A_1, \ldots, A_p) \leftarrow R(A_0, A_1, \ldots, A_p, \ldots, A_{m_R}) \)

**Union Operator**

Given two relations \( R \) and \( R' \) with the same arity, i.e., \( m_R = m_{R'} \), where each attribute \( A_k \in \text{schema}(R) \) is schema compatible with its corresponding attribute \( A'_k \in \text{schema}(R') \), then the union of the two relations is denoted by \( R \cup R' \) and returns a new relation \( R'' \) that contains every tuple in \( R \) and every tuple in \( R' \).

Relational Algebra expression: \( R'' \leftarrow R \cup R' \)

Tuple-generating dependency expression: For simplicity, we give the same names to the schema compatible attributes in the two relations:
\( R''(A_0, \ldots, A_{m_R}) \leftarrow R(A_0, \ldots, A_{m_R}) \)
\( R''(A_0, \ldots, A_{m_R}) \leftarrow R'(A_0, \ldots, A_{m_R}) \)

**Join Operator**

Given two subsets of attributes \( X_R = \{A_0, \ldots, A_{p_R}\} \), \( 0 \leq p_R \leq m_R \), and \( X_{R'} = \{A'_0, \ldots, A'_{p'_{R'}}\} \), \( 0 \leq p'_{R'} \leq m_{R'} \), then the equi-join of the two relations is \( R \bowtie_{\theta} R' \), where \( \theta \) is a join condition expressed as a conjunction of equality conditions on pairs of attributes from the two subsets \( X_R \) and \( X_{R'} \):
\( A_0 = A'_0 \land \cdots \land A_{p_R} = A'_{p_{R'}} \).
The result of an equi-join operation is a set of aligned tuples from the two relations based on the join condition.

Relational Algebra expression: \( R'' \rightarrow R \bowtie_{\theta} R', \theta = \{A_0 = A'_0, \ldots, A_{p_R} = A'_{p_{R'}}\} \)

Tuple-generating dependency expression:
\( R''(A_0, \ldots, A_{m_R}, A'_0, \ldots, A'_{p'_{R'}}) \leftarrow R(A_0, \ldots, A_{m_R}) \land R'(A'_0, \ldots, A'_{p'_{R'}}) \land A_0 = A'_0 \land \cdots \land A_{p_R} = A'_{p_{R'}} \)

**Note** that throughout the rest of the thesis we will refer to this operation as a join, although it represents an equi-join.

**Full Outer Join Operator**

Given two subsets of attributes \( X_R = \{A_0, \ldots, A_{p_R}\} \), \( 0 \leq p_R \leq m_R \), and \( X_{R'} = \{A'_0, \ldots, A'_{p'_{R'}}\} \), \( 0 \leq p'_{R'} \leq m_{R'} \), then the full outer join of the two relations is \( R \bowtie_{\theta} R' \), where \( \theta \) is a join condition expressed as a conjunction of equality conditions on pairs of attributes from the two subsets \( X_R \) and \( X_{R'} \):
\( A_0 = A'_0 \land \cdots \land A_{p_R} = A'_{p_{R'}} \).
The full outer join operation is similar to the join operation in the sense that there is a join condition, but instead of returning only the aligned
tuples, the result of a full outer join consists of the aligned tuples together with the rest of the tuples from the two relations.

Relational Algebra expression: \( R'' \rightarrow R \bowtie \bowtie_0 R' \), \( \theta = \{ A_0 = A'_0, \ldots, A_{pR} = A'_{pR} \} \)

Tuple-generating dependency expression:

\[
R''(A_0, \ldots, A_{pR}, \ldots, A_{mR}, A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'}) \leftarrow R(A_0, \ldots, A_{pR}, \ldots, A_{mR}) \land R'(A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'}) \land A_0 = A'_0 \land \cdots \land A_{pR} = A'_{pR'}
\]

\[
\exists A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'} : R''(A_0, \ldots, A_{pR}, \ldots, A_{mR}, A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'}) \leftarrow R(A_0, \ldots, A_{pR}, \ldots, A_{mR}) \land \neg(R'(A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'}) \land A_0 = A'_0 \land \cdots \land A_{pR} = A'_{pR'})
\]

\[
\exists A_0, \ldots, A_{pR}, \ldots, A_{mR} : R''(A_0, \ldots, A_{pR}, \ldots, A_{mR}, A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'}) \leftarrow R'(A'_0, \ldots, A'_{pR'}, \ldots, A'_{mR'}) \land \neg(R(A_0, \ldots, A_{pR}, \ldots, A_{mR}) \land A_0 = A'_0 \land \cdots \land A_{pR} = A'_{pR'})
\]

### 3.1.4 Merge Operators in Mappings

While tgds are ideal for expressing the mappings when the mapping generation algorithm is oblivious to the underlying data model, our focus is on tabular data and we mainly use relational algebra throughout the thesis.

In the rest of the thesis, the operators are used to express the merges between the input sources. To express a mapping for a target table, we use the following method: \( i\) express the merge between the sources through a relational algebra operator (union, (equi-)join, or full outer join), and then \( ii\) map the result of the merge to the target table by using the input matches. This approach is chosen so that when two or more sources merge, the schema of their merge result is not transformed into the schema of the required target until the mapping generation finishes, i.e., it keeps both matched and un-matched source attributes. We refer to these mappings as intermediate mappings as their schema does not comply to the target schema until mapping generation finishes. The intermediate mappings are used to iteratively build larger mappings (explained in Section 3.4).

**Example 3.1.1.** For the example in Figure 1.2, assuming UK Deprivation (UKD) and Manchester Real-estate (MR) merge through join on postcode attributes, the
mapping that represents their merge w.r.t the target table UK Real-estate (UKR) is expressed as the following:

i) \( MR \bowtie_{\text{postcode}=\text{postcode}} UKD \rightarrow \text{map}_{MR,UKD} \)

ii) \( \pi_{\text{cr,pc,st,price}} \text{map}_{MR,UKD} \rightarrow \text{UKR} \)

It can be observed that \( \text{map}_{MR,UKD} \) has the schema of the two source relations merged through join, i.e., it will contain all 13 attributes from the two source relations, while the second step transforms its schema (through projection) to the required schema by the target, i.e., it projects only the four attributes that the two sources match together in the target.

Now, if the third source in Figure 1.2, UK Life Quality Indices (UKQ), needs to be added to the already merged sources, this can be done through the join on county attributes. These attributes are contained by UKD and UKQ. Considering that \( \text{map}_{MR,UKD} \) contains all attributes from its source relations, then the merge is possible using \( \text{map}_{MR,UKD} \) as an intermediate mapping:

i) \( \text{map}_{MR,UKD} \bowtie_{\text{county}=\text{county}} UKQ \rightarrow \text{map}_{MR,UKD,UKQ} \)

ii) \( \pi_{\text{cr,pc,st,price,ir}} \text{map}_{MR,UKD,UKQ} \rightarrow \text{UKR} \)

Had the intermediate merge of UKD and MR, \( \text{map}_{MR,UKD} \), kept only the needed attributes by the target, i.e., had it been of the form \( \pi_{\text{cr,pc,st,price}} (MR \bowtie_{\text{postcode}} UKD) \rightarrow \text{map}_{MR,UKD} \), the subsequent merge with UKQ would not have been possible as county is not needed in the UKR target, thus, it would not have been included in the projection.

Note: Throughout the rest of the thesis, when we refer to an intermediate mapping, we will often omit ii) above, referring only to the merge of the two mappings, but the matching between the merged sources and the target table should be obvious from the context. However, all created (intermediate) mappings imply both i) and ii) above, as, in the end, they all have the schema of the target with respect to which they have been built.

3.2 Dynamap - Overview of the Approach

Our proposed mapping generation algorithm, Dynamap, tackles the problem of creating mappings over a pool of sources by performing a search through the space of possible mappings using a dynamic programming approach where the inference of relationships between the sources using profile data has a steering effect on the progress of the search.
The process of mapping generation is formalized as \( \text{MapGen}(S, T, MD_S, MD_T, MD_{S \rightarrow T}) \rightarrow M \), where \( M \) is a set of generated mappings; \( S \) is a set of source schemas; \( T \) is the target schema; \( MD_S \) is metadata about the sources; \( MD_T \) is metadata about the target; \( MD_{S \rightarrow T} \) is metadata that relates \( S \) to \( T \). In our proposal, we consider:

- \( M \) is a set of mappings formalized as relational algebra expressions;
- \( S \) is a set of relational schemas, where \( 1 \leq |S| \);
- \( T \) is a relational schema with a single relation;
- \( MD_S \) comprises source metadata, viz., the database statistics and profiling data described in Section 3.1.1;
- \( MD_T \) is an empty set as, in this chapter, we do not consider target constraints (in Chapter 5 the latter are then considered too);
- \( MD_{S \rightarrow T} \) is a set of matches (a.k.a. value correspondences) which relate attributes from the sources \( S \) to target attributes in \( T \).

**Example 3.2.1.** Let us consider the scenario in Figure 3.1 as a running example through this chapter. The figure depicts four source relations with two real-estate data sources from Manchester and Cambridge, and two sources showing quality of life indices, e.g., *UK Deprivation* (*UKD*) and *UK quality-of-life-statistics* (*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 public bodies. The chosen target has a schema depicted in the single table *MA\_CA\_statistics*, which aims to draw values from all the given four source relations. As a result, we need to combine the sources so as to obtain correlated data to populate the target.

Given the signature \( \text{MapGen}(S, T, MD_S, MD_T, MD_{S \rightarrow T}) \rightarrow M \), the evidence considered in this scenario is the following:

- \( S = \{S_1, S_2, S_3, S_4\} \) is the set of four single-table relational source schemas, so \( |S| = 4 \) and \( S_1 = \{MA\}, S_2 = \{CA\}, S_3 = \{UKD\}, S_4 = \{UKQ\} \);
- \( T = \{MA\_CA\_statistics\} \) is the target;
- \( MD_S \) comprises source metadata generated by a profiling tool: 7 candidate keys annotated with \([\text{ck}]\) in Figure 3.1 (e.g., *UKQ.county*, *UKD.postcode*), and 8 (partial) inclusion dependencies, represented by arrows in Figure 3.1, the color of which is that of the dependent table and the label is the overlap (e.g., \( \theta_{CA.postcode,UKD.postcode} = 0.334, \theta_{UKD.postcode,CA.postcode} = 0.25 \)).
CHAPTER 3. MAPPING GENERATION FOR A SIMPLE TARGET

Figure 3.1: Mapping generation scenario

- \( MD_T = \emptyset \) (throughout this chapter);
- \( MD_{S\rightarrow T} \) is a set of 9 matches. For simplicity, in this example, attributes that have similar names are matches.

From this input, Dynamap generates a set of candidate mappings \( M \) that can populate \( T \) with correlated data from the four sources.

Before starting the search, i.e., prior to building the mappings, there are three (major) pre-processing steps that take place:

i) \( MD_S \), i.e., the profile data corresponding to the input sources, is either read from the input or, in case there isn’t any, is obtained by running a profiling tool such as Metanome [Papenbrock et al. (2015)];

ii) the input sources that do not have any matches to the target are discarded;

iii) mappings between each (remaining) input source and the target relation are created and referred to as base mappings.

Example 3.2.2. Assuming that for the example in Figure 3.1 no profiling data is available, the first pre-processing step is to read or generate the profile data and store it as \( MD_S \). In this example, because there are no input relations that do not match the target all input relations are considered in the mapping generation process. The third pre-processing step is to create four base mappings that map each source relation to the target. For example, the base mapping between \( MA \) (source relation) and \( MA\_CA\_statistics \) (target relation) is a simple projection
applied on the source, where the non-matched attributes will be padded with nulls, but, for simplicity, this detail is omitted from the base mapping below (expressed as an s-t tgd):

\[ MA_{-CA_{-statistics}} \leftarrow \pi_{street,postcode,price}(MA) \]

The rest of the chapter describes how the final set \( M \) of mappings is generated. In Section 3.3, we describe how each pair of (base/intermediate) mappings are merged based on the input profile data. In Section 3.4, we describe how we adapted the dynamic programming paradigm for the mapping generation problem and how the mappings are built using the merge approach in Section 3.3. The procedure to evaluate the intermediate mappings is explained in Section 3.5. In Section 3.6, we describe the method for propagating the existing evidence to newly-created mappings, i.e., how we update \( MD_s \) such that a profile-informed operator can continue to use profile data corresponding to intermediate solutions. Moreover, we show that the updated metadata can be used to evaluate the newly-created intermediate mappings without the need to materialize the latter. In particular, we can decide whether a mapping is fitter to populate the target in comparison to other mappings. In Section 3.7, we describe how all the components work together. Finally, in Section 3.8, we show how Dynamap behaves in various benchmark-generated scenarios and in a real-world one.

### 3.3 Profile-informed Merge Operator

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. The method for applying an operator between two mappings is described in Section 3.1.4: the result of a merge between two (intermediate/base) mappings has the schema of the two mappings combined through the chosen operator, preserving matched and non-matched attributes, thus preserving merge opportunities for subsequent merges.

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.
Algorithm 1 Choose suitable merge operator

1: function ChooseOperator( map1, map2)
2: \( t\_rel \) is the target relation and it’s a global variable
3: if CommonAncestors(map1, map2) > 0 then
4: \ return null
5: map1\_ma ← FindMatchesAttr(map1, t\_rel)
6: map2\_ma ← FindMatchesAttr(map2, t\_rel)
7: operator ← null
8: if SameMatches(map1\_ma, map2\_ma) then
9: \( \) operator ← Union(map1, map2)
10: else
11: if DiffMatches(map1\_ma, map2\_ma) then
12: \( \) operator ← ChooseOperatorDiff(map1, map2)
13: return operator

We call this component ChooseOperator and formalize it at a high level in Algorithm 1. Before explaining the steps in the algorithm, we define an ancestor relation w.r.t. to a given mapping as an initial source relation that is merged with other initial source relations in that mapping (given that a mapping merges at least two relations).

This method takes as input two parameters, viz., two (base) intermediate mappings \( (map1, \text{ and } map2) \), 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, CommonAncestors\(^1\) (line 3) retrieves the number of common ancestor relations between map1 and map2. If they have at least one common ancestor, then the algorithm chooses not to merge them (line 4) so as to avoid creating mappings that would yield redundant data. The intuition is that if the common ancestor is the sole link between the two, this would (possibly unnecessarily) lead to merges that would not happen otherwise. If they do not have any common ancestors, then the sets of matched target attributes are retrieved for each mapping w.r.t. the target relation by a call to FindMatchesAttr (lines 5-6). 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

\(^1\)In this thesis, if a method call is typeset in \texttt{sans-serif}, then it is a helper method. All helper methods are written and described in more detail in Appendix A.
3.3. PROFILE-INFORMED MERGE OPERATOR

Different target attributes, or (ii) they match entirely different target attributes in the same target relation, \( t_{rel} \). **SameMatches** checks whether the matches are for the same target attributes. If they are, the two mappings are unioned (lines 8-9). **DiffMatches** checks whether the matches are for different target attributes. If they are, they become candidates for joining, to be decided by **ChooseOperatorDiff** (line 12). Finally, on line 13, the output, i.e., either an operator that merges the two input mappings, or **null** if no merge was found, is returned to the calling function.

**Example 3.3.1.** In Figure 3.1, 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.

**ChooseOperatorDiff** (Algorithm 2) 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 and global state 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 3.4, 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 value 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 **full outer join** because a foreign key relationship cannot be inferred so the

---

1In this thesis, if a method call is typeset in \texttt{sans-serif}, then it is a helper method. All helper methods are written and described in more detail in Appendix A.
Algorithm 2 Generate operator when two mappings match different target attributes

1: function ChooseOperatorDiff(map1, map2)
2: \( t_{rel}, pd \) (profile data) are global variables
3: \( op \leftarrow \text{null} \)
4: \( \text{subsumedMap} \leftarrow \text{IsSubsumed}(map1, map2) \)
5: if \( \text{subsumedMap} \) not null then
6: discard(subsumedMap)
7: return \( op \)
8: \( \text{map1}_\text{keys} \leftarrow \text{FindKeys}(pd, map1) \)
9: \( \text{map2}_\text{keys} \leftarrow \text{FindKeys}(pd, map2) \)
10: \( \text{ind} \leftarrow \text{MaxInd}(pd, \text{map1}_\text{keys}, \text{map2}_\text{keys}) \)
11: if \( \text{ind} \) exists then
12: if \( \text{ind.overlap} = 1.0 \) then
13: \( op \leftarrow \text{Join}(map1, map2, \text{ind}.attributes) \)
14: else
15: \( op \leftarrow \text{OuterJoin}(map1, map2, \text{ind}.attributes) \)
16: else
17: \( \text{map1}_\text{ind} \leftarrow \text{MaxInd}(\text{map1}_\text{keys}, \text{map2}.attributes) \)
18: \( \text{map2}_\text{ind} \leftarrow \text{MaxInd}(\text{map2}_\text{keys}, \text{map1}.attributes) \)
19: \( \text{ind} \leftarrow \text{MaxCoef}(\text{map1}_\text{ind}, \text{map2}_\text{ind}) \)
20: if \( \text{ind} \) exists then
21: if \( \text{ind.overlap} = 1.0 \) then
22: \( op \leftarrow \text{Join}(map1, map2, \text{ind}.attributes) \)
23: else
24: \( op \leftarrow \text{OuterJoin}(map1, map2, \text{ind}.attributes) \)
25: else
26: \( \text{map1}_\text{mk} \leftarrow \text{FindMatchedKeys}(map1, t_{rel}) \)
27: \( \text{map2}_\text{mk} \leftarrow \text{FindMatchedKeys}(map2, t_{rel}) \)
28: if \( \text{SameMatches}(\text{map1}_\text{mk}, \text{map2}_\text{mk}) \) then
29: \( op \leftarrow \text{OuterJoin}(map1, map2, < \text{map1}_\text{mk}, \text{map2}_\text{mk} >) \)
30: return \( op \)

Algorithm 2: Generate operator when two mappings match different target attributes.

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 that are 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, \text{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, \text{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.

It can be observed that the merge opportunities that use two candidate keys in the join conditions are preferred to the ones that use one candidate key and a set of non-key attributes in the condition. The intuition behind this is that the predicted profile data values (mostly overlaps of inclusion dependencies) are more accurate when it is known that the merge was performed using two candidate keys in the join condition, as explained later in Section 3.6.

### 3.4 Mapping Generation as a Dynamic Programming Problem

**Dynamic Programming.** Dynamic programming is a method that divides a complex problem into a collection of simpler overlapping sub-problems, and then combines the sub-solutions into a solution to the original compound problem [Aho and Hopcroft (1974)]. For mapping generation, this paradigm is applied as follows:

- The *compound problem* is to find 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.
- The *sub-problem* is 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 a suitable 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 $i$. 
Motivation. The mapping generation problem must explore a large search space making it a difficult problem to solve without building it up. Thus, we decided on a methodology which allows building bottom-up solutions, i.e., from simpler to the more complex. We now discuss two other methods that could have been used on this problem, and why we prefer dynamic programming as a better option.

Greedy. With a Greedy approach, the algorithm makes the optimal choice with the expectation that the final solution is close enough to the optimal one. Applying Greedy to our problem would lead to a similar approach to the current one: the merge of intermediate mappings would have been done in iterations, but keeping only one best mapping for each iteration. However, the discarded alternatives at each iteration might have, if retained, lead to combined mappings in further iterations that are better than those that can be derived from the best solution at the point in which they were discarded. Before adapting Dynamic Programming to our problem, we tried a Greedy approach which rarely reached the ground-truth merges for which we tested it. This was due to the fact that, through this approach, many promising candidate mappings were discarded because they were not the best at intermediate points of selection. Thus, the intended solutions were seldom built. Given this experiment, we concluded that applying Greedy to our problem might mean missing on merge opportunities that the algorithm would never consider because the intermediate mappings from which those opportunities arise would not get picked to being not as good as the best at that particular point in the search. Thus, we understood that the algorithm needed to keep more than one mapping at a time, even if this meant keeping less promising solutions.

Divide-and-Conquer. The main difference between a Divide-and-Conquer approach and a Dynamic Programming one is that the former divides the problem into independent sub-problems, solve each of them, and then combines the sub-solutions, while Dynamic Programming uses overlapping sub-problems. Each sub-problem is solved with sub-solutions being stored for reuse (a technique called memoization). Dynamic Programming can be seen as an extension or refinement of Divide-and-Conquer.

Mapping generation. The top level algorithm that exhibits the characteristic recursive behaviour of dynamic programming is called \textsc{GenerateMappings}, formalized in Algorithm 3. It is first called with $i$ set to $N$, the total number of source relations. When \textsc{GenerateMappings} evaluates iteration $i$, where $i \leq N$, it tries to find candidate mappings for merging each subset of the sources
Algorithm 3 Mapping generation - the recursive method of *dynamic programming*

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function generateMappings(i)</td>
</tr>
<tr>
<td>2</td>
<td>if sub_solution[i] exists then</td>
</tr>
<tr>
<td>3</td>
<td>return sub_solution[i]</td>
</tr>
<tr>
<td>4</td>
<td>else</td>
</tr>
<tr>
<td>5</td>
<td>iteration_maps ← []</td>
</tr>
<tr>
<td>6</td>
<td>for j ← 1, ceil(i/2) do</td>
</tr>
<tr>
<td>7</td>
<td>batch1 ← generateMappings(j)</td>
</tr>
<tr>
<td>8</td>
<td>batch2 ← generateMappings(i − j)</td>
</tr>
<tr>
<td>9</td>
<td>new_maps ← mergeMappings(batch1, batch2)</td>
</tr>
<tr>
<td>10</td>
<td>iteration_maps.add(new_maps)</td>
</tr>
<tr>
<td>11</td>
<td>sub_solution[i] ← iteration_maps</td>
</tr>
<tr>
<td>12</td>
<td>return sub_solution[i]</td>
</tr>
</tbody>
</table>

with *i* relations by merging the intermediate mappings returned by previous iterations. More specifically, when running *generateMappings* in iteration *i*, the sub-solutions from iterations *j* and (*i* − *j*) will be merged (lines 6-10), where *j* grows from 1 to (the rounded up value of) half of *i*. The sub-solutions for iterations *j* and (*i* − *j*) are generated by recursive call to *generateMappings* with parameters *j* and (*i* − *j*). Two sets of intermediate mappings are generated at lines 7 and 8, members of which are merged pairwise (line 9). *Batch1* is the collection of intermediate mappings for the input *j* and *batch2* is its counterpart for the input (*i* − *j*). The next step is to merge the sub-solutions from iterations *j* and (*i* − *j*), i.e., to merge mappings from *batch1* and *batch2* (line 9), where each mapping from one batch is merged with all the mappings from the other batch. Two intermediate mappings are merged at a time because the merge operators are binary: *union*, *join* or *full outer join*. The resulting merged mappings are *memoized* as sub-solutions for iteration *i* (line 11) so that this sub-solution is available for reuse in subsequent calls to *generateMappings* (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 the target table *t*, which resides in the global state of the mapping generation process. This sub-solution represents the base solution which is generated before the first call to *generateMappings*.

After iteration *N* of Algorithm 3, a set of mappings that merge all (or the possible subsets of) the initial source relations is obtained. The schema of the output mappings is that of the target relation.

Throughout the rest of the thesis, we use the following notions:
• base mappings: a base mapping is a mapping between each source relation and the target table, essentially representing a projection on the matched attributes of the source relation in order to populate the matched target attributes. These represent the base solution for the dynamic programming search and are obtained in a pre-processing step to the search;

• intermediate mapping: an intermediate mapping is a mapping generated between iterations 1 to N (inclusively) in the mapping generation process;

• parent mapping: a parent mapping is a base/intermediate mapping that was used in a merge, thus obtaining a new intermediate mapping, where the new intermediate mapping is called its child (intermediate) mapping;

• child mapping: a child mapping is an intermediate mapping that is the result mapping from a merge between two other base/intermediate mappings (from other iterations), where the two mappings that are merged are called its parent mappings. The only mappings that cannot be called child mappings are the base mappings as these represent the starting point from which other mappings are created, all other created intermediate mappings are the child mapping of a merge.

**Algorithm 4** Merge pairwise the mappings from 2 sets of mappings

```plaintext
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 ← CHOOSE_OPERATOR(map_i, map_j)
6:         if operator not null then
7:             new_map ← NEW_MAPPING(operator, map_i, map_j)
8:             md ← COMPUTE_METADATA(new_map)
9:             if ISFITTEST(new_map) then
10:                new_maps.add(new_map)
11: return new_maps
```

**MergeMappings** (Algorithm 4) combines batches of mappings returned by **GenerateMappings**. Specifically, given pairs of mappings `map_i` and `map_j` from batches of mappings from two iterations, **MergeMappings** calls **CHOOSE_OPERATOR** (described in Section 3.3) in line 5 to identify if `map_i` and `map_j` can usefully be merged. If so, then, in line 7, **NewMapping** builds a new intermediate mapping for the chosen operator. The new mapping keeps only those attributes in the parent mapping that are either needed in the target relation, i.e., have a match to the target, and/or that could be useful in further merges. An
attribute is potentially useful in future if it is either part of a candidate key or if it is involved in an inclusion dependency. This pruning of attributes is done for optimization purposes and without causing any loss in the mapping generation process. ComputeMetadata (line 8) computes metadata for the new mapping, i.e., fitness value, metadata, and profile data. Then, 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). We say that a sub-solution at iteration $i$ is a set of memoized mappings that merge $i$ initial source relations with each sub-solution being retained for use in the future iterations.

Example 3.4.1. For the example in Figure 3.1, where $N ← 4$, in the last iteration $i ← 4$, 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 instance, assume that in iteration 2, the following mappings (among others not shown here) were found, where $merge$ abstracts over the specific operation used to combine its operands:

\[
\begin{align*}
m_{2,1} &← merge(MA,CA) \\
m_{2,2} &← merge(UKQ,UKD) \\
m_{2,3} &← merge(CA,UKD)
\end{align*}
\]

Then, in iteration 4, GenerateMappings tries to merge each of the 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.

3.5 Mapping Fitness

In this section, we discuss possible characteristics and directions for designing a fitness function that is to be used in assessing a mapping without needing to materialize its instances.

Motivation. In our mapping generation setting, the fitness function is needed to compare candidate mappings that stem from the same source relations so as to discard the less promising mappings that merge the same sources, the goal being that we keep only one generated combination for each subset of initial source relations. A solution for evaluating the mappings without materializing them is needed in order to have a feasible approach in terms of processing time for generating mappings. If the algorithm were to materialize the mappings, then the mapping generation process would be slowed down by the need to evaluate
the queries then possibly write it to secondary storage, in a context where the search strategy generates many candidate mappings, in each iteration.

**Mapping characteristics.** Extracting mapping characteristics is a subject of on-going research. It is has been studied in, e.g., Gottlob and Nash (2008), where a formal framework is defined in which schema mappings can be expressed and which allows the evaluation of their quality. The cost function they design is based on the number of repairs the mapping needs so as to represent correctly the target instance using instance data generated by the execution of the mappings. Alexe et al. (2011a) present another approach. They focus on assigning a rank to the mappings based on data examples that *uniquely characterize the mapping*. This approach is also based on evaluating materialized data examples.

Previous works focus on example-based fitness functions because they assume that the mapping generation process has ended and the final set of candidate mappings can be evaluated using an instance-driven cost function. However, in our setting, this assumption is not valid as we need to evaluate the mappings immediately after they are generated in order to decide whether or not to keep them. The available evidence for computing mapping fitness consists of the metadata, profiling data and the following estimates w.r.t. the results of the mapping:

- number of nulls and distinct values on attributes,
- size of the extents,
- coverage of the target, i.e., the number of matched target attributes,
- candidate keys and (partial) inclusion dependencies, and
- merge opportunities.

The main requirement for a fitness function, in our setting, is to assign a fitness to each mapping, where the fitness indicates whether, upon evaluation, the mapping would output instances that, to a lesser or a greater extent in comparison to other mappings, correlate source data in such a way as to produce valuable target data. We define *valuable* w.r.t a set of desired data properties, e.g., as few nulls as possible, as many complete tuples as possible, etc.

**Fitness function.** Different fitness functions could be conceived. In this thesis, the fitness of a mapping is a function of the estimated number of largely complete tuples it would return, if evaluated. This approach was found to be effective in practice. The fitness function builds on an approach to infer profiling data (described in Section 3.6) for the result of a merge operator by propagating the
3.5. MAPPING FITNESS

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

Fitness is formalized in Algorithm 5. It is called after a new intermediate mapping is created (see Algorithm 4). Specifically, given the list of attributes of a mapping, 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 notion of fitness prefers mappings with larger results (thus retaining more data from the sources) and with fewer nulls.

The fitness function can be changed so as 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 target, which aims to populate as many attributes as possible.

Example 3.5.1. In our scenario, there are two options for merging Cambridge agency (CA), UK quality of life statistics (UKQ), and UK Deprivation (UKD):

1. $m_{3,1} \leftarrow (CA\bowtie_{CA.county=UKQ.county}UKQ)\bowtie_{CAUKQ.postcode=UKD.postcode}UKD$
2. $m_{3,2} \leftarrow (CA\bowtie_{CA.postcode=UKD.postcode}UKD)\bowtie_{CAUKD.county=UKQ.county}UKQ$

When both these mappings are generated (one was memoized and one is newly created), the algorithm detects that these mappings stem from the same three initial sources and chooses which one to keep. Both contain join operations, but only $m_{3,1}$ keeps all the tuples. One can observe that both mappings correlate the same data, while $m_{3,2}$ is actually subsumed by the output of $m_{3,1}$ as it loses the tuples that do not satisfy the join condition in the second merge. The algorithm will choose to keep mapping $m_{3,1}$ for further merges as $m_{3,1}$ has a better fitness.
as it will estimate that it will produce four complete tuples while for $m_{3,2}$ it will estimate just two.

## 3.6 Profile Data Propagation

This section describes how profiling data is propagated from the operands to the result of the merge operator without materializing data.

**Motivation.** In searching the space of candidate mappings, Dynamap requires metadata about the latter so as to consider how they relate to each other (e.g., can they be joined or unioned), and also as inputs to comparing their fitness (as seen in Section 3.5). Dynamap assumes the availability of profiling data, in the form of cardinalities (to compute mapping fitness – Section 3.5), and of keys and inclusion dependencies (to inform how mappings can be combined – Section 3.3). Such profiling data can be obtained for source data sets using a profiling tool such as Metanome [Papenbrock et al. (2015)]. However, given that we do not materialize intermediate results, we must propagate profile data from the operands to the result of merge operators. 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, i.e., taking into account the relationships between the new and the memoized mappings.

### 3.6.1 Generalized Profile Data Propagation

**Mapping cardinalities.** Mapping generation makes use of cardinality estimates for mappings. In particular, for a mapping $m$ and an attribute $C$, the result size in number of tuples ($|m|$), the numbers of distinct values for each attribute ($V(C)$), and the number of nulls in each attribute ($nulls(C)$). The result size and the number of nulls are used by the fitness function, whereas the number of distinct values is needed to derive properties of inclusion dependencies associated with the mapping. The result sizes and the numbers of distinct values returned by relational operators can be estimated using established techniques (e.g., Garcia-Molina et al. (1999)). These characteristics are computed from the profile data of the parent mappings and the operator used to combine the parent mappings, as detailed in Table 3.1. In Table 3.1, $r$ represents an intermediate mapping that results from the merge between two other mappings, where the merge operation can be either union, join or full outer join. As explained in Section 3.1.4, here we
### 3.6. PROFILE DATA PROPAGATION

<table>
<thead>
<tr>
<th>Merge operator</th>
<th>Result size -</th>
<th>Estimates for mappings</th>
<th>Nulls per attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r ← \text{merged}(m_1, m_2)$</td>
<td>$</td>
<td>r</td>
<td>←</td>
</tr>
<tr>
<td><strong>Union</strong></td>
<td>$r \leftarrow m_1 \bowtie X_1 \bowtie m_2 \bowtie X_2$, where $X_1 \subseteq X_2$, $Y_1 \in \text{schema}(m_1)$, $Y_1 \neq X_1$, $Z_2 \in \text{schema}(m_2)$, $Z_2 \neq X_2$</td>
<td>$V'(r.Y) ← V(m_1.Y_1)$</td>
<td>$\text{nulls}(r.Y) ← \text{nulls}(m_1.Y_1)$</td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td>$r \leftarrow m_1 \bowtie X_1 \bowtie m_2 \bowtie X_2$, where $X_1 \subseteq X_2$, $0 &lt; \theta &lt; 1$, $Y_1 \in \text{schema}(m_1)$, $Y_1 \neq X_1$, $Z_2 \in \text{schema}(m_2)$, $Z_2 \neq X_2$</td>
<td>$V'(r.Z) ← \begin{cases}</td>
<td>r</td>
</tr>
<tr>
<td><strong>Full Outer Join</strong></td>
<td>$r \leftarrow m_1 \bowtie X_1 \bowtie m_2$, where $X_1 \subseteq X_2$, $0 &lt; \theta &lt; 1$, $Y_1 \in \text{schema}(m_1)$, $Y_1 \neq X_1$, $Z_2 \in \text{schema}(m_2)$, $Z_2 \neq X_2$</td>
<td>$V'(r.Y) ← V(m_1.Y_1)$</td>
<td>$\text{nulls}(r.Y) ← \text{nulls}(m_1.Y_1)$</td>
</tr>
<tr>
<td>&amp; $V'(r.Z) ← V(m_2.Z_2)$</td>
<td>$\text{nulls}(r.Z) ← \text{nulls}(m_2.Z_2) \ast</td>
<td>r</td>
<td>$</td>
</tr>
</tbody>
</table>

Table 3.1: Cardinality estimations

In Table 3.1 the step where the resulting mapping $r$ is brought to the schema of the target table $t$, as $r$ represents an intermediate mapping that is meant to be merged in subsequent iterations (if any), thus, preserving both matched and unmatched attributes. Table 3.1 uses the notation introduced below.

Let $r[X \cup Y \cup Z] ← \text{merge}(m_1[X_1 \cup Y_1], m_2[X_2 \cup Z_2])$, where $X, Y, Z, X_1, Y_1, X_2$, and $Z_2$ are (sets of) attributes from mappings $r$, $m_1$ and $m_2$. The $X, Y, Z, X_1, Y_1, X_2$, and $Z_2$ attribute sets are defined as follows:

- if $r$ is the result of union then $X_1$ and $X_2$ represent the projections of $m_1$ and $m_2$ on which union is performed, while $Y_1$ and $Z_2$ are empty sets ($m_1.X_1$ and $m_2.X_2$ are schema compatible), thus $Y$ and $Z$ are empty as well.

- if $r$ is the result of join or full outer join, then $X_1$ and $X_2$ represent the join condition attributes on which $m_1$ and $m_2$ join, and their merge is represented by attribute $X$ in $r$, while $Y_1$ and $Z_2$ are attributes that are not used in the join condition, but are used to build attributes $Y$ and $Z$ in the new intermediate mapping.

**Profiling data.** Estimating properties of relationships between mappings in the wild is the subject of active investigation. For example, recent results have described probabilistic approaches to estimating the unionability [Nargesian et al. (2018)] and joinability [Zhu et al. (2016)] of attributes in large data sets, indexed using Locality Sensitive Hashing. Such solutions approximate measures of degree in relationship, e.g., degree of overlap, or containment, between attributes using special hash functions applied on their extents. In our setting, given that we do not materialize the extents of intermediate mappings, we cannot use such hash-based approximation techniques and, therefore, our focus is on propagating...
profiling data originally stemming from source tables.

This section describes how information about candidate keys and (partial)
inclusion dependencies is propagated through the algebraic operators used in
mappings, and thus how profile data can be propagated to new candidate mappings
they give rise to.

Propagating (partial) inclusion dependencies means inferring inclusion depen-
dencies for a child mapping from the inclusion dependencies in which the parent
mappings participate. The inference is done by replacing the parent attributes
with the new corresponding attributes (from the newly generated intermediate
mapping) and computing a new overlap. The computation of the new overlap
depends on several factors:

- which type of operator is applied to the parent mappings, i.e., union, full
  outer join (lossless merge) or join (lossy merge);
- whether the attribute(s) in the propagated inclusion dependency appear in
  the join condition (when the operator is join or full outer join);
- whether the parent attributes represent the dependent or the referenced
  attributes in the propagated inclusion dependency; and
- whether the parent attributes (referenced or dependent) are involved in
  other inclusion dependencies, i.e., there are overlaps between the same
  parent attributes and attributes in other intermediate mappings.

We now define the following notations:

- Let \( m_1 \) and \( m_2 \) be mappings that merge to give mapping \( r \); \( m_1 \) and \( m_2 \) are
  said to be the parent mappings of \( r \).
- Let \( r[R \cup X \cup Y] \leftarrow \text{merge}(m_1[S \cup X_1], m_2[P \cup Y_2]) \), where \( R, X, Y, S, X_1, P, \)
  and \( Y_2 \) are (sets of) attributes from mappings \( r, m_1 \) and \( m_2 \).
- Let \( R, X \) and \( Y \) be (sets of) attribute(s), \( R, X, Y \subseteq \text{schema}(r) \), where \( R \)
  denotes the merge of attributes \( S \) and \( P \) according to the chosen op-
erator, \( S \subseteq \text{schema}(m_1), P \subseteq \text{schema}(m_2) \), \( X \) denotes the data trans-
ferred from \( X_1 \subseteq \text{schema}(m_1) \), and \( Y \) denotes the data transferred from
\( Y_2 \subseteq \text{schema}(m_2) \). Attributes \( X_1 \) and \( Y_2 \) are not combined with other
attributes in the merge between \( m_1 \) and \( m_2 \).
- Let \( Q \) be a (set of) attribute(s), where \( Q \subseteq \text{schema}(m_3) \), where \( m_3 \neq m_1, \)
  and \( m_3 \neq m_2 \).

Inclusion dependency inference. Given an inclusion dependency \( S \subset_{\theta_S,Q} Q \), the
inferred inclusion dependency is of the form \( R \subset_{\theta_R,Q} Q \), given that attribute \( S \) is
3.6. PROFILE DATA PROPAGATION

a parent attribute of R. Given an inclusion dependency \( Q \subseteq_{\theta_{Q,S}} S \), the inferred inclusion dependency is of the form \( Q \subseteq_{\theta_{Q,R}} R \), given that S attribute is a parent attribute of R.

Tables 3.2 and 3.3 show the set of formulas that our work contributes for inferring overlaps (\( \theta \)) without materializing any of the mappings. In both tables, when both attributes from both parents are involved in the inclusion dependency \((S, P, X_1 \text{ or } X_2)\), and when one of the parents is involved in an inclusion dependency with another attribute \( Q \in \text{schema}(m_3) \), \( m_3 \neq m_1 \), \( m_3 \neq m_2 \). For simplicity, Table 3.2 only shows when parent attributes from \( m_1 \) are involved in an inclusion dependency with \( Q \), but the same formulas apply for the attributes of \( m_2 \). The tables differ based on whether the merge of two parent attributes, \( S \) and \( P \), causes the loss of distinct values in the new attribute \( R \). Note that the tables show the interactions between singleton sets of attributes (i.e., \( S, P \) and \( Q \) contain just one attribute) as the current state-of-the-art in discovering partial inclusion dependencies (e.g., Kruse et al. (2015)) does not yet support multi-attribute sets due to the high computational cost.

**Lossless merge.** Table 3.2 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 \( r.R \leftarrow m_1.S \cup m_2.P \), with \( S \) and \( P \) attributes that were projected from the parent mappings to compute the union, and where \( R \subseteq \text{schema}(r) \) is the result of \( S \) merging \( P \), where \( \theta_{S,P}, \theta_{P,S} \in [0, 1] \), and \( X_1 \) and \( Y_2 \) are sets of attributes with any number elements (0 ≤ arity(\( m_1.X_1 \))), 0 ≤ arity(\( m_2.Y_2 \)).

**Lossy merge.** Table 3.3 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 \( R \subseteq \text{schema}(r) \) is the result of \( S \) merging \( P \), where \( \theta_{S,P} = 1 \) and \( \theta_{P,S} \in (0, 1] \).

**Example 3.6.1.** In iteration 2, MA and CA are merged through union. The partial inclusion dependencies between these two relations and UKD and UKQ need to be propagated to the newly created intermediate mapping: \( m_{2,1} \leftarrow MA \cup CA \). For union operations, the merge is lossless, so Table 3.2 is used to compute the new overlaps.
In Table 3.4, the step where the resulting mapping $r$ is brought to the schema of the target table $t$, as $r$ represents an intermediate mapping that is meant to be merged in subsequent iterations (if any), thus, preserving both matched and unmatched attributes.

### Table 3.2: Inclusion dependencies propagation for lossless attribute merges

For the propagation of $MA.postcode \subset_0.667 UKD.postcode$ into $m_{2.1}.postcode \subset_{\theta} UKD.postcode$ the overlap estimation ($\theta$) needs to be computed. In Table 3.2, the dependent attribute of the parent mapping is $S$, i.e., $MA.postcode$, and the referenced attribute is not a parent, i.e., $Q$ is $UKD.postcode$, while the other parent attribute $P$ is $CA.postcode$. The two parent attributes ($MA.postcode$ and $CA.postcode$) are disjoint, thus the condition on the second row is satisfied ($\theta_{S,P} = 0$), so $\theta = \frac{3\times0.334+3\times0.667}{6} = 0.5$. A similar process applies for the remaining inclusion dependencies.

**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. In Table 3.4, $r$ represents the intermediate mapping that results from the merge between two other mappings. As explained in Section 3.1.4, we omit in Table 3.4 the step where the resulting mapping $r$ is brought to the schema of the target table $t$, as $r$ represents an intermediate mapping that is meant to be merged in subsequent iterations (if any), thus, preserving both matched and unmatched attributes.
### 3.6. PROFILE DATA PROPAGATION

#### 3.6.2 Profiling Data Propagation Examples

In this section, we give examples of the propagation cases in Tables 3.2 and 3.3 so as to explain how a newly inferred inclusion dependency is created and to show that most of the formulas can accurately predict the new overlap.

---

**Example 3.6.2.** After propagating the inclusion dependencies for mapping $m_{2,1}$, the algorithm tries to propagate the candidate keys that the parent relations have. In this case, it checks whether either of the parents have candidate keys, which, in this case, neither MA nor CA do.

However, if it were the case that the `postcode` attribute in both relations was a candidate key, and given that i) $\theta_{MA.postcode,CA.postcode} = 0$, i.e., there is no overlap between $MA.postcode$ and $CA.postcode$, and ii) $\text{nulls}(MA.postcode) = \text{nulls}(CA.postcode) = 0$, i.e., they do not have any nulls, then the algorithm would have concluded that $r.postcode$, where $r.postcode \leftarrow MA.postcode \cup CA.postcode$, remains a candidate key for the resulting mapping $r$.

---

**Table 3.3:** Inclusion dependencies propagation for *lossy* attribute merges

<table>
<thead>
<tr>
<th>Dependent</th>
<th>Referenced</th>
<th>Conditions</th>
<th>Overlap(s) for inferred inclusion dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(parent)$</td>
<td>$P(parent)$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{R,P} = 1$, $\theta_{S,R} = 1$, $\theta_{R,S} = 1$, $\theta_{R,P} = \frac{1}{\sqrt{</td>
</tr>
<tr>
<td>$P(parent)$</td>
<td>$S(parent)$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{R,P} = 1$, $\theta_{S,R} = \theta_{S,Q}$</td>
</tr>
<tr>
<td>$Q$ or $P(parent)$</td>
<td>$S$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{R,Q} = \theta_{S,X}$</td>
</tr>
<tr>
<td>$S$ or $P(parent)$</td>
<td>$Q$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{R,Q} = \theta_{S,Q}$</td>
</tr>
<tr>
<td>$Q$ or $P(parent)$</td>
<td>$X_1$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{R,Q} = \theta_{S,Q}$</td>
</tr>
<tr>
<td>$Q$</td>
<td>$X_1(parent)$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{R,Q} = \theta_{X_1,Q}$</td>
</tr>
<tr>
<td>$Y_2(parent)$</td>
<td>$Q$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{Y,Q} = \begin{cases} 1, &amp; \text{if } \theta_{Q,Y} &gt;</td>
</tr>
<tr>
<td>$Q$</td>
<td>$Y_2(parent)$</td>
<td>$\theta_{S,P} = 1$</td>
<td>$\theta_{Q,Y} = \begin{cases}</td>
</tr>
</tbody>
</table>

**Table 3.4:** Candidate keys propagation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Conditions for propagating a candidate key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>$r.X \leftarrow m_1.X_1 \cup m_2.X_2$</td>
</tr>
<tr>
<td>Join</td>
<td>$r \leftarrow m_1 \bowtie X_1 = X_2$</td>
</tr>
<tr>
<td></td>
<td>$m_2.X_2$, where $X_1 \subseteq X_2$</td>
</tr>
<tr>
<td></td>
<td>$Y_1 \in \text{schema}(m_1)$, $Y_1 \neq X_1$</td>
</tr>
<tr>
<td></td>
<td>$Z_2 \in \text{schema}(m_2)$, $Z_2 \neq X_2$</td>
</tr>
<tr>
<td>Full Outer Join</td>
<td>$r \leftarrow m_1 \bowtie X_1 = X_2$</td>
</tr>
<tr>
<td></td>
<td>$m_2.X_2$</td>
</tr>
</tbody>
</table>
3.6.2.1 Example Settings

For these examples, we use the relational schemas in Figure 3.1. However, the attribute values are changed so that we can adapt the overlaps as required for triggering the propagation cases. More specifically, we look at the following abstract case, using the notations from Section 3.6.1:

Let \( r[R \cup X \cup Y] \leftarrow \text{merge}(UKD[S \cup X_1], CA[P \cup Y_2]) \), where \( S, X_1, P, Y_2, R, X, \) and \( Y \) are single attributes from the base mappings of \( UKD, CA \), and their merge result, \( r \), and let \( Q \) be an attribute from the base mapping of \( UKQ \) where:

- \( S = UKD.postcode \),
- \( P = CA.postcode \),
- \( R = \text{merge}(S, P) \), i.e., \( R \) represents the merge of attributes \( S \) and \( P \) according to the chosen merge operator,
- \( X_1 = UKD.county \),
- \( X = r.UKD.county \), i.e., it represents the data transferred from \( X_1 \),
- \( Y_2 = CA.county \),
- \( Y = r.CA.county \), i.e., it represents the data transferred from \( Y_2 \),
- and \( Q = UKQ.county \).

Let us consider the following inclusion dependencies:

1. \( UKD.postcode \subset \theta_{UKD.postcode,UKQ.postcode} UKQ.postcode (S \subset \theta_{S,Q} Q) \),
2. \( UKQ.postcode \subset \theta_{UKQ.postcode,UKD.postcode} UKD.postcode (Q \subset \theta_{Q,S} S) \),
3. \( UKD.county \subset \theta_{UKD.county,UKQ.county} UKQ.county (X_1 \subset \theta_{X_1,Q} Q) \),
4. \( UKQ.county \subset \theta_{UKQ.county,UKD.county} UKD.county (Q \subset \theta_{Q,X_1} X_1) \),

then, given that attribute \( UKD.postcode \) is a parent attribute of \( R \), and \( UKD.county \) is a parent attribute of \( X \), their corresponding inferred inclusion dependencies are:

1. \( R \subset \theta_{R,UKQ.postcode} UKQ.postcode (R \subset \theta_{R,Q} Q) \),
2. \( UKQ.postcode \subset \theta_{UKQ.postcode,R} R (Q \subset \theta_{Q,R} R) \),
3. \( X \subset \theta_{X,UKQ.county} UKQ.county (X \subset \theta_{X,Q} Q) \), and
4. \( UKQ.county \subset \theta_{UKQ.county,X} X (Q \subset \theta_{Q,X} X) \)

It can be observed that the first two inclusion dependencies in each batch correspond to attributes that are used in the merge condition, while the latter two correspond to attributes that are not merged with other attributes during the merge.

Note that each pair of attributes has two corresponding inclusion dependencies each with different overlaps (e.g., for each two overlapping attributes \( S \) and \( Q \)
there are $\theta_{S,Q} \neq \theta_{Q,S}$). This is reflected in Tables 3.2 and 3.3, as each case has a mirroring case for computing the other overlap, e.g., in Table 3.2, Case 2 is mirrored by Case 11. Below, we consider such cases together.

![Figure 3.2: Lossless merges - attributes overlaps](image-url)
3.6.2.2 Lossless Merges

Table 3.2 is used in the case of a lossless merge, i.e., the merge operator is union or full outer join, as none of the attribute values are lost. In the case of both operators, $R \leftarrow S \cup P$ as all the distinct values from $S$ and $P$ will be contained by $R$.

**Distinct values.** For computing the overlaps, we need to compute $V(R)$, the number of distinct values in $R$. For this, we use the formulas in Table 3.1 for the corresponding merge operator. For both union and full outer join, $V(R) = V(UKD.postcode) + V(CA.postcode) − V(UKD.postcode \cap CA.postcode)$. In the case of full outer join, the attributes outside the join condition retain the number of distinct values their parent had.

Let us consider the Venn diagrams in Figure 3.2 to represent the interaction between the two sets of parent (postcode) attributes and the third attribute with which the child infers the relationships. For simplicity, the name of the attribute (postcode) is omitted from the diagrams.

**Case 1.** The first case in Table 3.2 represents the case when the inclusion dependency where both parent relations are involved is propagated to the child. This propagation would create a link between each parent and their child mapping. Here, two inclusion dependencies are inferred (one for each parent), and there are no preconditions for propagating them. An example is depicted in Figure 3.2(a). Distinct Values: $V(CA.postcode) = 5$, $V(UKD.postcode) = 6$, $V(R) = 6 + 5 - 2 = 9$

*Propagated ind:* $UKD.postcode \subseteq 0.334 CA.postcode $

Inferred inds:

1. $R \subseteq 0.556 CA.postcode$ as $
   \theta_{R,CA.postcode} = \frac{V(CA.postcode)}{V(R)} = \frac{5}{9}$
2. $UKD.postcode \subseteq 1.0 R$

**Cases 2 & 11.** This example is depicted in Figure 3.2(b) where the postcode attributes from $UKD$ and $CA$ are disjoint. However, both have common values with $UKQ$. Distinct Values: $V(CA.postcode) = 5$, $V(UKD.postcode) = 4$,

$V(R) = 5 + 4 - 0 = 9$, $V(UKQ.postcode) = 6$

**Propagated inds:**

1. $UKD.postcode \subseteq 0.25 UKQ.postcode$
2. $UKQ.postcode \subseteq 0.1667 UKD.postcode$
Inferred inds:
1. \( R \subset_{\theta_{R,UKQ.postcode}} UKQ.postcode, \theta_{R,UKQ.postcode} = \frac{5+0.4+4+0.25}{9} = \frac{3}{5} = 0.334 \)
2. \( UKD.postcode \subset_{\theta_{UKQ.postcode,R}} R, \theta_{UKQ.postcode,R} = \frac{5+0.4+4+0.25}{6} = \frac{3}{6} = 0.5 \)

Cases 3 & 12. This example is depicted in Figure 3.2(c) where the postcode values from \( UKD \) are fully included in the values of \( CA.postcode \).

Distinct Values: \( V(CA.postcode) = 5, V(UKD.postcode) = 2, V(R) = 5 + 2 - 2 = 5, V(UKQ.postcode) = 6 \)

Propagated inds:
1. \( UKD.postcode \subset_{1.0} UKQ.postcode \)
2. \( UKQ.postcode \subset_{0.334} UKD.postcode \)

Inferred inds:
1. \( R \subset_{\theta_{R,UKQ.postcode}} UKQ.postcode, \theta_{R,UKQ.postcode} = \frac{5+0.6}{5} = \frac{3}{5} = 0.6 \)
2. \( UKD.postcode \subset_{\theta_{UKQ.postcode,R}} R, \theta_{UKQ.postcode,R} = \frac{5+0.6}{6} = \frac{3}{6} = 0.5 \)

Cases 4 & 13. This example is depicted in Figure 3.2(d) where the postcode values from \( CA \) are fully included in the values of \( UKD.postcode \).

Distinct Values: \( V(CA.postcode) = 5, V(UKD.postcode) = 9, V(R) = 5 + 9 - 5 = 9, V(UKQ.postcode) = 7 \)

Propagated inds:
1. \( UKD.postcode \subset_{0.44(4)} UKQ.postcode \)
2. \( UKQ.postcode \subset_{0.571} UKD.postcode \)

Inferred inds:
1. \( R \subset_{\theta_{R,UKQ.postcode}} UKQ.postcode, \theta_{R,UKQ.postcode} = \frac{9+0.44(4)}{9} = \frac{4}{9} = 0.444 \)
2. \( UKD.postcode \subset_{\theta_{UKQ.postcode,R}} R, \theta_{UKQ.postcode,R} = \frac{9+0.44(4)}{7} = \frac{4}{7} = 0.571 \)

Cases 5 & 14. This example is depicted in Figure 3.2(e) where the postcode values from \( UKD \) are fully included in the values of \( UKQ.postcode \).

Distinct Values: \( V(CA.postcode) = 5, V(UKD.postcode) = 6, V(R) = 5 + 6 - 2 = 9, V(UKQ.postcode) = 9 \)

Propagated inds:
1. \( UKD.postcode \subset_{1.0} UKQ.postcode \)
2. \( UKQ.postcode \subset_{0.667} UKD.postcode \)

Inferred inds:
1. \( R \subset_{\theta_{R,UKQ.postcode}} UKQ.postcode, \theta_{R,UKQ.postcode} = \frac{6-6+0.334+5+0.4}{9} = \frac{6}{9} = 0.667 \)
2. \( UKD.postcode \subset_{\theta_{UKQ.postcode,R}} R, \theta_{UKQ.postcode,R} = \frac{6-6+0.334+5+0.4}{9} = \frac{6}{9} = 0.667 \)
**Cases 6 & 15.** This example is depicted in Figure 3.2(f) where the postcode values from CA are fully included in the values of UKQ.postcode.

Distinct Values: $V(\text{CA.postcode}) = 5$, $V(\text{UKD.postcode}) = 5$,

$V(R) = 5 + 5 - 1 = 9$, $V(\text{UKQ.postcode}) = 8$

Propagated inds:
1. $\text{UKD.postcode} \subset_{0.4} \text{UKQ.postcode}$
2. $\text{UKQ.postcode} \subset_{0.25} \text{UKD.postcode}$

Inferred inds:
1. $R \subset_{\theta_{R,UKQ.postcode}} \text{UKQ.postcode}$, $\theta_{R,UKQ.postcode} = \frac{5 - 5 + 0.2 + 5 + 0.4}{9} = \frac{6}{9} = 0.667$
2. $\text{UKD.postcode} \subset_{\theta_{UKQ.postcode,R}} R$, $\theta_{UKQ.postcode,R} = \frac{5 - 5 + 0.2 + 5 + 0.4}{8} = \frac{6}{8} = 0.75$

**Cases 7 & 16.** This example is depicted in Figure 3.2(g) where the postcode values from UKQ are fully included either in the values of CA.postcode or in UKD.postcode. We describe below the case where they are included in CA.postcode, but the same applies for the other case.

Distinct Values: $V(\text{CA.postcode}) = 8$, $V(\text{UKD.postcode}) = 5$,

$V(R) = 8 + 5 - 1 = 12$, $V(\text{UKQ.postcode}) = 3$

Propagated inds:
1. $\text{UKD.postcode} \subset_{0.2} \text{UKQ.postcode}$
2. $\text{UKQ.postcode} \subset_{0.334} \text{UKD.postcode}$

Inferred inds:
1. $R \subset_{\theta_{R,UKQ.postcode}} \text{UKQ.postcode}$, $\theta_{R,UKQ.postcode} = \frac{3}{12} = 0.25$
2. $\text{UKD.postcode} \subset_{\theta_{UKQ.postcode,R}} R$, $\theta_{UKQ.postcode,R} = \frac{3}{3} = 1.0$

**Cases 8 & 17.** This example is depicted in Figure 3.2(h) where the postcode values from UKQ do not have any common values with CA.postcode, but both UKQ and CA have overlapping values with UKD.

Distinct Values: $V(\text{CA.postcode}) = 5$, $V(\text{UKD.postcode}) = 6$,

$V(R) = 5 + 6 - 2 = 9$, $V(\text{UKQ.postcode}) = 5$

Propagated inds:
1. $\text{UKD.postcode} \subset_{0.334} \text{UKQ.postcode}$
2. $\text{UKQ.postcode} \subset_{0.4} \text{UKD.postcode}$

Inferred inds:
1. $R \subset_{\theta_{R,UKQ.postcode}} \text{UKQ.postcode}$, $\theta_{R,UKQ.postcode} = \frac{6 + 0.334}{9} = 0.222$
2. $\text{UKD.postcode} \subset_{\theta_{UKQ.postcode,R}} R$, $\theta_{UKQ.postcode,R} = \frac{6 + 0.334}{5} = 0.4$

**Cases 9 & 18.** This example is depicted in Figure 3.2(i) where there are no full inclusion dependencies between the three attributes, i.e., there are only partial
overlaps between each pair of attributes. Given this, it is not possible to produce an accurate estimate for the overlaps and the new overlaps are pessimistically estimated, i.e., we assume that the number of common values between $UKQ.postcode$ and the resulting attribute $R$ remains as small as possible. Another approach would have been to estimate optimistically, i.e., assume that the overlap is as large as possible. However, this optimistic approach could give rise to unhelpful merge opportunities, as the latter are chosen based on the degree of overlap.

**Distinct Values:** $V(CA.postcode) = 7$, $V(UKD.postcode) = 6$, $V(R) = 7 + 6 - 3 = 10$, $V(UKQ.postcode) = 6$

**Propagated inds:**

1. $UKD.postcode \subset_{0.5} UKQ.postcode$
2. $UKQ.postcode \subset_{0.5} UKD.postcode$

**Inferred inds:**

1. $R \subset_{\theta_{R,UKQ.postcode}} UKQ.postcode$, $\theta_{R,UKQ.postcode} = \frac{6 \cdot 0.5 + 7 \cdot 0.428 - 6 \cdot 0.5}{10} = 0.3$
2. $UKD.postcode \subset_{\theta_{UKQ.postcode,R}} R$, $\theta_{UKQ.postcode,R} = \frac{6 \cdot 0.5 + 7 \cdot 0.428 - 6 \cdot 0.5}{7} = 0.428$

Based on the actual data, the real overlaps should have been $\theta_{R,UKQ.postcode} = \frac{4}{10} = 0.4$ and $\theta_{UKQ.postcode,R} = \frac{4}{7} = 0.571$. It can be observed that the estimated overlaps are smaller than the real ones, i.e., are pessimistic in general.

**Cases 10 & 19.** This example is not depicted in the set of figures as, here, the parent attribute is no longer an attribute that is being merged with another, and given that the merge is lossless, then every other attribute that is not used in the merge will keep exactly the same distinct values as before the merge, thus, any overlap that the parent attribute had is propagated unchanged to the child.

**Propagated inds:**

1. $UKD.county \subset_{\theta} UKQ.county$
2. $UKQ.county \subset_{\theta'} UKD.county$

**Inferred inds:**

1. $X \subset_{\theta} UKQ.county$
2. $UKD.postcode \subset_{\theta'} X$
3.6.2.3 Lossy Merges

Table 3.3 is used in the case of a \textit{lossy merge}, i.e., the merge operator is \textit{join}, where some of the attribute values from the \textit{referenced} table are \textit{lost}. For these examples, we assume that there is a full inclusion dependency between \textit{UKD.postcode (S)} and \textit{CA.postcode (P)}, i.e., $\theta_{UKD.postcode,CA.postcode} = 1$, meaning that all the \textit{postcode} values in \textit{UKD} are fully included in the values of \textit{CA.postcode}. In other words, according to this profile data, we consider that we only have \textit{crime-rank} information for some or all \textit{postcodes} in Cambridge. In this case, $R \leftarrow UKD.postcode \cap CA.postcode (S \cap P)$ as only the values that are common between \textit{UKD.postcode} and \textit{CA.postcode} are output in $R$.

\textbf{Distinct values.} For computing the overlap, similarly to Section 3.6.2.2, the number of distinct values on each attribute needs to be estimated. For this, we use the formulas in Table 3.1 for the \textit{join} operator considering $\theta_{UKD.postcode,CA.postcode} = 1$:

- $V(R) = V(UKD.postcode)$,
- $V(X) = V(UKD.county)$, and
- $V(Y) = \begin{cases} |r|, & \text{if } V(CA.county) > |r| \\ V(CA.county), & \text{otherwise} \end{cases}$, where $V(Y)$ is an estimate of the number of distinct values in \textit{CA.county} that remain after the merge. The intuition is that we assume the attributes keep as many distinct values as possible without exceeding the estimated size for $r$.

Let us assume a set of simple example values for each attribute, and consider the Venn diagrams in Figure 3.3.

\textbf{Cases 1 & 2.} The first two cases in Table 3.3 represent the cases where both
parent mappings are involved in the propagated inclusion dependencies. This propagation would create a link between each parent and their child mapping. Here, two inclusion dependencies are inferred for each propagated inclusion dependency (one for each parent), and there are no preconditions for propagating them. An example is depicted in Figure 3.3(a) where the result of $R$ is the intersection of the two sets $S$ and $P$, i.e., is equal to $S \cap P = S$.

**Distinct Values:** $V(CA.postcode) = 5$, $V(UKD.postcode) = 2$, $V(R) = 2$

**Propagated inds:**
1. $UKD.postcode \subset 1.0 \ CA.postcode$
2. $CA.postcode \subset 0.4 \ UKD.postcode$

**Inferred inds:**
1. (a) $UKD.postcode \subset_{\theta_{UKD.postcode,R}} R$, where $\theta_{UKD.postcode,R} = 1.0$
   (b) $R \subset_{\theta_{R,CA.postcode}} CA.postcode$, where $\theta_{R,CA.postcode} = 1.0$
2. (a) $CA.postcode \subset_{\theta_{CA.postcode,R}} R$, where $\theta_{CA.postcode,R} = 2/5 = 0.4$
   (b) $R \subset_{\theta_{R,UKQ.postcode}} UKD.postcode$, where $\theta_{R,UKQ.postcode} = 1.0$

**Cases 3 & 4.** The two cases in Table 3.3 represent the cases where only one parent relation is involved in the propagated inclusion dependencies. This propagation creates a link between that parent and the child mapping. Here, one inclusion dependency is inferred for each inclusion dependency, and there are no preconditions for propagating it. An example is depicted in Figure 3.3(b) where the result of $R$ is the intersection of the two sets $S$ and $P$, i.e., is equal to $S \cap P = S$.

**Distinct Values:** $V(CA.postcode) = 5$, $V(UKD.postcode) = 2$, $V(R) = 2$, $V(UKQ.postcode) = 5$

**Propagated inds:**
1. (a) $UKD.postcode \subset 0.5 \ UKQ.postcode$
   (b) $UKQ.postcode \subset 0.2 \ UKD.postcode$
2. (a) $CA.postcode \subset 0.4 \ UKQ.postcode$
   (b) $UKQ.postcode \subset 0.4 \ CA.postcode$

**Inferred inds:**
For 1, 2(a): $R \subset_{\theta_{R,UKQ.postcode}} UKQ.postcode$,
   where $\theta_{R,UKQ.postcode} = \theta_{UKQ.postcode,UKQ.postcode} = 0.5$
For 1, 2(b): $UKQ.postcode \subset_{\theta_{UKQ.postcode,R}} R$,
   where $\theta_{UKQ.postcode,R} = \theta_{UKQ.postcode,UKD.postcode} = 0.2$

**Cases 5 & 6.** This example is not depicted in the set of figures as, here,
the parent attribute is no longer an attribute that is being merged with another. Given that (i) the merge is lossy, (ii) the parent attribute is from the dependent relation, and (iii) every attribute from the dependent table keeps the distinct values it had before the merge, any overlap that the parent attribute had is propagated unchanged to the child.

Propagated inds:

1. \( UKD\.county \subset_\theta UKQ\.county \)
2. \( UKQ\.county \subset_\theta UKD\.county \)

Inferred inds:

1. \( X \subset_\theta UKQ\.county \)
2. \( UKQ\.county \subset_\theta X \)

Cases 7 & 8. This example is not depicted in the set of figures as, here, the parent attribute is no longer an attribute that is being merged with another. The merge is lossy and the parent attribute is from the referenced relation, so every attribute from the referenced table may lose some of the distinct values it had before the merge. However, because there is no way to determine which values are kept and which are lost (if any), any overlap that the parent attribute had is propagated to the child using the estimates in Cases 7 and 8. The intuition behind the proposed estimates is that the referenced attributes keep as many distinct values as possible without exceeding the estimated size for \( r \).

3.7 Dynamap Workflow

In the previous sections of this chapter, we described the main components in the mapping generation algorithm, Dynamap. Figure 3.4 depicts the workflow of the complete mapping generation process.

Input. The basic input comprises the input sources, the schema of the target relation, and the source-to-target matches (generated with a matching tool or by an expert user).

Preprocessing. This input undergoes a preprocessing step where a profiler (e.g., Metanome [Papenbrock et al. (2015)]) analyses the source data and produces the postulated candidate keys and inclusion dependencies between sources. In the same processing step, the metadata, and database statistics are read off the sources, i.e., relation sizes, number of nulls and number of distinct values in the attributes. Then, the last preprocessing step is to generate the base mappings.
representing the base solution (sub-solution for iteration 1) that bootstraps the recursive search.

**Mapping generation as search.** The profile data and the base mappings are input to the search process and the algorithm starts building the mappings in a bottom-up fashion (recursively calling `GenerateMappings` - Algorithm 3). At each iteration, the memoized mappings are merged to generate the intermediate mappings for that iteration. Every pair of mappings from previous iterations are considered for merging. The algorithm tries to find a suitable merge operator (Algorithm 1 – `ChooseOperator`) using the available candidate keys and inclusion dependencies on the two mappings. If a suitable operator was found, a new mapping is created for which the algorithm computes the estimated size, nulls and distinct values on attributes (using the formulas in Table 3.1). New profile data is also inferred for the new mapping using the data from the parent mappings (using the formulas in Tables 3.2, 3.3, and 3.4). The new profile data is then used to create relationships between the child and the other memoized mappings that had a relationship with (at least) one of the parents. These relationships are needed in order to merge the child mapping further with other mappings in subsequent iterations. In the next step, the algorithm uses the estimated metadata and computes a fitness value for the new mapping (Algorithm 5 – `Fitness`). Dynamap then retrieves previously generated mappings (if any) that stem from the same initial source relations as the new mapping and compares the fitness of the new mapping with that of the memoized mappings. If the fitness of the new mapping is better than the one of the memoized mappings, then Dynamap discards the previous mappings and memoize the new mapping for being the best (up to this point in the search) for that subset of initial relations. The process of merging mappings is repeated until all mappings that could be merged for that iteration are combined and the sub-solution for that iteration is created.
Output mappings. In Dynamap, in different iterations, too 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 that merge a maximum of $r$ initial relations, where $k$ is an integer and $1 \leq r \leq n$, where $n$ is the total number of input source relations. The parameter $k$ is given as input by the user to the mapping generation algorithm and it is used at the end of the mapping generation process so as to filter the $k$ best mappings from the whole set of output mappings. The set of output mappings comprises intermediate mappings that merge subsets of $i$ source relations, $1 \leq i \leq r$, that were obtained in intermediate iterations, and that are ranked according to their fitness (as explained in Section 3.5). In practice, the process of mapping selection could also involve user preferences (e.g., Abel et al. (2018)).

Example 3.7.1. An example of an output mapping for the scenario in Figure 3.1 has the following form:

\[
\begin{align*}
imap_{CA \_ MA} & \leftarrow CA \cup MA \\
imap_{CA \_ MA \_ UKD} & \leftarrow imap_{CA \_ MA} \bowtie_{postcode} UKD \\
output\_ map & \leftarrow imap_{CA \_ MA \_ UKD} \bowtie_{county} UKQ
\end{align*}
\]

where $output\_ map$ has the same schema as the chosen target relation, e.g., $CA\_ MA\_ statistics(postcode, price, street, income\_ rank, crime\_ rank)$. Note that it can be seen as a combination of horizontal and vertical fragments, so to speak.

3.8 Algorithm Evaluation

In this section, we evaluate the performance of Dynamap on various synthetic scenarios (Section 3.8.1) and a part of a larger real-world scenario (Section 3.8.2). The purpose of this section is to evaluate the extent to which Dynamap can tackle mapping generation scenarios created by a state-of-the-art benchmark ($iBench$), and representative real-world scenarios. Here, we only show results for a part of a larger real-world scenario because we aim to follow step-by-step what Dynamap does in this case for the workflow described in Section 3.7. The results on the complete real-world scenario are described in Section 4.5.3.

Experimental setup. We run Dynamap over relational data sources and target schemas. For storage, we used PostgreSQL 9.6. For the real-world scenarios, in order to maintain a focus on mapping generation, matches were generated
3.8. ALGORITHM EVALUATION

by a human expert. The profile data was generated through two Metanome modules, i.e., HyUCC [Papenbrock and Naumann (2017)] for candidate keys, and Sindy [Kruse et al. (2015)] for (partial) inclusion dependencies. In the case of the synthetic scenarios, the matches, the profile data, the source and the target schemas are generated automatically (without human input) by the scenario generator. The experiments were run in an Intel Core i5 with 2×2.7 GHz, and 8 GB RAM.

Evaluation. The evaluation depends on the scenario type:

iBench scenarios. Given that here we test each basic integration scenario at a time (named primitives by iBench), we compare the output of the mapping created with Dynamap with the output of the mapping indicated by iBench as being correct.

Real-world scenario. The result of the output mapping is compared with that of a ground-truth mapping. Given that the scenario is of a reasonable size, the comparison is done in terms of expected and generated operators between the sources.

3.8.1 Benchmark Experiment - iBench

Motivation. iBench [Arocena et al. (2015)] is a tool that generates data integration/exchange scenarios where the sources have explicit keys and foreign keys. Although not over autonomous data sources, i.e., not in the wild, these scenarios are relevant for our purpose as they make use of a variety of data integration primitives that mapping generation algorithms must (ideally) 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.

Scenarios. We have generated a separate scenario for each type of base case that iBench is able to generate. Given that all the relationships between the sources are automatically created as explicit foreign keys, there is no need to populate the sources with data and analyze them with a profiler tool as the required profile data is already specified. Also, the source-to-target matches are automatically created by the generator. For generating the scenarios, we set the arity range to [4-7], i.e., the source and target relations that are created can have between four and seven attributes.
The primitives that iBench makes use of are:
1. CP - Copy a relation.
2. HP - Horizontal partition a relation in multiple relations.
3. ADD - Copy a relation and add new attributes.
4. DEL - Copy a relation and delete attributes.
5. ADL - Copy a relation and add and delete attributes in tandem.
6. ME - Merge two source relations.
7. MA - Merge two source relations and add attributes.
8. VP - Vertical partitioning.
9. VHA - Vertical partitioning into a HAS-A relationship.
10. VIA - Vertical partitioning into an IS-A relationship.
11. VNM - Vertical partitioning into an N-to-M relationship.
12. SU - Copy a relation and create a surrogate key.
13. SJ - Copy a relation (S) and a relationship table (T) through a self-join.

Results. We ran Dynamap over the iBench generated scenarios to ascertain whether Dynamap correctly handles that scenario. The results can be seen in Table 3.5. Dynamap correctly handles 7 out of 13 scenarios, which is the expected outcome since these are the cases where either the source data needs merging (through a foreign key) or simply copying the data (without any merge) to the target relation. We now explain why Dynamap does not handle the remaining six scenarios ¹.

Vertical Partitioning. The VP primitive is described by the transformation
\[ R(a, b) \rightarrow S_1(f(a, b), a) \land S_2(f(a, b), b), \quad S_1.f(a, b) \text{ references } S_2.f(a, b), \quad \text{and vice versa.} \]

It can be observed that, in this primitive, the source data is transferred from a single relation to two target relations linked through a foreign key. As stated in the preamble of this chapter, Dynamap does not take into consideration target constraints, and, as a result, in vertical partitioning scenarios, Dynamap cannot make use of the specified foreign key relationship in the target, and therefore creates two separate mappings to populate \( S_1 \) and \( S_2 \), separately.

The next three primitives, i.e., VHA, VIA, and VNM, are variations of vertical partitioning (VP):

• (VH) Vertical partitioning in a HAS-A relationship:

¹These types of scenarios are handled through the extended version of Dynamap, i.e., Dynamap³, which we describe in Chapter 5.
### 3.8. ALGORITHM EVALUATION

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Transformation</th>
<th>As Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>( R(a,b) \rightarrow T(a,b) )</td>
<td>Yes</td>
</tr>
<tr>
<td>HP</td>
<td>( R(a,b) \rightarrow T_1(b), R(a,b) \rightarrow T_2(b) )</td>
<td>Yes</td>
</tr>
<tr>
<td>ADD</td>
<td>( R(a,b) \rightarrow T(a,b,f(a,b)) )</td>
<td>Yes</td>
</tr>
<tr>
<td>DEL</td>
<td>( R(a,b) \rightarrow T(a) )</td>
<td>Yes</td>
</tr>
<tr>
<td>ADL</td>
<td>( R(a,b) \rightarrow T(a,f(a)) )</td>
<td>Yes</td>
</tr>
<tr>
<td>ME</td>
<td>( R(a,b) \land S(b,c) \rightarrow T(a,b,c) ), and ( S.b ) references ( R.b ).</td>
<td>Yes</td>
</tr>
<tr>
<td>MA</td>
<td>( R(a,b) \land S(b,c) \rightarrow T(a,b,c,f(a,b,c)) ), and ( S.b ) references ( R.b ).</td>
<td>Yes</td>
</tr>
<tr>
<td>VP</td>
<td>( R(a,b) \rightarrow S_1(f(a),b) \land S_2(f(a),b) ), ( S_1.f(a,b) ) references ( S_2.f(a,b) ), and vice versa.</td>
<td>No</td>
</tr>
<tr>
<td>VHA</td>
<td>( R(a,b) \rightarrow S(f(a),a) \land T(g(a),b,f(a)), T.f(a) ) references ( S.f(a) )</td>
<td>No</td>
</tr>
<tr>
<td>VNM</td>
<td>( R(a,b) \rightarrow S_1(f(a),a) \land M(f(a),g(b)) \land S_2(g(b),b) ) where ( M.f(a) \rightarrow S_1.f(a) ) and ( M.g(b) \rightarrow S_2.g(b) ) are FKs</td>
<td>No</td>
</tr>
<tr>
<td>SU</td>
<td>( R(a,b) \rightarrow T(f(a),b,g(b)) ), where ( T.f(a,b) ) is a key</td>
<td>No</td>
</tr>
<tr>
<td>SJ</td>
<td>( R(a,b,c) \rightarrow T(a,b), R(a,b,c) \land R(b,d,e) \rightarrow T(a,b) ), and ( R.b ) references ( R.a ).</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.5: Results for iBench primitives

\[
R(a,b) \rightarrow S(f(a),a) \land T(g(a),b,f(a)), \text{ where } T.f(a) \rightarrow S.f(a) \text{ is a FK}
\]

- (VI) Vertical partitioning in a IS-A relationship:
  \[
  R(g,b,c) \rightarrow S(g,b) \land T(g,c), \text{ where } T.a \rightarrow S.a \text{ and } S.a \rightarrow T.a \text{ are FKs}
  \]

- (VNM) Vertical partitioning in N-to-M relationship:
  \[
  R(a,b) \rightarrow S_1(f(a),b) \land M(f(a),g(b)) \land S_2(g(b),b) \text{ where } M.f(a) \rightarrow S_1.f(a) \text{ and } M.g(b) \rightarrow S_2.g(b) \text{ are FKs}
  \]

For these primitives, Dynamap’s behaviour, for the same reason as above, does not handle them.

**Surrogate Key.** – The SU primitive is described by the transformation

\[
R(a,b) \rightarrow T(f(a),b,g(b)), \text{ where } T.f(a,b) \text{ is a key that is expected to be populated with labelled nulls created using a Skolem function over the source attributes.}
\]

This primitive requires that the mapping generation algorithm detects the need of unique values in the target on the declared key attribute that is, in fact, not matched by the source. In order to tackle this scenario, it needs to create unique values, i.e., labelled nulls, using the source data.

As in the case of the VP primitive, this scenario is not tackled by Dynamap because the mapping generation algorithm would need to consider key constraints on the target to produce the expected result. As before, Dynamap is not designed to make use of this information and, as a result, it generates mappings that transfer data from the matched source attributes to the target without generating labelled nulls to satisfy the key constraint.

**Self-join.** The SJ primitive is described by the transformation

\[
R(a,b,c) \rightarrow S(a,b), R(a,b,c) \land R(b,d,e) \rightarrow T(a,b), \text{ and } R.b \text{ references } R.a.
\]
This primitive implies a scenario where a source relation \((R)\) populates a target that requires its self-join, i.e., \(R(a, b, c) \land R(b, d, e)\) means that \(R.b\) references \(R.a\), thus, if a join is performed as the following \(R(a, b, c) \bowtie_{R.a=R.b} R(b, d, e)\) then it will correlate data within the same table \(R\), and populate a target table \(T\).

Here the issue is not whether constraints are taken into account, but rather the fact that it is not straightforward to establish, with the evidence available to Dynamap, when a mapping should involve a self-join. For example, assume a relation \(Employee(eID, eID\_manager, address)\) and assume that we would like to find the correlation between where employees live and where their managers live. This requires joining \(Employee(eID, eID\_manager, address)\) with itself. Suppose the target table is \(Target(addressManager, addressEmployee)\), then there are only two matches from \(Employee.address\) to the two target attributes, but this information does not imply the necessity of a self-join. It seems plausible from the limited evidence available that this is a simple copy scenario. Thus, one can say it is difficult to express such a scenario without explicitly stating the self-join requirement, and such information is not expected to be available for mapping generation in the wild.

3.8.2 Real-world Scenario - Schools Domain

**Motivation.** This section presents a part of a larger scenario, which is evaluated in full in Section 4.5.2. Here, we run Dynamap over only part of that because we aim to describe, in a step-by-step manner, the extent to which the algorithm is able to generate profile-informed mappings given a set of independent sources.

**Scenario.** The data sources contain information about schools, more specifically, about the facilities in those schools, and the target needs to gather data from all input sources.

**Data Sources.** The sources contain open-government data from the United Kingdom and do not have any declared relationships between them. Their content is outlined in Table 3.6. The table shows: the type of information made available, the attributes that each source has for the target, i.e., the matches, the number of sources containing the same type of information, the arity range, and the cardinality range.

**Target schema.** The target schema brings together the information about each school with the information about its activities and facilities, i.e., school name, school type, headteacher contact, and number of students with English as
3.8. ALGORITHM EVALUATION

<table>
<thead>
<tr>
<th>data.gov.uk source</th>
<th>Data for the target</th>
<th>#Data Sources</th>
<th>Arity Range</th>
<th>Size Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>All schools</td>
<td>dfe code(dc), school name(sn), headteacher(ht)</td>
<td>1</td>
<td>16</td>
<td>99</td>
</tr>
<tr>
<td>Additional languages</td>
<td>dfe code(dc), #students with English as additional language (adl)</td>
<td>2</td>
<td>3 - 6</td>
<td>24 - 88</td>
</tr>
<tr>
<td>Road and Safety training</td>
<td>school name(sn), school type(st)</td>
<td>1</td>
<td>3</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 3.6: Input source files - schools information

Figure 3.5: Schools scenario

additional language, dfe code:

Target(sn, st, ht, adl, dfe code)

Profile data. To obtain the profile data on the input sources, HyUCC [Papenbrock and Naumann (2017)] were run to detect the candidate keys, and SINDY [Kruse et al. (2015)] was run to obtain the (partial) inclusion dependencies. The input profile data contains: 7 candidate keys, 11 partial inclusion dependencies, and 1 full inclusion dependency. The relationships between the attributes are shown in Figure 3.5. Note that the schemas represent only subsets of attributes that either had matches to the target and/or had shared profile data with other relations.

Matches. Given that the scenario is of modest size, and that we focus on mapping generation, the matches were created by hand. There are a total of 9 matches from the sources to the target. The matched attributes for each source are in the Data for the target column in Table 3.6.

Ground truth. Given that the sources do not have any explicit schema constraints, and given the numerous relationships that can be inferred using the profile data, it is not obvious how to merge them in order to achieve good quality results. A best-effort ground-truth mapping was created by hand, as follows: first, the datasets with data about additional languages were unioned, as they contain
the same type of information needed in the target, i.e., _dfe code_ and _number of students with English as Additional Language_. Then, the other two sources, i.e., _all schools_ and _road and safety training_, were outer joined using _postcode_ attributes as it seems to be the better option, given that they have the highest overlap (0.869) across the two relations. Finally, the result of the union is merged with the result of the full outer join using, again, a full outer join operator on _dfe code_. The ground-truth mapping, therefore, corresponds to the relational algebra expression:

\[
 gt \leftarrow (AM17 \cup AM16) \bowtie_{dfe} (AS \bowtie_{pc} RST)
\]

**Mapping generation.** We describe here the complete mapping generation process, following the workflow depicted in Figure 3.4.

**Pre-processing.** The mapping generation process starts by analyzing the four input sources w.r.t. the target, i.e., read the profile data, metadata and database statistics and generate the _base mappings_ which represent the sub-solution for iteration 1 (_SS_1).

- **Metadata:**
  - relation sizes: \(|AS| = 99, |RST| = 46, |AM16| = 70, and \(|AM17| = 86,\)
  - nulls: \(nulls(AS.pc) = 0, nulls(RST.st) = 0, nulls(AM16.adl) = 0, \)
    \(nulls(AM17.adl) = 0.\) The rest of the attributes are candidate keys, i.e., they contain no nulls.
  - distinct values: \(V(AS.pc) = 96, V(RST.st) = 8, V(AM16.adl) = 34, \)
    \(V(AM17.adl) = 36.\) The rest of the attributes are candidate keys, i.e.,
    they contain the same number of distinct values as the size of their corresponding relations.

- **Profiling data (as seen in Figure 3.5):**
  - 7 candidate keys, e.g., _AS.pc_, _AM16.dc_
  - 11 partial inclusion dependencies, e.g., _AS.dc \subset_{0.86} AM16.dc_
  - 1 full inclusion dependency: _AM16.dc \subset_{1.0} AS.dc_

- **Base mappings (sub-solution for iteration 1 - _SS_1):**
  - All schools: _imapAS_ \(\leftarrow \pi_{sn,ht,pc,dc}AS\)
  - R&S training: _imapRT_ \(\leftarrow \pi_{sn,st,pc}RST\)
  - ADL May17: _imapA17_ \(\leftarrow \pi_{dc,adl}AM17\)
  - ADL May16: _imapA16_ \(\leftarrow \pi_{dc,adl}AM16\)
3.8. ALGORITHM EVALUATION

Dynamic programming search. Next, the algorithm starts building the mappings in a bottom-up fashion. First, it calls \texttt{GenerateMappings} (Algorithm 3) with \( i = 4 \), i.e., try to retrieve the mappings for the set of four sources. At this point, there are no mappings generated (besides the base mappings), so it will recursively call \texttt{GenerateMappings} for the pairs \((i = 2, j = 2)\) and \((i = 1, j = 3)\) (lines 7-8, in Algorithm 3). Again, there are no mappings for subsets of 2, and 3 sources, respectively. When it steps in \texttt{GenerateMappings} with \( i = 2 \), it will recursively call \texttt{GenerateMappings} with \( i = 1 \) twice for which the base mappings will be returned. Now, it can start merging pairwise the base mappings. The result of \texttt{GenerateMappings} with \( i = 2 \) will represent the mappings generated in iteration 2. After the mappings for iteration 2 are created, the mappings for iteration 3 can be created, i.e., for the call of \texttt{GenerateMappings} with input parameter \( i = 3 \) it will recursively call \texttt{GenerateMappings} with \( i = 1 \) (corresponding to iteration 1) and \texttt{GenerateMappings} with \( i = 2 \) (corresponding to iteration 2) for which the base mappings and the iteration 2 mappings are returned, respectively. After the mappings for each 3 sources are generated, the algorithm will return to the initial call of \texttt{GenerateMappings} for \( i = 4 \), and now the set of mappings for all 4 sources can be created. The mappings with all four relations are obtained merging the mappings from iteration 3 with those in iteration 1, and pairwise the mappings from iteration 2. After this last step, the recursive search ends. Below, we will explain step by step each of the iterations in a bottom-up fashion.

\textbf{Iteration 2 \((\text{for GenerateMappings}(i=2))\)}

Choose operator & create mappings. Table 3.7 shows the intermediate mappings created in iteration 2. In most cases, Dynamap finds the expected merges according to the ground truth. However, when deciding to merge \( AS \) with \( RST \), it chooses the expected operator (\textit{full outer join}), but it does not perform the join on the expected condition, i.e., on \textit{postcode}, instead it uses \textit{school_name}. The ground truth was set to join the two relations on \textit{postcode} attributes as this

<table>
<thead>
<tr>
<th>SS1</th>
<th>SS2</th>
<th>AS</th>
<th>RST</th>
<th>AM16</th>
<th>AM17</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>imap2,1 ← AS (\bowtie) \textit{RST}</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>RST</td>
<td>imap2,2 ← AS (\bowtie) \textit{AM16}</td>
<td>NOP</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>AM16</td>
<td>imap2,3 ← AS (\bowtie) \textit{AM17}</td>
<td>NOP</td>
<td>imap2,4 ← AM16 (\cup) AM17</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

\(\text{NOF} = \text{no operator found} \)
pair has the highest overlap, i.e, \( \theta_{\text{postcode}} = 0.869 \). Referring back to `Choose-OperatorDiff` (Algorithm 2), we can see that Dynamap prefers condition attributes where both are candidate keys, and in this situation, between `AS` and `RST` only the `school_name` attributes are both keys, not the `postcode`, thus, joins on `school_name`. Looking closer into the data, we find that `AS.postcode` is not a key, but `AS.school_name` is, then at least two different schools are based at the same postcode, e.g., a highschool and a primary school are in the same building. Merging `AS` and `RST` on `postcode` means that some tuples in `AS` are assigned the wrong `school_type` (from `RST`) as more than one institution has the same `postcode`. This merge would lead to incorrectly aligned tuples. Moreover, merging the two tables using keys ensures that the profile data is propagated more accurately as the merged tuples do not produce duplicates on the attributes that are not involved in the merge, e.g., `AS.dfe_code` would still have unique values after `AS` and `RST` merged by `school_name`. To conclude, Dynamap prefers merges based on pairs of keys to inclusion dependencies with higher overlaps. This leads to fewer correlated tuples, but those tuples are all correctly aligned in terms of semantics.

**New metadata.** For each of the intermediate mappings, e.g., `imap_{i, j}`, \( i \in \{1, 2, 3, 4\} \), new metadata is computed. We describe below how this is done for one of the intermediate mappings, as it is the same for the others.

For `imap_{2, 4} ← AM16 \cup AM17`, the metadata is estimated using Table 3.1.

- \( |imap_{2, 4}| ← |\pi_{dc, adl} AM16| + |\pi_{dc, adl} AM17| - |AM16 \bowtie_{dc, adl} AM17| \). However, we do not know the size of the join result as the number of overlapping values between pairs of attributes \((dc, adl)\) from the two relations, `AM16` and `AM17`, cannot be derived from the available information. Thus, the result size is estimated considering the join as the empty set, assuming that the mapping creates as many tuples as possible, i.e., \( |imap_{2, 4}| ← 156 \).
- \( \text{nulls}(imap_{2, 4}.dc) ← 0, \text{nulls}(imap_{2, 4}.adl) ← 0 \),
- \( V(imap_{2, 4}.dc) ← 91, V(imap_{2, 4}.adl) = 52 \).

**New profile data.** For each of the intermediate mappings, new candidate keys and inclusion dependencies are inferred from the parent mappings. We will describe below how this inference is carried out for the intermediate mapping, `imap_{2, 4}`, as the others are done in a similar fashion.

As `imap_{2, 4}` is the result of a `union` operation, the merge is lossless, thus, Table 3.2 is used for propagating all 8 inclusion dependencies and Table 3.4 for the 2
3.8. ALGORITHM EVALUATION

candidate keys.

The propagation of $AS.dc \subset \theta$ to $AS.dc \subset \theta$ is done using the formula for Case 14 in Table 3.2: $\theta \leftarrow \frac{86 - 86 + 0.928 + 70 + 0.928}{99}$, so $\theta = 0.718$.

The propagation of $AS.dc \subset \theta$ to $AS.dc \subset \theta$ is done using the formula for Case 15 in Table 3.2: $\theta \leftarrow \frac{86 - 86 + 0.928 + 70 + 0.928}{99}$, so $\theta = 0.718$.

Although two different inclusion dependencies were propagated (relating mappings $AM16$ and $AM17$ both to mapping $AS$), the two inferred inclusion dependencies are, in fact, the same with the same inferred degree of overlap because the new intermediate attribute ($R \leftarrow AM17.dc \cup AM16.dc$) has both inclusion dependencies relating to the same other attribute, $AS.dc$.

*Fitness.* The fitness function relies on the number of null values. Computing the fitness of a new mapping depends on the chosen operator for its creation. For instance, for $imap_{2,4}$, which resulted from a union, there are no new nulls on the attributes in the target, so the fitness of the new mapping is equal to its estimated size, i.e., the number of largely complete tuples, so $fitness(imap_{2,4}) = 156$. For the other mappings, the fitness value depends on the number of nulls on the attributes needed in the target. For instance, $imap_{2,1}$ resulted from a full outer join, so the fitness value is $fitness(imap_{2,1}) = 99$ although its estimated size is $|imap_{2,1}| = 124$. This is because the total size of the mapping is not equal to the estimated number of (largely) complete tuples, as we know that some tuples will be sparse (viz., the ones that do not merge).

*Memoize/Discard mappings.* In iteration 2, all created mappings are memoized as sub-solution $SS_2$ as there are no other memoized mappings that are better than the ones produced in this iteration.

The memoized mappings for sub-solution for iteration 2 is

$$SS_2 = \{imap_{2,1}, imap_{2,2}, imap_{2,3}, imap_{2,4}\}$$

**Iteration 3 (for GenerateMappings($i$=3)):**

*Choose operator & create mappings.* For computing the sub-solution in iteration 3, the memoized intermediate mappings from $SS_2$ are merged with the ones in $SS_1$ (the base mappings). In Table 3.8, the merges that were found in this step are described.

Note the related case to the one explained above, in iteration 2: $imap_{2,2} \leftarrow AS \triangleright_{dc} AM16$ is a mapping built through a full outer join on the $dc$ attributes. Given this merge, candidate keys in both parent mappings are lost due to the creation of nulls on the attributes not used in the join condition. The only
candidate key remaining in $imap_{2,2}$ is on $dc$. In iteration 2, we explained that a merge between $RST$ and $AS$ is made on the `school_name` as it avoids aligning incorrect tuples for schools with the same postcode. Now, for merging $AS$, $AM16$, and $RST$, Dynamap has no other choice than to merge them using the `postcode` attribute as this is the only candidate key (in $RST$) that shares an inclusion dependency between $imap_{2,2}$ and $RST$. This leads to a case where some tuples might align incorrect data, but avoiding this (as was done in iteration 2) is no longer possible because the available profile data does not enable any other merge. This shows that the design choice in the algorithm to prioritize the maintenance of accurate profile data is a reasonable one as it postpones or completely avoids merge decisions that might lead to semantic inconsistencies. A faulty mapping that is generated early in the iterations can proliferate to more incorrect mappings.

New metadata. For each of the intermediate mappings, e.g., $imap_{3,i}$, $i \in \{1, 2, 3, 4, 5, 6, 7\}$, new metadata is computed, as before, using Table 3.1.

New profile data. Profile data from the parent mappings are propagated to the new intermediate mappings. The profile data from the parent mappings can either come from the previously inferred profile data (e.g., in iteration 2), or from the initial profile data if one of the parents was a base mapping.

Fitness. In this step, the differences between the fitness values and the size of the mappings becomes more apparent, especially in the cases where full outer join operators are nested, as these are prone to creating nulls on the attributes, thus, the number of largely complete tuples is estimated to diminish.

For instance, $imap_{3,5}$ is the result of one full outer join and one join operation. Its fitness value is $\text{fitness}(imap_{3,5}) = 86$ while its size is $|imap_{3,5}| = 111$.

Memoize/Discard mappings. In this iteration, the algorithm starts discarding
mappings for which it finds better mappings (in terms of fitness) which stem from the same initial relations.

For example, in this scenario, imap_{3,5} and imap_{3,2} both stem from AS, AM17, and RST, so the algorithm chooses the fittest one. In this case, the fittest one is imap_{3,2} as fitness(imap_{3,5}) = 86 and fitness(imap_{3,2}) = 99, so imap_{3,5} is discarded, while imap_{3,2} is memoized to be used in the next iteration(s).

At the end of the iteration, the memoized mappings for the sub-solution at iteration 3 are \( SS_3 = \{imap_{3,2}, imap_{3,3}, imap_{3,7}\} \).

**Iteration 4 (for \texttt{GenerateMappings}(i=4))**:  
Choose operator \& create mappings. This iteration is the last one as the total number of input sources is four. For building the mappings for this iteration, the mappings from \( SS_1 \) are merged with the mappings from \( SS_3 \) and then pairwise the mappings from \( SS_2 \). Tables 3.9 and 3.10 describe the merges that were found in this step.

Looking at the merge between imap_{2,4} and imap_{2,1}, one would say that it was expected to (outer) join as they seem to have an overlap between them on dfe\_code. However, this does not happen because dfe\_code is no longer a candidate key in either of the two mappings as it was not propagated. For imap_{2,4}, it was not propagated because it can have duplicate values (due to the overlap between the parent attributes) and, in the case of imap_{2,1}, it is predicted to contain nulls as the two parent mappings are merged using full outer join, so the resulting dfe\_code attribute is predicted to contain nulls.

**New metadata.** The metadata created here is still needed to compute the fitness value of the mappings, so as to choose the fittest one among the created ones. The procedure to compute the estimates is the same as in the previous iterations.

For comparing the mappings, we mention here their estimated sizes:

\[
\begin{align*}
|imap_{4,1}| &= 129, \\
|imap_{4,2}| &= 110, \text{ and} \\
|imap_{4,3}| &= 175.
\end{align*}
\]

**New profile data.** Given that this is the last iteration, the profile data is no longer propagated as this is the last step and no further merges are necessary.

**Fitness.** Similar to iteration 3, at this step, the algorithm needs to compute the fitness of the mappings w.r.t. populating the chosen target.

The fitness of the three generated mappings are:
# CHAPTER 3. MAPPING GENERATION FOR A SIMPLE TARGET

**Table 3.9: Schools scenario - merges \(SS_2, SS_2\) in iteration 4**

<table>
<thead>
<tr>
<th>(SS_1)</th>
<th>(SS_2)</th>
<th>(imap_{2,1})</th>
<th>(imap_{2,2})</th>
<th>(imap_{2,3})</th>
<th>(imap_{2,4})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SS_1)</td>
<td>(SS_2)</td>
<td>CA</td>
<td>CA</td>
<td>NOF</td>
<td></td>
</tr>
<tr>
<td>(imap_{2,1})</td>
<td>–</td>
<td>CA</td>
<td>CA</td>
<td>NOF</td>
<td></td>
</tr>
<tr>
<td>(imap_{2,2})</td>
<td>CA</td>
<td>–</td>
<td>CA</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>(imap_{2,3})</td>
<td>CA</td>
<td>CA</td>
<td>–</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>(imap_{2,4})</td>
<td>NOF</td>
<td>CA</td>
<td>CA</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

*NOF = no operator found
CA = mappings were not merged as they have a common ancestor*

**Table 3.10: Schools scenario - merges \(SS_1, SS_3\) in iteration 4**

<table>
<thead>
<tr>
<th>(SS_1)</th>
<th>(SS_3)</th>
<th>(imap_{3,2})</th>
<th>(imap_{3,3})</th>
<th>(imap_{3,7})</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>CA</td>
<td>CA</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td>CA</td>
<td>CA</td>
<td></td>
<td>(imap_{3,3} \leftarrow RST \bowtie_{pc} imap_{3,7})</td>
</tr>
<tr>
<td>AM16</td>
<td>(imap_{4,1} \leftarrow AM16 \bowtie_{dc} imap_{4,2})</td>
<td>CA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM17</td>
<td>CA</td>
<td>(imap_{4,2} \leftarrow AM17 \bowtie_{dc} imap_{4,3})</td>
<td>CA</td>
<td></td>
</tr>
</tbody>
</table>

*CA = mappings were not merged as they have a common ancestor*

\[\text{fitness}(imap_{4,1}) = 99,\]
\[\text{fitness}(imap_{4,2}) = 104,\] and
\[\text{fitness}(imap_{4,3}) = 169.\]

**Memoize/Discard mappings.** This is the last step before returning the memoized mappings as the solution to the complex problem the algorithm started with.

At the end of the iteration, the memoized mapping for the sub-solution of iteration 4, which is the final solution, is \(SS_4 = \{imap_{4,3}\}\) as it has a better fitness than both \(imap_{4,1}\) and \(imap_{4,2}\) which stem from the same initial relations.

**Output mappings.** In the output of the mapping generation process, with the current approach (as mentioned in Section 3.7), the top \(k\) memoized mappings are output. For this experiment, we set \(k = 100\), so all intermediate mappings that were kept will be output.

Our set ground truth mapping is:
\[gt \leftarrow (AM17 \cup AM16) \bowtie_{dc} (AS \bowtie_{pc} RST)\]

The mapping that was considered as being the fittest is \(imap_{4,3}\) mapping in Table 3.10, which, using the previously generated mappings, unfolds into the
3.9 Conclusions

In Section 2.3, we discussed assumptions on mapping generation that no longer hold for mapping generation over autonomous sources. They are:

- $A_1$: data sources have well-defined schemas,
- $A_2$: all the available sources need to be integrated,
- $A_3$: the input data is mostly consistent,
- $A_4$: the global schemas have reasonable sizes.

In this chapter, we have addressed challenges arising from the violation of $A_1$–$A_4$ as follows:

1. We have addressed the challenge stemming from the violation of $A_1$ by devising a decision procedure that combines pairs of different sources using profile data, metadata and statistics about the sources. The algorithm was described in Section 3.3. In Section 3.8, we showed that the algorithm can handle the mapping generation scenarios posed by state-of-the-art benchmarks except for the case of complex target schema (which we address in Chapter 5). These scenarios comprise (foreign key) merges between the sources and/or copying of the data to a target without constraints. The experiment on real-world data, in Section 3.8.2, shows that the designed profile-informed technique leads to sensible decisions when building mappings for real-world independent data sources.

2. We have addressed the challenge stemming from the violation of $A_2$ by proposing a dynamic programming approach to explore the search space of candidate mappings that is able to create complex mappings in a bottom-up fashion, choosing the best merge opportunities on the basis of propagated profiling data and a fitness measure that prefers largely complete tuples. Section 3.4 described how the dynamic programming paradigm was adapted to our search problem and the experiment in Section 3.8.2 shows how the approach explores all possible merge combinations. Although the algorithm
is capable of generating all combinations, we proposed a fitness function (in Section 3.5) that allows the algorithm to keep only the mappings that are predicted to satisfy the desiderata expressed through the fitness function. The fitness function can, of course, be changed so that the algorithm can support search on different criteria defined through the fitness function.

3. We have addressed the challenge stemming from the violation of $A_3$, i.e., dealing with data inconsistencies, by accepting that data from different origins (and/or domains) cannot be perfectly aligned, thus, making best-effort decisions to merge such heterogeneous information is essential. In Section 3.3, we discussed our design decisions for a set of merge conditions for combining the relations in a best-effort manner using the available (possibly scant) evidence. In Section 3.6, we introduce a profile data propagation technique to infer relationships between non-materialized mappings, which facilitates the search on profile-informed merge opportunities without the need to materialize the intermediate mappings.

4. We have addressed the challenge stemming from the violation of $A_4$ by generating mappings over global schemas of any size in terms of number of relations and/or arities. However, Dynamap is, by design, oblivious of the existence of constraints on the target schema. We return to this issue in Chapter 5, where we extend Dynamap to handle this case as well.

Discussion. In the past twenty years, mapping generation has been the subject of constant research following the seminal work on Clio [Miller et al. (2000)]. However, until now, all proposals assume the sources to be well-behaved, i.e., they have declared metadata including foreign key constraints [Mecca et al. (2009)] and keys [Marnette et al. (2010)]. This assumption does not hold for mapping generation in the wild, i.e., over sources that are autonomous and about which not much is known (e.g., web-extracted data).

Mapping generation in the wild faces challenges brought by heterogeneous data from (possibly) different domains. In the light of integrating autonomous sources, the problem of finding related sources has been the focus of work such as Das Sarma et al. (2012), Zhu et al. (2016), and Nargesian et al. (2018), where methods are proposed for detecting relationships between sources. Das Sarma et al. (2012) contribute a method for detecting various types of relatedness, among which one for detecting if the sources from a repository of heterogeneous relations are unionable and/or joinable. They translate this in terms of entity complementarity
(i.e., candidates for union) or schema complementarity (i.e., candidates for join) based on the combination of projections and selections. The problem they tackle is that of finding the $k$ tables that are most related to an input target table, where the desired type of relatedness, i.e., entity or schema complementarity, is given. In Zhu et al. (2016), they detect join paths between sources based on the domains of the attributes. If the domains are similar, then they consider the sources to be joinable on that pair of attributes. A similar idea can be found in Nargesian et al. (2018), where they propose the idea of table unionability, i.e., if pairs of attributes in different sources have similar domains, then the sources are considered as being unionable. These approaches differ from ours as our focus is on building bottom-up mappings between multiple sources w.r.t. to a target, while those contributions focus on finding related sources w.r.t. a target. Other authors try to address the challenges brought by heterogeneous data by proposing different approaches such as aligning source attributes with counterparts in a mediated schema [Sarma et al. (2008)]; by using feedback [Belhajjame et al. (2013)], or through data fusion [Stonebraker et al. (2013); Fernandez et al. (2017)]. However, these projects explore different aspects of a problem that has quite a few of them.
Chapter 4

Mapping Generation at Scale

"The computing scientist's main challenge is not to get confused by the complexities of their own making."


Referring back to the objectives outlined in Section 1.3, this chapter addresses Objective 2, viz., a method for generating mappings between a large set of source schemas and one target schema (i.e., tackles the same mapping generation problem as in Chapter 3), and describes techniques that allow Dynamap to perform mapping generation at scale. By mapping generation at scale we mean that, with the adoption of the pruning strategies presented in this chapter, Dynamap is able to tackle input scenarios where the number of sources scales to hundreds of sources. When a source repository contains thousands or millions of sources, it is unlikely that they are all equally relevant, thus, we assume that mapping generation is preceded by a dataset discovery method such as Castro Fernandez et al. (2018). This chapter, therefore, describes a set of pruning strategies for the search space traversed by Dynamap for profile data, the combination of which contains the growth of the search space.

Motivation. In Chapter 3, we described Dynamap, a mapping generation algorithm, that tackles the problem of generating mappings between autonomous heterogeneous sources. At the core of Dynamap is a search process based on the dynamic programming paradigm. However, dynamic programming performs well when the number of combinations of sub-problems can be kept small [Aho and Hopcroft (1974)]. Otherwise, it may attempt too many different ways of solving the same (sub-)problem. This leads to a large number of combinations, which compromises scalability. Since the mapping generation problem that we are
4.1 Algorithm Complexity

As explained in Section 3.4, we use dynamic programming for merging multiple smaller mappings to create larger mappings until all relations are merged (if possible) in the final iteration.

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. As shown in Algorithm 4, for each subset of initial relations just one fittest mapping is retained. This being the case, the maximum number of kept mappings as a sub-solution $i$ (corresponding to an iteration $i$) is the number of possible combinations of $i$ initial relations from the set of $N$ initial relations:

$$C_N^i = \frac{N!}{i!(N-i)!}$$  \hspace{1cm} (4.1)

This upper limit can be reached in scenarios where all initial relations merge pairwise, e.g., in a union-dominated scenario where all initial relations have the same schema as the target. Although the maximum number of memoized mappings for an iteration $i$ is $C_N^i$, the algorithm tries to merge many more pairs of mappings than are actually kept as a sub-solution for that iteration. 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:

\[ \sum_{i=1}^{N} \sum_{j=1}^{\frac{1}{2} C_N^i} C_N^{i-j} \]  

(4.2)

where \( N \) is the total number of initial source relations.

This worst case behavior represents a combinatorial explosion in the search
space, so the approach can only be practical if: (i) the fraction of the mappings
that can be combined by \texttt{MergeMappings} is small; and (ii) the search space
is pruned to avoid the retention of less promising candidate mappings. Section
4.3 describes the pruning strategies, and Section 4.5 evaluates the approach for
different sizes of integration problem on both synthetic and real-world data.

4.2 Profiling Data Propagation Complexity

Consider a set of \( N \) source relations \( R_1, R_2, \ldots, R_N \), where each of them is of the
form \( R_k(a_1, a_2, \ldots, a_{n_k}) \), where \( k \in \{1, 2, 3, \ldots, N\} \) and \( n_k \) is the arity of relation
\( R_k \) and where is a finite integer, \( n_k > 0 \).

For profile data propagation, in a worst-case scenario, all the attributes in
the sources are kept in the schemas of the generated intermediate mappings.

For example, if \( m \) relations merge, \( R_1, R_2, \ldots, R_m \) \( m \in \{1, 2, 3, \ldots, N\} \), then
\( \text{arity(schema(merged(R_1, R_2, \ldots, R_m)))} = n_1 + n_2 + \cdots + n_m \). Here, \texttt{merged}
abstracts over the applied operations. Transferring all initial source attributes
to the child mapping means that all the initial inclusion dependencies that a
source attribute had are propagated to intermediate mappings that keep it in
their schema. This leads to the idea that in a worst-case scenario, all initial
source relations are (outer) joined as only through join the attributes of the
parent mappings are transferred to the schema of the resulting intermediate
mapping. In the case of a \texttt{union}, \texttt{Dynamap} needs to make the two operands schema
compatible by keeping only the unionable attributes in the schema of the newly-
generated intermediate mapping. Thus, not all initial inclusion dependencies
would be propagated for the child mapping, i.e., only the inclusion dependencies
corresponding to the transferred attributes are propagated.

Also, in a worst-case scenario, all attributes of each source relation have
inclusion dependencies with all the attributes in the other relations:

At an iteration $i < N$, each source attribute $a_q \in \text{schema}(R_k)$ of a source relation $R_k$ with arity $n_k$, where $k \in \{1, 2, \ldots, N\}$ and $1 \leq q \leq n_k$, has a corresponding number of inferred inclusion dependencies given by the formula in Eq. 4.3.

$$2 \cdot \left( \sum_{j=1}^{i} \left( \sum_{q_1, q_1 \neq k}^{N} \cdots \sum_{q_j, q_j \neq k}^{N} (n_{q_1} + \cdots + n_{q_j}) \right) \right)$$ (4.3)

The intuition behind the formula is that the initial set of inclusion dependencies does not contain inclusion dependencies between attributes of the same relation (thus not counting the inferred inclusion dependencies between mappings that involve relation $R_k$). The sum on $j$ represents the number of previous iterations, thus considering all intermediate mappings that were generated in the previous iterations. The sums on $q_1 \ldots q_j$ represent the combinations of initial source relations in the intermediate mappings with which it will infer inclusion dependencies. As explained above, each intermediate mapping will have as arity the sum of arities of the initial sources that were combined to obtain an intermediate mapping. The multiplication with 2 is due to the fact that each pair of attributes has two inclusion dependencies (i.e., each inclusion dependency has a mirroring one, as explained in Section 3.6.2).

Given that all sources are pair-wise mergeable, then each source relation is merged $C_{N-1}^{i-1}$ times in an iteration $i$, where $1 \leq i \leq N$ and $N$ is the number of input source relations. Intuitively, each time a source is involved in a merge, all inclusion dependencies of all its attributes are inferred to the newly-generated intermediate mapping.

Thus, it can be computed that, for an iteration $1 \leq i \leq N$ and $N$ input source relations, the number of inclusion dependencies that are inferred for a source attribute is given by the formula in Eq. 4.4,

$$C_{N-1}^{i-1} \cdot 2 \cdot \left( \sum_{j=1}^{i} \left( \sum_{q_1, q_1 \neq k}^{N} \cdots \sum_{q_j, q_j \neq k}^{N} (n_{q_1} + \cdots + n_{q_j}) \right) \right)$$ (4.4)

which leads to the number of inferred inclusion dependencies for a source relation.
with arity $n_k$ given by the formula in Eq.4.5,

$$n_k \cdot \left[ C_{N-1}^{i-1} \cdot 2 \cdot \left( \sum_{j=1}^{i} \left( \sum_{q_1,q_1 \neq k}^{N} \cdots \sum_{q_j,q_j \neq k}^{N} (n_{q_1} + \cdots + n_{q_j}) \right) \right) \right] \text{ (4.5)}$$

which leads to the number of inclusion dependencies that are inferred for all $N$ source relations given by the formula in Eq.4.6.

$$\sum_{k=1}^{N} \left[ n_k \cdot \left[ C_{N-1}^{i-1} \cdot \left( \sum_{j=1}^{i} \left( \sum_{q_1,q_1 \neq k}^{N} \cdots \sum_{q_j,q_j \neq k}^{N} (n_{q_1} + \cdots + n_{q_j}) \right) \right) \right] \text{ (4.6)}$$

Given the above, it can be concluded that for all $N$ iterations, the worst-case scenario for propagating all initial inclusion dependencies for all $N$ initial source relations can reach up to the number given by the formula in Eq.4.7, where $i$ represents the iterations ($1 \leq i \leq N$), $k$ and $q$ represent the index for each source.

$$\sum_{i=1}^{N} \sum_{k=1}^{N} \left[ n_k \cdot \left[ C_{N-1}^{i-1} \cdot \left( \sum_{j=1}^{i} \left( \sum_{q_1,q_1 \neq k}^{N} \cdots \sum_{q_j,q_j \neq k}^{N} (n_{q_1} + \cdots + n_{q_j}) \right) \right) \right] \right] \text{ (4.7)}$$

**Example 4.2.1.** For example, assume a simple scenario with 4 sources:

- $R_1(a_1, a_2, \ldots, a_{10})$
- $R_2(b_1, b_2, \ldots, b_{10})$
- $R_3(c_1, c_2, \ldots, c_{10})$
- $R_4(d_1, d_2, \ldots, d_{10})$, where, for simplicity, each has an arity of 10, i.e., $n_i = 10, i \in \{1, 2, 3, 4\}$. Also, all attributes in each relation overlap with all the other attributes in all relations. Considering the above setting, the initial number of inclusion dependencies which is detected is between each attribute of a relation $R_i, i \in \{1, 2, 3, 4\}$, with all attributes in all other relations $R_j, j \in \{1, 2, 3, 4\}$, provided that $i \neq j$.

**Initial inclusion dependencies (i.e., here iteration index is $i=1$).**

Using Eq. 4.3, the number of inclusion dependencies for

- one attribute in
- one mapping that involves
- one relation $R_k, k \in \{1, 2, 3, 4\}$ in
- one iteration ($i = 1$) is:
4.2. PROFILING DATA PROPAGATION COMPLEXITY

\[ 2 \cdot (\sum_{q_1=1,q_1 \neq k}^4 n_{q_1}) = 2 \cdot (10 + 10 + 10) = 60 \]

Using Eq. 4.4, the number of inclusion dependencies associated for one attribute in all mappings that involve one source relation \( R_k \) in one iteration \((i = 1)\) is:

\[ C_{4-1}^{1-1} \cdot 60 = 60 \quad \text{(as this is the first iteration so there are no merges)} \]

Using Eq. 4.5, the number of inclusion dependencies for all attributes for all mappings that involve one source relation \( R_k \) in one iteration \((i = 1)\) is:

\[ n_k \cdot 60 = 10 \cdot 60 = 600 \]

Using Eq. 4.6, the number of inclusion dependencies for all attributes in all mappings that involve all four relations in one iteration \((i = 1)\) is:

\[
\sum_{k=1}^{4} (n_k \cdot (C_{3-1}^{1-1} \cdot (\sum_{j=1}^{4} (\sum_{q_1=1,q_1 \neq k}^4 \sum_{q_j=1,q_j \neq k}^4 n_{q_1} + \cdots + n_{q_j})))) = \\
4 \cdot (10 \cdot (\sum_{q_1=1,q_1 \neq k}^4 n_{q_1}))) = 4 \cdot 10 \cdot (10 + 10 + 10) = 1200
\]

**Second iteration (i.e., iteration index is i=2).**

In this iteration, all sources are merged pairwise obtaining \((C_4^2 = 6)\) new intermediate mappings:

\[
m_{1,2} \leftarrow R_1 \bowtie R_2, m_{1,3} \leftarrow R_1 \bowtie R_3, m_{1,4} \leftarrow R_1 \bowtie R_4, \\
m_{2,3} \leftarrow R_2 \bowtie R_3, m_{2,4} \leftarrow R_2 \bowtie R_4, \\
m_{3,4} \leftarrow R_3 \bowtie R_4
\]

The inclusion dependencies inferred to the each of these mappings are propagated from the parent sources \( R_k, k \in \{1, 2, 3, 4\} \).

Using Eq. 4.3, the number of inclusion dependencies for one attribute in one mapping that involves one relation \( R_k, k \in \{1, 2, 3, 4\} \) in one iteration \((i = 2)\) is:

\[
2 \cdot (\sum_{j=1}^{2} (\sum_{q_1=1,q_1 \neq k}^4 \sum_{q_j=1,q_j \neq k}^4 n_{q_1} + \cdots + n_{q_j}))) = \\
\]
2 \cdot ((\sum_{q_1 \neq k}^4 n_{q_1} + (\sum_{q_1 \neq k}^4 \sum_{q_2 \neq k}^4 n_{q_1} + n_{q_2}))) =
2 \cdot (30 + (20 + 20 + 20)) = 120

Using Eq. 4.4, the number of inclusion dependencies associated for
one attribute in
all mappings that involve
one source relation \( R_k \) in
one iteration \((i = 2)\) is:
\( C_2^{4-1} \cdot 120 = 360 \)

Using Eq. 4.5, the number of inclusion dependencies for
all attributes for
all mappings that involve
one source relation \( R_k \) in
one iteration \((i = 2)\) is:
\( n_k \cdot 360 = 10 \cdot 360 = 3600 \)

Using Eq. 4.6, the number of inclusion dependencies for
all attributes in
all mappings that involve
all four relations in
one iteration \((i = 2)\) is:
\[ \sum_{k=1}^4 [n_k \cdot (C_4^{3-1} \cdot (\sum_{j=1}^2 (\sum_{q_1 \neq k}^4 \sum_{q_j \neq k}^4 n_{q_1} + \cdots + n_{q_j}))))] =
4 \cdot [10 \cdot 3 \cdot (\sum_{j=1}^2 (\sum_{q_1 \neq k}^4 \sum_{q_j \neq k}^4 n_{q_1} + \cdots + n_{q_j}))))] =
4 \cdot [10 \cdot 3 \cdot (\sum_{q_1 \neq k}^4 n_{q_1} + (\sum_{q_1 \neq k}^4 \sum_{q_2 \neq k}^4 n_{q_1} + n_{q_2}))))] =
120 \cdot (30 + (20 + 20 + 20)) = 120 \cdot 90 = 10800

Third iteration \((i.e., \ iteration \ index \ i=3)\)
In this iteration, all subsets of three sources are merged obtaining \( C_4^3 \) = 4
new intermediate mappings:
\( m_{12,3} \leftarrow m_{1,2} \bowtie R_3, m_{12,4} \leftarrow m_{1,2} \bowtie R_4, m_{13,4} \leftarrow m_{1,3} \bowtie R_4, m_{23,4} \leftarrow m_{2,3} \bowtie R_4 \)

Using Eq. 4.6, the number of inclusion dependencies for
all attributes in
all mappings that involve
all four relations in
one iteration \((i = 3)\) is:
\[ \sum_{k=1}^4 [n_k \cdot (C_4^{3-1} \cdot (\sum_{j=1}^3 (\sum_{q_1 \neq k}^4 \sum_{q_j \neq k}^4 n_{q_1} + \cdots + n_{q_j}))))] = \]
4.2. PROFILING DATA PROPAGATION COMPLEXITY

\[ 4 \cdot [10 \cdot (\sum_{q_1}^4 (\sum_{q_1,q_1\neq k}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + \cdots + n_{q_j}))))] = \]

\[ 4 \cdot [10 \cdot (\sum_{q_1}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + n_{q_2} + n_{q_3} + n_{q_4})))] = \]

\[ 120 \cdot (30 + 60 + 30) = 14400 \]

**Fourth/Final iteration (i.e., iteration index i=4)**

In this iteration, the set of four sources are merged obtaining \( C_4^4 = 1 \) new intermediate mapping:

\[ m_{123,4} \leftarrow m_{123} \bowtie R_4 \]

Using Eq. 4.6, the number of inclusion dependencies for

- all attributes in
- all mappings that involve
- all four relations in

one iteration \( (i = 4) \) is:

\[ \sum_{k=1}^4 [n_k \cdot (\sum_{j=1}^4 (\sum_{q_1,q_1\neq k}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + \cdots + n_{q_j}))))] = \]

\[ 4 \cdot [10 \cdot (\sum_{q_1}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + n_{q_2} + n_{q_3} + n_{q_4})))] = \]

\[ 4 \cdot [10 \cdot (30 + 60 + 30 + 0)] = 40 \cdot 120 = 4800 \]

To conclude, after all iterations, the total number of inclusion dependencies can be computed by summing up the totals from the four iterations:

\[ 1200 + 10800 + 14400 + 4800 = 31200 \]

The same number can be computed using Eq. 4.7, to compute the total number of inclusion dependencies for

- all attributes in
- all mappings that involve
- all four relations in

all iterations:

\[ \sum_{i=1}^N \sum_{k=1}^N [n_k \cdot (\sum_{j=1}^4 (\sum_{q_1,q_1\neq k}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + \cdots + n_{q_j}))))] = \]

\[ \sum_{i=1}^4 (n_k \cdot (\sum_{j=1}^4 (\sum_{q_1}^1 (\sum_{q_1,q_1\neq k}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + \cdots + n_{q_j})))))) = \]

\[ 4 \cdot (10 \cdot (1 \cdot (\sum_{q_1}^4 (n_{q_1} + n_{q_2} + n_{q_3} + n_{q_4})))) + \]

\[ 4 \cdot [10 \cdot [3 \cdot (\sum_{q_1,q_1\neq k}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + n_{q_2})))]] + \]

\[ 4 \cdot [10 \cdot [3 \cdot (\sum_{q_1,q_1\neq k}^4 (\sum_{q_1,q_1\neq k}^4 (n_{q_1} + n_{q_2})))]] + \]

\[ \cdots \]

\[ \cdots \]

\[ \cdots \]
4.3 Pruning Strategies

We aim to optimize the mapping generation algorithm presented in Chapter 3 by implementing a set of pruning strategies that:

1. discard intermediate mappings that do not promise to yield better results than other (to be) memoized mappings (Section 4.3.2); and
2. discard profiling data that is not predicted to be required for further merges (Section 4.3.3).

Fewer memoized mappings lead to a smaller search space to be explored. Less (useless) profiling data facilitates two aspects: (i) a faster search for merge operators (as it loops through less profiling data), and (ii) a faster propagation step as only potentially useful profiling data is inferred to the child mappings. The efficiency and impact of the proposed strategies is evaluated in Section 4.5.8.

4.3.1 Preliminaries

In this section we define the notions and notations to be used in formalizing the pruning strategies.

Let \( r, m_1, m_2 \in M \), where \( r \leftarrow \text{merged}(m_1, m_2, t) \), \( M \) is the space of mappings, and \( t \) a target relation.

Let \( \text{parents}(r) \leftarrow \{m_1, m_2\} \), where \( m_1 \) and \( m_2 \) are the two intermediate mappings that were merged to create \( r \).

Let \( \text{ancestors}(r) \leftarrow \{m_1, \ldots, m_n\} \), where \( m_i, i \in [1, n] \), are the initial input relations that lie behind the generation of \( r \).
4.3. **PRUNING STRATEGIES**

Let \( \text{operations}(r) \in \{ \text{union}, \text{join}, \text{mixed}, \text{base} \} \) characterise the types of operation that were used to build \( r \), e.g., if \( r \) was built by applying only union/join between its ancestor relations then \( \text{operations}(r) = \text{union} \) or \( \text{join} \), respectively; if \( r \) was built by applying both union and join operations between its ancestor relations then \( \text{operations}(r) = \text{mixed} \); if \( r \) is a base mapping then \( \text{operations}(r) = \text{base} \).

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

Let \( \text{join}_w(m_1, m_2, t) \), resp., \( \text{union}_w(m_1, m_2, t) \), be true if \( m_1 \) can join (resp., union) with \( m_2 \) w.r.t. target \( t \), and false otherwise. For two mappings \( m_1 \) and \( m_2 \) and a target relation \( t \), \( \text{join}_w(m_1, m_2, t) \) and \( \text{union}_w(m_1, m_2, t) \) cannot simultaneously be true, as only one operator is chosen by the algorithm to merge two mappings.

Let \( \delta(U \subseteq \theta V) \) 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 3.2 on rows 9 and 18, and in Table 3.3 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 \( \text{preserved}(r, m_1, t) \) 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. The conditions that need to be satisfied to qualify as a set of preserved mappings for child mapping \( r \), parent \( m_1 \), and target \( t \) are:

\[
\text{preserved}(r, m_1, t) = \{ n \mid n \in \text{mergeable}(m_1, t) \land \\
\quad n \in \text{mergeable}(r, t) \land n \notin \text{parents}(r) \land \\
\quad \delta(r.a_1 \subset n.a_2) = \delta(m_1.a_1 \subset n.a_2) \}
\]

The conditions in \( \text{preserved}(r, m_1, t) \) can be described as follows: given a mapping \( n \) with which the parent mapping \( m_1 \) has a merge opportunity \( (n \in \text{mergeable}(m_1, t)) \), then \( n \) may be a *preserved mapping* for the child mapping \( r \) if \( n \) has a merge opportunity with \( r \) \( (n \in \text{mergeable}(r, t)) \), as well. Also, \( n \) must **not** be a parent of \( r \) \( (n \notin \text{parents}(r)) \). After it is established that
CHAPTER 4. MAPPING GENERATION AT SCALE

$n$ is a mapping with which $r$ can merge, it is checked if the merge between them is 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. This is checked in the last condition, i.e., $\delta(r.a_1 \subset n.a_2) = \delta(m_1.a_1 \subset n.a_2)$. If the degree of degradation of the inferred $\text{ind} (\delta(r.a_1 \subset n.a_2))$ is equal with the degradation of the propagated $\text{ind} (\delta(m_1.a_1 \subset n.a_2))$, then the merge opportunity $m_1$ has with $n$ is preserved under similar conditions between $n$ and $r$, thus $n$ is a preserved mapping.

4.3.2 Pruning the Search Space

Motivation. In searching the space of candidate mappings, the sub-solutions produced by each call to GENERATE MAPPINGS (Algorithm 3) are memoized, so that they can be reused in subsequent calls. As a result, a collection of intermediate mappings is maintained, members of which are considered for merging during successive iterations. As discussed in Section 4.1, 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 intermediate mappings that compare well with related mappings.

Strategies. We define three major strategies for pruning the search space, which we describe in the following sections:

1. Removing unnecessary parent mappings
2. Preventing creation of superfluous mappings
   (a) Avoiding permutations
   (b) Avoiding subsumption
3. Pruning subsumed union mappings

4.3.2.1 Removing Unnecessary Parent Mappings

After a merge, parent mappings are discarded if the child mapping (i) has better fitness, and (ii) has the same merge opportunities as the parent. More formally, a parent mapping $m_1$ is discarded if:

\[
\text{mergeable}(m_1,t) \subset \{\text{preserved}(r,m_1,t) \cup \text{parents}(r)\} \land \text{fitness}(m_1) \leq \text{fitness}(r)
\]

By discarding parent mappings, the sets of sub-solutions from previous iterations become smaller, and the number of merge attempts decreases in subsequent iterations.
Example 4.3.1. Given the example in Figure 3.1, in iteration 2, after the base mappings for Manchester and Cambridge relations are merged, mapping $m_{2,1} \leftarrow MA \cup CA$ is created w.r.t. target $t$. Dynamap checks if the parents can be discarded. The conditions for removing unnecessary parent mappings are checked. First, Dynamap computes the following:

- $\text{parents}(m_{2,1}) = \{MA, CA\}$, $\text{fitness}(m_{2,1}) = 8$, $\text{fitness}(CA) = 4$
- $\text{mergeable}(CA, t) = \{UKD, UKQ, MA\}$
- $\text{mergeable}(m_{2,1}, t) = \{UKD, UKQ\}$
- $\text{preserved}(m_{2,1}, CA, t) = \{UKQ, UKD\}$

Now it checks if the same merge opportunities are preserved for $m_{2,1}$. This is true as

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

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

The inclusion dependencies between $m_{2,1}$ and $UKD$ are propagated using formulas in 2 and 11 in Table 3.2, and with $UKQ$ they are propagated using formulas 8 and 17 in the same table. These overlaps were not approximated, thus, the degradation did not increase when the inclusion dependencies were inferred. As both pruning conditions are met, it can be concluded that $CA$ can be discarded.

The same steps are followed for deciding whether $MA$ should be discarded as well.

4.3.2.2 Preventing Creation of Superfluous Mappings

This pruning technique exploits the associativity and commutativity of union and join. We divide this strategy into two different pruning methods: avoiding permutations and avoiding subsumption.

Avoiding permutations. Before a merge, the algorithm detects whether the mapping that would be generated is a superfluous variation of another mapping that has been memoized already.

Let $r$ be a memoized mapping, where $m_3$ and $m_4$ are the current candidates for merging. The merge is superfluous if:

- $r$ covers the same initial relations as $m_3$ and $m_4$:
  $$\text{ancestors}(r) = \text{ancestors}(m_3) \cup \text{ancestors}(m_4)$$
- $r$ contains only union or only join operations, and that same type of operation would be used to merge $m_3$ and $m_4$ w.r.t. target $t$: 
Example 4.3.2. Let us assume we extend the example in Figure 3.1 by adding another agency relation for Oxford which matches the same target attributes as Manchester and Cambridge sources, i.e., they all match street, postcode, and price. In this setting, all three relations are expected to union as they contain the same type of information needed in the target. Let us assume that we have the following mappings for iteration 2 (n.b., these are just a subset of the possible merges), where CA, MA and OX are base mappings:

\[
\begin{align*}
m_{2,1} & \leftarrow MA \cup CA \\
m_{2,2} & \leftarrow CA \cup OX
\end{align*}
\]

In iteration 3, let us assume that mapping \( m_{3,1} \leftarrow m_{2,1} \cup OX \) has already been memoized. The algorithm tries to merge \( m_{2,2} \) with \( MA \) and detects that these should be merged through union. It checks if their merge result is a superfluous mapping using the pruning strategy conditions. Dynamap

- computes the ancestors:
  \[
  \begin{align*}
  \text{ancestors}(m_{3,1}) & \leftarrow \{MA, CA, OX\} \\
  \text{ancestors}(m_{2,2}) & \leftarrow \{CA, OX\} \\
  \text{ancestors}(MA) & \leftarrow \{MA\}
  \end{align*}
  \]

- checks first pruning condition:
  \[
  \text{check if } \text{ancestors}(m_{3,1}) = \text{ancestors}(m_{2,2}) \cup \text{ancestors}(MA), \text{ and it is true.}
  \]

- computes the operations values:
  \[
  \begin{align*}
  \text{operations}(m_{3,1}) & \leftarrow \text{union} \\
  \text{operations}(m_{2,2}) & \leftarrow \text{union} \\
  \text{operations}(MA) & \leftarrow \text{base}
  \end{align*}
  \]

- checks second pruning condition, i.e., check if:
  \[
  \begin{align*}
  (\text{operations}(m_{3,1}) = \text{operations}(m_{2,2}) \lor \text{operations}(m_{2,2}) = \text{base}) \land \\
  (\text{operations}(m_{3,1}) = \text{operations}(MA) \lor \text{operations}(MA) = \text{base}) \land \\
  (\text{operations}(m_{3,1}) = \text{join} \land \text{join\_with}(m_{2,2}, MA, t)) \lor
  \end{align*}
  \]
4.3. PRUNING STRATEGIES

\[(\text{operations}(m_{3,1}) = \text{union} \land \text{union}_\text{with}(m_{2,2}, MA, t))\]

The condition is satisfied as the expected operation between \(m_{2,2}\) and \(MA\) is a union, and it seems to be just another permutation of the same mapping expressed by \(m_{3,1}\) as \(m_{3,1}\) is a union dominant mapping that unions all three relations, but in a different order.

**Avoiding subsumption.** Building on the same properties, viz., associativity and commutativity, we also prevent the generation of join or union mappings that would otherwise be redundantly generated in subsequent iterations, as well as of 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 between \(m_1\) and \(m_2\) is superfluous if either of the following conditions is satisfied:

1. 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 \(m_3\), where \(m_3\) is also joinable with \(m_1\) w.r.t. \(t\):
   \[\text{join}_\text{with}(m_1, m_2, t) \land \text{join}_\text{with}(m_1, m_3, t) \land \text{union}_\text{with}(m_2, m_3, t)\]
2. the candidate union mapping between \(m_1\) and \(m_2\) would become subsumed by a future mapping containing only union operations:
   \[\text{parents}(r) = \{m_2, m_3\} \land \text{operations}(r) = \text{union} \land \text{union}_\text{with}(r, m_1) \land \text{union}_\text{with}(m_1, m_2)\]

Both conditions prove to be useful in union-heavy scenarios where many union mappings are equivalent as, essentially, they are permutations of the same unioned initial relations.

**Example 4.3.3.** Let us assume the same running example in Figure 3.1. With the avoiding subsumption strategy, the algorithm avoids the creation of two mappings that are subsumed by another:

In our scenario, for merging \(MA, CA\) and \(UKD\), there are two options:

- \(m_{3,1} \leftarrow (MA \cup CA) \bowtie_{\text{postcode}} UKD\)
- \(m_{3,2} \leftarrow (MA \bowtie_{\text{postcode}} UKD) \cup (CA \bowtie_{\text{postcode}} UKD)\)

The two mappings are equivalent, but only one mapping is reached by Dynamap, as \(m_{3,2}\) does not satisfy the avoiding common ancestors condition (see Algorithm 4) – which keeps the search space contained.

In a scenario where it is desirable to merge complex mappings (which merge many sources), generating mappings \((MA \bowtie_{\text{postcode}} UKD)\) and \((CA \bowtie_{\text{postcode}} UKD)\)
UKD) in iteration 2 is redundant as their results are included in \( m_{3,1} \) in iteration 3, and \( m_{3,2} \) is never reached in iteration 4 (as explained above). Thus, the separate join of UKD with MA and CA in iteration 2 can be avoided as the same result can be produced in iteration 3.

In this example, the pruning condition that is satisfied is the first one in avoiding subsumption, i.e., the following condition is true when trying to merge UKD with MA (or CA):

\[
\text{join} \_\text{with}(UKD, MA, t) \land \text{join} \_\text{with}(UKD, CA, t) \land \text{union} \_\text{with}(MA, CA, t).
\]

### 4.3.2.3 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, no instance data is lost through the merge, however, the algorithm does not check whether the child has the same merge opportunities so, in practice, this strategy is useful if the algorithm is meant to explore and yield some mappings, but not necessarily all possible mappings as some parent mappings are discarded only based on the subsumption of their instances.

A 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 \( r \): \( \text{ancestors}(m) \subseteq \text{ancestors}(r) \),
- and both \( m \) and \( r \) were created using only union operations: \( \text{operations}(m) = \text{union} \land \text{operations}(r) = \text{union} \).

**Example 4.3.4.** Assuming the example in Figure 3.1, after the merge of \( r \leftarrow MA \cup CA \), for detecting whether to remove \( MA \) (or \( CA \)) from the search space, Dynamap:
- computes the ancestors:
  \[
  \text{ancestors}(r) \leftarrow \{MA, CA\},
  \text{ancestors}(MA) \leftarrow \{MA\},
  \]
- checks if: \( \text{ancestors}(MA) \subseteq \text{ancestors}(r) \), which is true,
- computes the operations values:
  \[
  \text{operations}(r) \leftarrow \text{union},
  \text{operations}(MA) \leftarrow \text{base}
  \]
- checks if: \( \text{operations}(r) = \text{union} \land \text{operations}(MA) = \text{union} \), which is false as \( \text{operations}(MA) \leftarrow \text{base} \).

Given that the last condition is false, the \( MA \) (base) mapping is not dropped. The same steps apply for \( CA \), which is not dropped either.

The intuition behind keeping the base mappings is that these are the starting point of the mapping generation, thus, they are used in all iterations to try to compute the next sub-solutions, so eliminating them may lead to missed merge opportunities.

Now, assume an extended example with the addition of the Oxford real-estate agency, and that the mapping \( m_{2,1} \leftarrow MA \cup CA \) has been memoized and the algorithm is at the step where it computed \( r \leftarrow m_{2,1} \cup OX \). The same steps are taken as before, however, now \( \text{operations}(m_{2,1}) \leftarrow \text{union} \), thus, the last condition is satisfied when deciding whether to discard \( m_{2,1} \). Since \( m_{2,1} \) is subsumed by \( r \), Dynamap discards it.

### 4.3.3 Pruning the Profiling Data

**Motivation.** By removing inclusion dependencies, the complexity of finding a merge operator decreases as there are fewer of them to be considered for the join conditions. Also, pruning the profile data results in reduced runtime for propagating profiling data to child mappings from their parents’ corresponding profiling data.

**Strategies.** Let \( I \) be an inclusion dependency of the form \( I \leftarrow m_1.S \subseteq_{\theta} m_2.P \) where \( m_1 \) and \( m_2 \) are mappings and \( S \) and \( P \) are their attributes. \( I \) can be discarded from the pool of profile data by any of the pruning strategies. We define three strategies for pruning the profiling data, which are described in the next sections:

1. Discarding idle inclusion dependencies
2. Discarding same-mapping inclusion dependencies
3. Minimum threshold overlap

### 4.3.3.1 Discarding Idle Inclusion Dependencies

As shown in Section 3.3, only full outer join and join operators depend on the existence of inclusion dependencies to merge two mappings. However, a candidate key is also needed to determine whether a merge is possible, thus, if an inclusion
dependency $I$ does not involve candidate keys can be safely discarded as it cannot be used in a merge:

Discard $I$ if:

\[ \theta > 0 \land \left((m_1.S \text{ not a candidate key in } m_1) \land (m_2.P \text{ not a candidate key in } m_2)\right) \]

### 4.3.3.2 Discarding Same-mapping Inclusion Dependencies

Given that inclusion dependencies are only used in deciding whether to join two mappings or not, having inclusion dependencies between the attributes of the same mapping is useless as these are not used. This setting may arise when a mapping is the result of two other mappings merging with shared inclusion dependencies, thus, the newly inferred inclusion dependency $I$ is between attributes of the same mapping:

Discard $I$ if $m_1 = m_2$.

### 4.3.3.3 Minimum Threshold Overlap

An inclusion dependency is discarded if both its overlap and the overlap of its mirroring inclusion dependency are below a set minimum threshold. This type of strategy steers the mapping generation in the sense of merging mappings only on attributes which have a significant overlap between their values, thus, obtaining a high fraction of correlated tuples. However, this type of pruning strategy can lead to missed opportunities if the user decides to set a high minimum. Varying the minimum threshold leads to a trade-off between run-time and explored mapping combinations, as a higher threshold can eliminate merge opportunities. In this way, a final mapping with a high correlation is returned sooner, but, if the sources are from different domains, their attributes may not have high overlaps, and too-high a threshold can lead to not having any generated mappings at all (excluding the base mappings created in the preprocessing step).

If only one of the inclusion dependencies is below the threshold, then both inclusion dependencies are kept as it could be the case that one source contains a large amount of information and it includes the other source which contains only a small subset of the large one, thus, a merge between the two would be meaningful.

Let $\theta_{\text{min}}$ be the set minimum threshold and $I'$ be the mirroring inclusion dependency of $I$, $I' \leftarrow m_2.P \subset_{\theta'} m_1.S$. 
Discard $I$ and $I'$ if $\theta' < \theta_{\text{min}} \land \theta < \theta_{\text{min}}$.

### 4.3.4 Pruning Strategies in Dynamap

Figure 3.4 depicts the workflow of the mapping generation algorithm as it was described in Chapter 3, i.e., without the pruning strategies. In Figure 4.1 we show the workflow of Dynamap with the pruning strategies added to the previous algorithm. We describe below how the pruning strategies are integrated to the previously described mapping generation algorithm.

**Integration into Dynamap.** The input remains unchanged, while the output may change in the light of the pruning component, i.e., output mappings may be discarded by the pruning strategies. The preprocessing step is done as before with the purpose of reading/generating profile data and creating the base mappings to bootstrap the dynamic programming recursive method.

After the preprocessing step finishes, the first step in the core of the algorithm is to filter the (initial) profiling data and keep the data that satisfies the pruning strategies described in Section 4.3.3. After pruning the profiling data, the mapping generation process calls, as before, `GenerateMappings` for each iteration. In the process of merging two mappings (i.e., within `MergeMappings`, where mappings from two sub-solution batches are pairwise merged), before trying to choose an operator, the algorithm decides whether to search for an operator or else postpone (or even avoid) the merge. The pruning strategy that this decision step implements consists of the conditions described in *Preventing creation of superfluous mappings*, in Section 4.3.2. If the two intermediate mappings are not to be merged at that step, then the algorithm returns to the main method and moves on to the next pair of mappings. Otherwise, mapping generation proceeds as in the previous version (steps 5-10 in Figure 4.1).

After an iteration finishes generating candidate mappings, the newly created mappings are filtered by discarding either *unnecessary parent mappings* or *union-subsumed mappings*, as described, in Section 4.3.2, in *Removing unnecessary parent mappings* and *Pruning subsumed union mappings*, respectively. After the mappings (if any) are discarded, the algorithm proceeds to pruning profiling data. The newly created intermediate mappings can include candidates for pruning if they are not useful in subsequent merges. Once the profiling data is filtered, Dynamap moves on to the next iteration.
4.4 Integration Scenarios

In this section, we briefly describe work on scenario generators with a view to evaluating integration systems, including mapping generation. We then proceed to discuss the need for an extension to these generators such that the created scenarios can simulate *mapping generation in the wild* as existing generators tend to take into consideration settings where there is a single well-defined (potentially complex) source schema and a multi-relation target schema (often with constraints).

4.4.1 Integration Generators

4.4.1.1 STBenchmark

**STBenchmark** [Alexe et al. (2008)] is a generator of integration scenarios that provides various mapping scenarios with nested sources. For each scenario, it proposes one or more sample source instances, and the corresponding ground-truth solution. They provide 11 basic scenarios including copying, horizontal and vertical partitioning, object fusion, identifier generation, normalization and denormalization, flattening and nesting, and constant value assignment. For each source, STBenchmark provides instance data that is extracted from real-world data. STBenchmark can generate more complex scenarios where a set of input parameters are expected such that they vary the level of nesting in the schemas, the number of subelements of each schema element, the number of elements involved in a join between two elements, the length of the join paths formed in the schemas, the kind of joins (star or chain), etc. Complex scenarios are built by combining the basic scenarios. All the generated mapping scenarios comprise a source schema with its metadata, and a target schema exhibiting one (for basic
scenarios) or more (for complex scenarios) data transformation patterns. Note that STBenchmark-generated scenarios do not cater for key constraints on the target schema.

4.4.1.2 iBench

iBench [Arocena et al. (2015)] is a tool that generates data integration/exchange scenarios, where the sources have explicit keys and foreign keys. Similarly to STBenchmark, these scenarios 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. As described in Section 3.8.1, the type of merge involves a variation of copying and/or joining source relations to populate the target. The input parameters vary the number of source attributes; the number of added/deleted target attributes w.r.t. the matching sources; the percent of reused generated source/target relations, i.e., for complex scenarios the primitives can reuse already generated relations instead of generating new ones; whether the generated scenarios are to be built on a stub scenario etc. In the cases where the parameters are contradictory, e.g., reuse the target relations but the primitives cannot be accommodated by the already generated relations, then iBench relaxes the constraints by prioritizing the creation of the sources as needed, and adapting the target schema such that the primitive is correctly created.

The generated scenarios include one source schema and one target schema together with their metadata which comprises key and foreign key constraints, and matches (S-T tgds). iBench is publicly available and it proposes data generation using ToXgene [Barbosa et al. (2002)].

4.4.1.3 ToXgene

ToXgene [Barbosa et al. (2002)] is a generator for synthetic XML data based on a given template. The generator takes as input a template that comprises the structure of the XML document to be generated. More precisely, it contains annotations that describe the structure and the data values to be generated, i.e., vocabularies and ranges used for generating values, and distributions used for sampling of the values. The template specification language is similar to an XML schema in the sense that it is extended with concepts that allow defining
certain characteristics and constraints on the desired generated target. This type of generator is useful for generating synthetic XML data. Its focus is not on generating integration scenarios although their mechanisms for generating data can prove useful in scenarios where one wants to create instance data for specific types of merges, however, this implies that there are no automatically generated transformations, i.e., ground-truth mappings, so the user needs to steer the generation of the data (through the template annotations) based on the corresponding metadata characteristics, e.g., iBench proposed using it together with the metadata generator they propose in Arocena et al. (2015).

The output of the generator is an XML document with synthetic data values.

### 4.4.1.4 Thalia

**Thalia** [Hammer et al. (2005)] provides test scenarios along with data instances extracted from real-world applications. The source information comes from 25 data sources from the academic domain, e.g., course catalogs, which have a corresponding set of 12 queries, and a cost function for ranking the performance of an integration system. The 12 queries are each between a *reference schema* (source) and a *challenge schema* (target), thus, each query can be used as a ground-truth mapping to transform the data from the *reference* in the format of the *challenge schema*. This benchmark can prove useful in evaluating several types of integration systems, not only for mapping generation, as the queries tackle different integration challenges, e.g., evaluating a matches generator, detecting data transformations, etc.

### 4.4.1.5 Amalgam

**Amalgam** [Miller et al. (2001)] is a test suit for schema integration comprising of several databases with bibliographic information, but with different schemas and the data extracted from four different sources. Given that the schemas are from the same domain, but with slight information variations, similarly to Thalia, Amalgam can be used for evaluating several data integration aspects such as matching and mapping generation on real-world data.
4.4. INTEGRATION SCENARIOS

4.4.2 Synthe grate

In this chapter, we have extended the mapping generation algorithm with pruning strategies for the purposes of scalability, i.e., so that it can take as input large numbers of sources. We need, therefore, to evaluate the impact of specific properties of the integration scenarios on the runtime performance of Dynamap (Section 4.5). To this end, we developed Synthegrate\(^1\), a generator of integration scenarios that provides control over the scale and composition of mappings in generated scenarios.

Motivation. The motivation behind developing Synthegrate is that we could not find a benchmark/generator to suit our requirements for evaluation settings with stress tests, i.e., evaluating at scale a mapping generator that does not rely on explicit constraints and that needs to infer join paths to merge the autonomous sources, all the while keeping the target schema fixed to one relation, as this is the setting under which Dynamap works (although it is extended in Chapter 5 to multiple-relation schemas).

None of the benchmarks/generators described in Section 4.4.1 are very useful. Thalia and ST-Benchmark are no longer available, however iBench is considered to be the successor of ST-Benchmark as a specification generator [Benedikt et al. (2017)]. 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. Also, given that Dynamap tackles the problem of merging independent, heterogeneous source relations, Synthegrate can create separate schemas and provides automatically generated profile data that iBench is not intended to produce. ToXgene does not necessarily address the problem of generating scenarios for evaluating mapping generation as it does not create scenarios where source data needs to be transformed into target data, and so it is not generating a ground-truth mapping with which to compare the generated mappings. Thalia and Amalgam are two benchmarks that could have been used to evaluate Dynamap, but not at scale. However, the scenarios they propose are rather limited in number and the merge characteristics exhibited by the defined integration scenarios (suitable for evaluating mapping generation) are more specific cases of

\(^1\)https://github.com/MLacra/Synthegrate.git
what iBench scenarios can describe. We use iBench-generated scenarios wherever feasible.

**Syntheegrate.** We developed **Syntheegrate** as 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 profiling data are created automatically by **Syntheegrate**, reflecting the database schemas and the extents (which are generated using Datafiller [Coelho (2013)]).

**Syntheegrate** uses a top-down approach that starts with the creation of the target and then uses the target table(s) to create the source table(s) – which, at the end, are grouped in one or more source schemas. The target tables are then populated using synthetic data. Using the target tables as starting point for creating the source tables ensures that the generated ground-truth mapping, when executed on the source tables, recreates the same tuples as in the target table(s).

Appendix B provides a detailed description of the design decisions behind **Syntheegrate** and how the integration scenarios are built.

### 4.5 Scalability Evaluation

In this section, we evaluate the performance of Dynamap on two types of synthetic scenario and two real-world scenarios\(^1\). The experimental results show how Dynamap performs at large scale, and also the effect of the pruning strategies, as follows:

1. In Section 4.5.1, Dynamap is compared against the current state-of-the-art mapping generation algorithm (++Spicy [Marnette et al. (2010)]) on **standard** scenarios generated by a state-of-the-art integration benchmark (iBench [Arocena et al. (2015)]). **Standard** scenarios involve one single source and one target schema with explicit and correct schema constraints and matches as might be produced by a data scientist working with a tool such as ++Spicy or Clio.

\(^1\)The details of the experiments, including the data sets, are available from: https://github.com/MLacra/mapping_generation_experiments
2. In Sections 4.5.2 and 4.5.3, we compare Dynamap with ++Spicy on two real-case examples where the candidate keys and inter-table relationships are inferred using profiling data.

3. In Section 4.5.4 we evaluate the accuracy of the propagation rules described in Section 3.6.

4. In Sections 4.5.5, 4.5.6, 4.5.7, we run Dynamap under a set of synthetic scenarios (generated by SYNTHEGRATE) that are meant to merge multiple sources through a variety of combinations of two relational operators, i.e., union and join.

5. In Section 4.5.8, we measure the impact of the pruning strategies described in Section 4.3 by separately running Dynamap over the same scenarios with different pruning strategies activated or deactivated at a time.

Comparison with ++Spicy. In both real-world and synthetic scenarios, Dynamap is compared to the state-of-the-art mapping generation algorithm, i.e., ++Spicy [Marnette et al. (2010)]\(^1\). ++Spicy creates mappings and transformations that attempt to produce core solutions given as input a set of matches and schema constraints, i.e., primary keys, foreign keys (as described in Section 2.1.4).

In this section, comparisons are drawn with ++Spicy in terms of result quality and runtime performance. ++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, but 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.

Experimental setup. Dynamap and ++Spicy were run over the same data sources, and the same target schemas. For storage, we used PostgreSQL 9.6. In the case of the real-world scenarios, in order to maintain a focus on mapping generation, matches were generated by a human expert. The profiling data was generated through two Metanome modules, i.e., HyUCC [Papenbrock and Nau mann (2017)] for generating candidate keys and Sindy [Kruse et al. (2015)] for

\(^1\)http://www.db.unibas.it/projects/spicy/
generating (partial) inclusion dependencies. Given that ++Spicy uses explicit schema constraints, based on the profiling data, foreign keys are inferred where possible, i.e., if a candidate key shares a (full) inclusion dependency with an attribute from another relation then a foreign key is inferred. In the case of the synthetic scenarios, the matches, the profiling data, the data sources, and the target schema are generated automatically (without human input) by a scenarios generator. Tuples in the synthetic scenarios were generated using Datafiller [Coelho (2013)]. The experiments were run over an Intel Core i5 with 2x2.7 GHz, and 8 GB of RAM. We report the results over the average of 10 runs when runtime is measured.

Data quality evaluation metrics. Given that both algorithms are run over complex scenarios that may contain data that can be combined in multiple correct ways, we decided not to compare the generated mappings, but to compare the output tuples of the mappings, executing the mappings over the same sources.

iBench scenarios. For the iBench scenarios we used metrics from Arocena et al. (2015), viz., the number of constants and the number of nulls produced by the mappings. In Arocena et al. (2015), mappings that output fewer constants and fewer nulls are considered to be desirable. The intuition behind this metric is that if the data is correlated as best as possible, then there are no (or few) redundant tuples, while mappings that do not correlate (i.e., join) tables will create many duplicate values and undesirable nulls. Thus, if the number of constants and nulls is minimal then it means that the data has been correlated.

Real case scenarios. There is no widely-accepted way of measuring the quality of the output tuples against a ground truth. For example, Mecca et al. (2012) suggest that the output of a transformation system can be analyzed based on the number of identical tuples that are found in the output tuples, thus, assigning a quality score based on the number of fully correct tuples. However, in our experiments, we wanted to take into consideration both fully and partially correct tuples, thus, our chosen metrics show results at a finer-grained level, i.e., based on the overall number of correct values at attribute level, and the number of correct values at tuple level. The output tuples and the ground-truth tuples are correlated for comparison based on their values on key attributes.

Attribute level. 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 (TP) is a correct non-null value in the output of the compared
4.5. SCALABILITY EVALUATION

mapping.
- A true negative (TN) is a correctly output null, i.e., it was expected to be null in the ground truth.
- A false negative (FN) is a missing value (a null) in the output of the compared mapping.
- A false positive (FP) is an non-null incorrect value in the output of the compared mapping.

Tuple level. Given that a tuple typically has several attribute values, the correctness of the tuple is computed based on the dominant correctness label among the labels for its attribute values.
- A true positive is an output tuple where all or the majority of its attribute values are correct, i.e., a TP or a TN at the attribute level.
- A false positive is an output tuple where all or the majority of its attribute values are not nulls, but incorrect, i.e., not as expected in the ground truth.
- A false negative is an output tuple where all or the majority of its values are missing, i.e., are nulls.
- True negatives – We do not measure the true negative tuples as these would represent the number of correctly eliminated tuples and this is not the focus of the evaluation.

Level of completeness of tuples. The correctness of a tuple is determined by using the dominant correctness in its attribute values, so we say that a tuple is complete if all its attribute values have the same type of correctness label, and incomplete if they are mixed. In Figures 4.3(b) and 4.4(b), the number at the top of each bar represents the number of incomplete tuples, while the number below it represents the number of complete tuples, e.g., a complete true positive tuple has all the attribute values that the ground truth tuple has, and an incomplete true positive tuple is one in which the majority (but not all) of the values were as expected in the ground truth.

Mapping selection. In Dynamap, many plausible candidate mappings may be produced, in different iterations. As such, there is a need to select a subset of these mappings. In practice, this could involve declared user preferences [Konstantinou et al. (2019)]. Dynamap uses a fitness function. In these experiments, mappings were selected by choosing, from the fittest $k$ mappings (Section 3.7), as few mappings as possible such that all initial relations are involved. The mappings are selected by applying the set-cover method [Aho and Hopcroft (1974)] to the
subsets of initial relations merged in each mapping. We chose to have large complex mappings that involve all source relations so that all input data is transferred to the target, which makes it simpler to observe the quality of the output tuples (originating from all sources) generated by the mappings. 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 [Abel et al. (2018)].

**Runtime evaluation metrics.** 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 Mecca et al. (2012), 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.

### 4.5.1 Benchmark Experiment - iBench

The iBench experiments follow the methodology presented in Arocena et al. (2015), where iBench is used to compare several mapping generation algorithms. The measures proposed in Arocena et al. (2015) imply that the mappings that produce smaller target instances produce less incompleteness, so they measure the size of the target which consists in the number of atoms [Alexe et al. (2012)] that could be either a constant or a null.

**Scenarios.** The scenarios are built as follows.

*Target schema.* 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 [Coelho (2013)]. 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 Sections 4.5.5, 4.5.6, 4.5.7).
4.5. SCALABILITY EVALUATION

We used the following iBench primitives: \textit{add attribute}, \textit{add-delete}, \textit{delete}, \textit{copy} and \textit{merge-add}. As explained in Section 3.8.1, the only type of primitive that Dynamap is not able to tackle as expected in this version, is when a source relation is split into two target relations that can be joined through a foreign key. In this case, Dynamap generates two mappings that populate the two target relations since, as mentioned above, the foreign key relationships in the target are not used in Dynamap. This type of scenario is revisited in Chapter 5, where we tackle scenarios in which the target schema has constraints over multiple tables.

In generating the different scenarios, we varied the number of primitives so that 20 input source relations are created that 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) and 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.

\textit{Matches}. All sources will match the target; the matches are generated by iBench according to the primitives.

\textit{Profiling data}. The profiling data is generated according to the inclusion dependencies in each scenario and the defined primary keys in each relation. We set the iBench parameters so as to vary the number of added and/or deleted attributes, to reuse the target schema 100%.
Results. The results are shown in Figure 4.2, 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 \textit{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 less 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 atom values, all output values of Dynamap were identical with the corresponding tuple values of ++Spicy.

4.5.2 Real-world Experiment - \textit{Real-estate} Domain

Motivation. In this section we investigate how Dynamap performs on 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. For the data extraction we used OXPath [Furche et al. (2012)], following the representation of the data on the web page. Additionally, open-government datasets were included in the scenario, i.e., the indices of deprivation dataset that measures the relative deprivation in small areas in England, and freely available open-government addresses data.

Scenario. The purpose of this scenario is to generate a mapping that associates crime statistics with the properties information from the real-estate agencies.

\textit{Target schema}. The target schema is a single table:

\textbf{Target} (\texttt{postcode, city, street, price, crimerank})

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data for the target</th>
<th>#Sources</th>
<th>Arity</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manchester real-estate</td>
<td>street, price, city, postcode</td>
<td>5</td>
<td>5-9</td>
<td>20-171</td>
</tr>
<tr>
<td>London real-estate</td>
<td>street, price, city, postcode</td>
<td>2</td>
<td>6-13</td>
<td>20-35</td>
</tr>
<tr>
<td>Oxford real-estate</td>
<td>price, postcode, street</td>
<td>4</td>
<td>10-14</td>
<td>28-152</td>
</tr>
<tr>
<td>Manchester deprivation</td>
<td>postcode, crimerank</td>
<td>1</td>
<td>28</td>
<td>391</td>
</tr>
<tr>
<td>London deprivation</td>
<td>postcode, crimerank</td>
<td>1</td>
<td>28</td>
<td>54</td>
</tr>
<tr>
<td>Manchester &amp; Oxford addresses</td>
<td>postcode, street, city</td>
<td>1</td>
<td>4</td>
<td>235</td>
</tr>
</tbody>
</table>

Table 4.1: Web-extracted and open government datasets
4.5. SCALABILITY EVALUATION

*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 4.1 where the first column states what the dataset is about, the second lists which attributes from the dataset contribute to populating the target (not necessarily all the available attributes in the source), the third column is the number of input data sources that contain that type of data, the fourth is the arity range, and the fifth the cardinality range. It can be observed that the real-estate datasets comprise separate information from three UK areas: Oxford, Manchester, and London, but the information about deprivation is from Manchester and London areas only, while the addresses are from Manchester and Oxford only. One can see that not all datasets should be merged with one another, e.g., London deprivation with Oxford real-estate, while others could merge with more than one, e.g., addresses could merge with both Oxford and Manchester properties.

*Profiling data.* To obtain the profiling data on the input sources, HyUCC [Papenbrock and Naumann (2017)] was run to detect the candidate keys, and SINDY [Kruse et al. (2015)] was run to obtain the (partial) inclusion dependencies. The input profiling data contains: 68 candidate keys, 1734 partial inclusion dependencies, and 510 full inclusion dependencies. Given that ++Spicy takes as input only one schema and explicit schema constraints, all sources were transferred to a common PostgreSQL schema and, based on the profiling data, a set of 68 unique constraints was added to the schema, and 7 foreign keys could be inferred considering the standard foreign key conditions (a candidate key is referenced by an attribute, and the values of the attribute need to be fully contained in the values of the candidate key).

*Ground truth.* The ground truth mapping was created by hand. The mapping unions all four Oxford properties datasets and then (outer)joins them with addresses. The result becomes schema compatible w.r.t. to the target matches with Manchester real-estate so they are unioned, and then all information from Manchester and Oxford is (outer)joined with Manchester deprivation. We chose not to join Manchester real-estate with addresses because all the information covered by addresses is already found in Manchester real-estate, while Oxford needed information about city and this was found in addresses. Also, we chose to merge Manchester deprivation after the union between Manchester and Oxford properties because by merging Oxford properties with addresses, their merge result will contain information from both Manchester and Oxford. In the end,
information about London real-estate merged with London deprivation is unioned with the information from Manchester and Oxford.

Comparison. 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 mapping.

Results. The results of the two mappings against the ground truth mapping can be seen in Figures 4.3(a) and 4.3(b).

Attribute level. The results at attribute level are shown in Figure 4.3(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 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 4.3(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 disjoint 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).

One can observe that, overall, although Dynamap does not generate the exact
4.5. SCALABILITY EVALUATION

(a) Attribute level

(b) Tuple level

Figure 4.3: Performance of Dynamap and ++Spicy on a real-estate scenario

Ground truth mapping, the output tuples show that the mapping is a variation of the ground truth, thus, Dynamap produced better results than ++Spicy. This is due to the fact that Dynamap combined as much data as possible by performing full outer join operations where foreign key constraints could not be inferred, i.e., partial inclusion dependencies were taken into consideration for correlating data that overlapped in the sources. In comparison, ++Spicy managed to combine only the datasets that were linked through explicit foreign key constraints or where egds could be used to remove redundancy (as explained in Section 2.1.4).

In the case of each algorithm, the method for producing the output is as follows: ++Spicy materializes 30 relations in the database, while Dynamap produces a mapping that can run directly on the input sources, without materializing intermediate relations, thus, the mapping that Dynamap produces is easier and more efficient to execute.

4.5.3 Real-world Experiment - Schools Domain

In this experiment we have used open-government data from data.gov.uk, with a particular focus on heterogeneous and disjoint sources that could be correlated to populate a target schema.

Motivation. As in Section 4.5.2, this type of evaluation investigates the extent to which our proposed technique can tackle mapping generation cases where the sources are not well behaved, i.e., do not come from the same database/schema where tables are explicitly connected to one another through foreign keys.

Scenario. The data sources contain information about schools, more specifically, about the facilities in those schools. The sources are outlined in Table 4.2.
### Table 4.2: Input source files - schools information

<table>
<thead>
<tr>
<th>Data.gov.uk source</th>
<th>Data for the target</th>
<th>#Sources</th>
<th>Arity</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>All schools</td>
<td>dfe code, school name, headteacher</td>
<td>1</td>
<td>16</td>
<td>99</td>
</tr>
<tr>
<td>Free meals eligibility</td>
<td>dfe code, school name, #eligible students</td>
<td>1</td>
<td>4</td>
<td>85</td>
</tr>
<tr>
<td>Additional languages</td>
<td>dfe code, school name, students with English as additional language</td>
<td>6</td>
<td>3-6</td>
<td>88</td>
</tr>
<tr>
<td>Defibrillators equipment</td>
<td>school name,#defibrillators</td>
<td>1</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>Road and Safety training</td>
<td>school name, school type</td>
<td>1</td>
<td>3</td>
<td>46</td>
</tr>
<tr>
<td>Bikeability courses</td>
<td>school name, #courses</td>
<td>1</td>
<td>6</td>
<td>87</td>
</tr>
</tbody>
</table>

**Target schema.** The target schema brings together the information about each school with the information about its activities and facilities, i.e., Bikeability, Road and Safety training, English courses, Free meals programs, and Defibrillators. Given this example, the target relation contains information to be gathered from all input sources:

```
Target(dfe code, school name, school type, headteacher contact, no. bikeability courses, no. students with English as additional lang., no. defibrillators, no. students eligible for free meals).
```

**Input sources.** Part of the input sources were used in the experiment in Section 3.8.2. We extended that set with an additional 7 sources that contain open-government data about schools. Details about the input sources are found in Table 4.2. It can be observed that the datasets contain different kinds of information about the schools, thus, they should be merged with one another.

**Profiling data.** The same method as in Section 4.5.2 was used to obtain the profiling data. The input contains 48 candidate keys, 681 partial inclusion dependencies, and 47 full inclusion dependencies. A set of 48 unique constraints was added to the schema, and 5 foreign keys were inferred. Although explicit schema constraints were inferred, in this scenario, the way of merging the input sources is less obvious than for the scenario in Section 4.5.2.

**Ground truth.** The ground truth mapping was created by hand, as follows. First, the datasets with data about additional languages were unioned, as they contain the same type of information, and then, the results were merged with the remaining sources through sequential full outer join operations.

**Comparison.** After running Dynamap and ++Spicy over the same mapping task, the output mappings are executed and the output attributes and tuples are compared to the ground-truth output.

**Results.** The results of the two mappings against the ground truth can be seen in Figures 4.4(a) and 4.4(b), for attribute-level and tuple-level, respectively.
4.5. SCALABILITY EVALUATION

(a) Attribute level

(b) Tuple level

Figure 4.4: Performance of Dynamap and ++Spicy on a schools scenario

Attribute level. The results at attribute level are depicted by Figure 4.4(a). Similar to Section 4.5.2, 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, this is not enough for a scenario that is not well-behaved. 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 matches with key target attributes are not merged. For example, All schools, Free meals eligibility and Schools with additional languages all match the key target attributes, and thus are joined, but Defibrillators, Road and Safety Training and Bikeability only match non-key target attributes, so ++Spicy does not consider them for merging. 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. For example, Road and Safety Training and Bikeability are merged with the other relations through full outer joins.

Although Dynamap outputs almost all expected attribute values (its recall is 0.994), the output contains tuples that are only partially correct. This is because of the low overlap between the data about defibrillators and other sources, and after each iteration there is the pruning phase where the profile data that does not seem helpful is discarded. In this situation, the source containing information about defibrillators was overlapping in a very low proportion with the other sources, thus, the partial inclusion dependencies that could have been used in a full outer join operation were discarded, so the merge was no longer possible. This type of pruning is performed in order to avoid merging sources where only
a reduced number of tuples can be correlated, and the others padded with nulls. Thus, Dynamap decides not to add defibrillators to other mappings through join as a merge with a low overlap produces more nulls than it adds constants to the target. However, the defibrillators source is added through union to the complex mapping that is built through set-cover method as stated in the procedure of mapping selection in Section 4.5.

Tuple level. Considering the completeness of the output tuples, it can be observed that Dynamap outperforms ++Spicy, i.e., the number of complete true positive tuples produced by Dynamap is 2.34 times higher than for ++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), i.e., ++Spicy makes use of explicit join paths, and removes redundancy by using equality generating dependencies, but this is not enough in this scenario. 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. Thus, some information is aligned even when the tuples in two sources only partially align. Due to this difference between the two algorithms, ++Spicy is not able to correlate all the information so it outputs mostly partially correct tuples (266), 40 tuples are considered incomplete false negative tuples as they have a few correct values, but mostly null values where in the ground truth there were expected non-null values.

For creating the output tuples, ++Spicy creates 24 intermediate tables, while Dynamap generates a mapping that can run over the initial input sources to populate the target schema.

4.5.4 Profiling Data Propagation - Accuracy at Scale

Motivation. In this section we investigate the accuracy of the propagated inclusion dependencies (Section 3.6) for the scenarios in Sections 4.5.2 and 4.5.3, i.e., how accurate the profiling data is after a set of iterations given that, as shown in Tables 3.2 and 3.3, some of the overlaps are computed through estimations.

Experiment. To measure accuracy, we materialized the intermediate mappings generated in all iterations, and we used SINDY [Kruse et al. (2015)] to obtain the ground-truth inclusion dependencies with accurate overlap. Then, we compared
4.5. SCALABILITY EVALUATION

Percentage of estimated overlaps with error in the range

<table>
<thead>
<tr>
<th></th>
<th>(0.0,0.1)</th>
<th>(0.1,0.2)</th>
<th>(0.2,0.3)</th>
<th>(0.3,0.4)</th>
<th>(0.4,0.5)</th>
<th>(0.5,0.6)</th>
<th>(0.6,0.7)</th>
<th>(0.7,0.8)</th>
<th>(0.8,0.9)</th>
<th>(0.9,1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Realestate</td>
<td>93.3</td>
<td>3.1</td>
<td>1.1</td>
<td>0.8</td>
<td>0.6</td>
<td>0.5</td>
<td>0.4</td>
<td>0.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>Schools</td>
<td>56.6</td>
<td>19.4</td>
<td>10.4</td>
<td>8.7</td>
<td>2.4</td>
<td>1.8</td>
<td>0.5</td>
<td>0.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 4.3: Error ranges

the estimated and ground-truth overlaps for the inclusion dependencies that Dynamap propagated using the formulas in Tables 3.2 and 3.3.

Results. The results of the experiment can be seen in Table 4.3, which identifies the percentage 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 overlaps, and 28,530 were different. The ones that were different were split into error ranges that can be seen in Table 4.3. It can be observed 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 4.3, it can be observed that most estimates had no error or an error below 0.1. The mean average error is 0.12.

In both scenarios, most of the erroneous overlaps stem from the usage of previously approximated overlaps, thus, one could say that once an inclusion dependency degrades, its overlap will lead to other propagated inclusion dependencies’ degradations. Also, although the number of generated inclusion dependencies for the schools scenario is smaller than in the real-estate scenario, the percentage of different overlaps is greater in the schools scenario, i.e., 25% in real-estate compared to 74% in schools. This is reflected by the mean average error values as well because only \(\approx6\%\) of the inclusion dependencies in the schools scenario are full inclusion dependencies, c.f. \(\approx23\%\) for the real-estate scenario. Full inclusion dependencies are used in most conditions in Tables 3.2 and 3.3 that yield accurate overlaps, and their absence leads to more cases where approximations are necessary.
CHAPTER 4. MAPPING GENERATION AT SCALE

4.5.5 Instance Complementarity at Scale

Motivation. In this section we investigate the impact of specific properties of union-dominant integration scenarios on the runtime performance of Dynamap and ++Spicy. 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, e.g., by bringing together property data from many real estate agencies. Referring to the example in Section 4.5.2, suppose the scenario had various agency relations that contained street, price, postcode, and city from different UK cities, and this information was needed in the target, thus the agency relations are instance complementary w.r.t. to a target.

Scenario. In this type of scenario, we use Synthegrate to vary the number of union operations in the correct mapping. 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). A timeout of one hour was set and the generation process was stopped if this time limit was reached.

Results. The mapping generation times for different numbers of union operations in the mapping scenario are in Figure 4.5(a). In terms of result quality, the result tuples are exactly as in the ground truth for both algorithms. In Figure 4.5(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 approximately 22 minutes. The time increase for both algorithms comes from the fact that, in such scenarios, all permutations of the input relations are reasonable 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 the formula in Section 4.1, and thus more rapidly than is reflected in Figure 4.5(a).

4.5.6 Schema Complementarity at Scale

Motivation. In this section we investigate the impact of specific properties of join-dominant integration scenarios on the runtime performance of Dynamap and ++Spicy. These scenarios are a synthetic representation of real-world cases where the relations that need to be merged each contain different attributes that are needed in the target, e.g., by bringing together information about a school from many sources.

Scenario. In this type of scenario, we use Synthegrate to create scenarios in which we vary the number of join operations in the correct mapping. In order to avoid any unintentional inclusion dependencies between the sources, we set the generator to always create a new pair of attributes to use in the join conditions when splitting a table. Otherwise, reusing already created attributes could create unwanted overlaps over which we would not have any control, thus, making it difficult to correlate the runtime results with the merge opportunities.

Results. The results can be seen in Figure 4.5(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, the generated ++Spicy mapping could not be run on the input database.

In Figure 4.5(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. In this type of scenario the opportunities for combining relations are fewer than in a union dominant case (Section 4.5.5) as not all sources can be merged.
with all others as they have overlapping attributes only with some of the other sources. The time difference between Dynamap and ++Spicy comes from the fact that 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 which are, in fact, equivalent mappings.

### 4.5.7 Instance & Schema Complementarity at Scale

**Motivation.** In this section we investigate the impact of specific properties of mixed-operation integration scenarios on the runtime performance of Dynamap and ++Spicy. These synthetic scenarios represent real-world cases where some but not all relevant source relations contain the same type of information. For example, in the schools experiment (Section 4.5.3), several relations could contain data about the schools in each county, and other relations contain data that can supplement the basic data about all schools through joins.

**Scenario.** In this type of scenario, we use Synthegrate to vary the number of union and join operations expected in the correct mapping. Through the variation of operators, merge opportunities are 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 this is not by design and it is not possible to avoid as in Section 4.5.6.

**Results.** The results are shown in Figure 4.5(c). Note, firstly, the different scale in the horizontal axes in Figure 4.5(c). There is an order of magnitude difference in values with unions much more frequent than joins. In practice, we could argue that there could be cases where there are more sources that are union candidates than join candidates, thus, this difference should be representative for real-world scenarios as well.

For these scenarios, 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 the target key constraints to decide to merge the sources, while Dynamap does
4.5. SCALABILITY EVALUATION

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 profiling 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 opportunity is detected as a full outer join.

In Figure 4.5(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 3.6, the propagation of the profiling data is influenced by the chosen operators. In this case, some mappings became unavailable to merge with other mappings as less profiling information was transferred to them from the parent mappings because of the use of a full outer join instead of a join.

Unlike scenarios in Section 4.5.5, in this type of scenario, Dynamap is no longer able to prune so many merges as not all input sources merge through union, i.e., some sources need to be joined as they contain different types of information than the sources that are expected to union. Thus, the merge of all subsets of instance complementary tables will be reasonable candidate mappings.

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 4.5(c).

4.5.8 Efficiency of Pruning Strategies

Motivation. In this experiment we investigate the effectiveness of the pruning strategies presented in Section 4.3.

4.5.8.1 Pruning the Search Space

Scenario. The scenarios were created with SYNTHEGRATE by increasing the number of expected join and union operations, while keeping them equal, e.g., the smallest scenario (11 sources) contains 5 unions and 5 joins, and the largest contains 11 of each type of operation (23 sources).
We measure and report the runtimes, the number of generated mappings and the number of pruned mappings for the effectiveness of the pruning strategies for the search space in 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, the pruning of the profiling data is disabled. Dynamap generated the expected mapping under all settings. The runtime results are shown in Figure 4.6 and the numbers for generated and pruned mappings are shown in Table 4.4.

**Results.** In Figure 4.6, 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 (depicted at 3600 seconds in Figure 4.6), 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 in the first place considerably affects subsequent iterations, thus increasing the runtime.

Table 4.4 shows the impact of each of the pruning strategies by comparing the
4.5. SCALABILITY EVALUATION

<table>
<thead>
<tr>
<th>Scenario (sources number)</th>
<th>Number of generated and pruned mappings per strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>generated</td>
</tr>
<tr>
<td>11</td>
<td>2,045</td>
</tr>
<tr>
<td>13</td>
<td>5,792</td>
</tr>
<tr>
<td>15</td>
<td>16,923</td>
</tr>
<tr>
<td>17</td>
<td>48,275</td>
</tr>
<tr>
<td>19</td>
<td>138,263</td>
</tr>
<tr>
<td>21</td>
<td>396,302</td>
</tr>
<tr>
<td>23</td>
<td>N/A</td>
</tr>
</tbody>
</table>

* N/A - the numbers could not be computed as the mapping generation process exceeded one hour

Table 4.4: Impact of pruning strategies for the search space – number of generated mappings (existed in the search space at some point during the mapping generation process) compared to number of pruned mappings

The number of *generated mappings* (i.e., mappings that exist as intermediate mappings in the search space at some point) and the number of *pruned mappings* (i.e., mappings that are discarded from the search space considering their usefulness for the end result). For the strategies where the runtime exceeded an hour, the figures could not be computed as the process did not terminate (described as not applicable, N/A, in Table 4.4). An interesting observation can be made here for the scenario with 21 sources, comparing PCSM and PSUM strategies. Here, although the numbers of pruned mappings are close (1,981 for PSUM and 2,163 for PCSM), the number of generated mappings for PSUM is 1,188.5 times larger than the number of generated mappings with the PCSM strategy active. As stated above, a strategy that prevents the generation of mappings (PCSM) is more effective than a strategy that allows the generation of mappings (PSUM or RUPM) and then prunes them from the search space.

### 4.5.8.2 Pruning Profiling Data

**Scenario.** Given that the scenarios created by Synthegrate create some unintentional merge opportunities, but not a significant number, we decided to use the two real-world scenarios in Sections 4.5.2 and 4.5.3 where there are more attributes with overlapping values:

- **Real-estate scenario:**
  - 14 web-extracted and open-government sources
  - 68 candidate keys
  - 1734 partial inclusion dependencies
  - 510 full inclusion dependencies

- **Schools scenario:**
CHAPTER 4. MAPPING GENERATION AT SCALE

Figure 4.7: Effectiveness of the pruning strategies for the profile data

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Number of generated and pruned inclusion dependencies per strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
</tr>
<tr>
<td>real-estate</td>
<td>generated</td>
</tr>
<tr>
<td>schools</td>
<td>N/A</td>
</tr>
</tbody>
</table>

N/A - the numbers could not be computed as the mapping generation process exceeded one hour

Table 4.5: Impact of pruning strategies for the profiling data – number of generated inclusion dependencies (inferred at some point during the mapping generation process) compared to number of pruned inclusion dependencies

- 11 open-government sources
- 48 candidate keys
- 681 partial inclusion dependencies
- 47 full inclusion dependencies

We measure and report the runtimes, the number of generated and pruned inclusion dependencies for the effectiveness of the pruning strategies for the profiling data in five different cases: with all pruning strategies active (all), with none active (none), and, with each pruning strategy activated separately, i.e., removing idle inclusion dependencies (IDLE), removing inclusion dependencies with overlap below a minimum threshold (THR), and pruning inclusion dependencies between attributes in the same mapping (SMAP). In each case, the pruning of the search space is disabled and the runtimes were capped to one hour. The runtime results are in Figure 4.7, and the numbers of generated and pruned inclusion dependencies for each strategy can be seen in Table 4.5.

Results. The runtime results are shown in Figure 4.7, where the cases that exceeded one hour are plotted at 3600 seconds. One can observe that, for both
scenarios, the pruning strategies significantly improve the running time if they are all active at the same time, however, activated separately, only some of them run under one hour.

For both scenarios, with all the pruning strategies for profiling data active (all case), the runtime is less than five minutes. However, comparing with the pruning strategies for the search space, the profiling data pruning is not as effective (as can be seen in Figure 4.6 – all case). Nonetheless, it significantly changes the runtime for the better as without any pruning strategies active, both real-world scenarios run in over an hour (none case – depicted at 3600 seconds as it exceeds one hour).

Moreover, the SMAP case runs in more than an hour (e.g., in SMAP case, for the realestate scenario, Dynamap computes the sub-solution for iteration 4 in 2 hours, 54 minutes and 59 seconds). This is due to the fact that the purpose of SMAP strategy is to prune inclusion dependencies that are between attributes belonging to the same mapping, however, in the early iterations, this pruning strategy has little effect, thus, most of the inclusion dependencies are kept. SMAP and none runtimes are similar, i.e., they both exceed one hour, given the little improvement that SMAP is able to make in early iterations.

In the IDLE case, the runtime for the schools scenario is significantly smaller (approximately 40 minutes) than the runtime for real-estate scenario (over one hour – e.g., Dynamap is able to compute the sub-solution for iteration 7 in 6 hours, 54 minutes and 8 seconds). The purpose of the IDLE strategy is to prune inclusion dependencies that are not predicted to be used in further merges, i.e., which do not share candidate key attributes on either its referenced or dependent sides. As seen in the scenarios details above, the real-estate scenario has almost three times more inclusion dependencies (2244 inds) than the schools scenario (728 inds). Moreover, the real-estate scenario has 20 more candidate keys than in the schools one, meaning that more inclusion dependencies have a chance of sharing candidate keys, thus, more inclusion dependencies are kept, instead of discarded. Given these differences, one can say that the runtime discrepancy comes from two reasons: i) the large difference of inclusion dependencies (that need to be propagated and searched through), and ii) the difference of candidate keys which are bound to accommodate more merge opportunities in the case of real-estate.

For the THR case, the minimum threshold was set to 0.5, i.e., keep only
inclusion dependencies that have an overlap above or equal to 0.5. It can be observed that the strategy improves over the runtime, however, as explained in Section 4.3.3, this strategy is a trade-off between timely results and the amount of explored search space. Discarding inclusion dependencies could mean that merge opportunities will not be reached without the profile data that is discarded during pruning. For instance, in the real-estate scenario, with a minimum threshold set to 0.5, the ground-truth results are not reached as necessary inclusion dependencies are discarded in intermediate iterations, but with a minimum threshold of 0.3 it produces the results seen in Section 4.5.2.

As seen in this experiment, the results on the pruning strategies for profiling data are highly dependent on the data values in the sources, thus, the more overlapping involving sources with merge opportunities, the smaller the effect of the pruning phase. By keeping profiling data, merge opportunities are kept which leads to more mappings that are added to the search space. This considerably affects subsequent iterations, thus increasing the runtime.

4.6 Conclusions

In this chapter, we have extended the work in Chapter 3 by adding a pruning component to the mapping generation process. The pruning strategies that we propose rely on (propagated) profiling data such that the mapping algorithm can safely discard mappings and/or profiling data that are not promising in further merges, thereby keeping the search space from growing to become infeasibly large for even modest scenarios. We have evaluated the new version of Dynamap against a set of complex scenarios generated by the state-of-the-art generator, i.e., iBench. We also have shown how Dynamap performs on real-world scenarios, one of which is an extension of the experiment in Section 3.8.2, adding to it more source schemas. Furthermore, for the experiments in Sections 4.5.5, 4.5.6, and 4.5.7 we used Synthegrate to generate large scale scenarios with controlled properties. These scenarios are suitable for the integration settings on which we have our focus, i.e., keeping the target schema fixed while the sources can become quite complex to merge. They have shown that Dynamap performs well under tests where hundreds of (possibly) autonomous sources are given as input. The largest scenario on which Dynamap was tested was composed of 555 operations (55 joins and 500 unions), i.e., 556 source relations, for which it produced the
4.6. CONCLUSIONS

expected mapping in less than 7 minutes, while ++Spicy exceeded the set timeout of one hour. Although ++Spicy was not designed to work at scale, in Mecca et al. (2009), the algorithm was tested against a set of scenarios where the most complex one contains 100 tables with an average join-path of 3 joins for which it manages to create a mapping in approximately 12 seconds. Our closest test to this scenario contained 116 relations where 100 unions and 15 joins are expected (Figure 4.5(c)), where ++Spicy ran for 122.41 seconds while Dynamap’s runtime was 2.96 seconds. The time difference between ++Spicy’s performance in Mecca et al. (2009) and in our test could have various reasons, but the most obvious are that the number of input sources is higher, i.e., 116 tables compared to 100, and that the average join-path is 5 times shorter than the number of expected joins in our test, i.e., 15 joins in the join-path compared to multiple join-paths averaging at 3 joins.

The effectiveness of each pruning strategy was shown in Section 4.5.8, where Dynamap was run under different settings where each of the pruning strategies was tested against a set of scenarios that increase in terms of the number of sources.

Throughout the evaluation section, Dynamap was compared to ++Spicy [Marquette et al. (2010)] as we aimed to show that Dynamap is able to operate with less cleanly integrated input sources than the current schema mapping generation approaches, i.e., that it can handle new cases of mapping generation in the wild providing good quality results in a timely manner.

The problem of mapping generation at scale might seem like an over-engineered process when (say) only two schemas are involved, however, the creation of mappings becomes a problem if tens/hundreds of sources are involved. Other approaches for merging independent sources, e.g., at Internet scale, were presented, e.g., in Das Sarma et al. (2012), where a method is proposed for detecting if source relations are either unionable or/and joinable. Their experiments were run over a million sources from Wikipedia. Zhu et al. (2016) determine join paths between the sources treating the problem as a domain search where each attribute is considered a domain, thus, the search becomes a problem of finding attributes from different sources which are pairwise similar, concluding they are joinable. To this end, they use LSH Ensemble [Zhu et al. (2016)] for indexing the domains in a pool of sources. Their experiments were run on relational data comprising 10,635 relations. Nargesian et al. (2018) tackle the problem of attribute
unionability where they present work on determining whether attributes in the different sources come from the same domains, thus, it could be determined if tables are candidates for unioning. Their experiments were driven on repositories of up to 165,236 tables. The work on Skluma [Beckman et al. (2017)] stems from the need to organize datasets by using their extracted metadata, thus, it is presented as a system for organizing large-scale data which extracts deeply embedded metadata, relationships between sources, and contextual metadata. It is proposed that Skluma can be used to manage a large collection of files, e.g., more than half a million, allowing data to be manipulated by its content and topics. These works advance in a similar direction to the work we conduct in this thesis, i.e., data integration/management over autonomous sources, and they could complement our work by, for example, integrating into Dynamap another method for providing join-path information [Zhu et al. (2016)], or adding information about which tables are candidates for unioning [Nargesian et al. (2018)], or both [Das Sarma et al. (2012)]. These additions could extend our proposed method for performing the operator search presented Section 3.3, while Skluma could be used to create more intricate relationships between the sources and determine which merges could be favoured based on their metadata labels, e.g., to change the fitness function to a more complex one that can use various types of metadata about the sources.

Mapping generation at scale is part of a larger problem, i.e., data integration at scale, where numerous heterogeneous data sources are expected to be integrated, thus, our tackled problem is closely related to a series of other issues involved by data integration at scale. The problem of data integration at scale was recognized in works such as Dong and Rekatsinas (2018) where a synergy between data integration and machine learning is presented given that machine learning algorithms can rely on usefully integrated large-scale data for the training process, and vice versa as data integration is being helped by recent machine learning advancements such as highly-scalable inference engines and deep learning for improving entity resolution, data fusion, schema alignment, etc. Also, another related problem was studied in Pimplikar and Sarawagi (2012) where they show a method for applying a graphical model to align web tables with a knowledge base, by concomitantly aligning schemas and entities.

To conclude, there have been works on different facets of data integration at scale, such as data discovery [Zhu et al. (2016); Nargesian et al. (2018)], entity
resolution [Pimplikar and Sarawagi (2012); Trivedi et al. (2018)], however, there hasn’t been much focus on mapping generation at scale. There are still multiple problems to solve given that the landscape of the integration problem has shifted to scale [Miller (2018)]. Some of the open issues in this direction are mentioned in Chapter 6.
Chapter 5

Mapping Generation for a Complex Target

"Simple things should be simple, complex things should be possible." – Alan Kay (1940)

This chapter describes how we have extended Dynamap to a mapping generation technique \( \text{Dynamap}^{(e)X(\text{tended})} \) that tackles the problem of mapping generation for a complex target by an extension to the work described in Chapters 3 and 4. By mapping generation for a complex target we mean that Dynamap\(^X\) is able to tackle input scenarios where the target schema consists of several tables that can have candidate/primary key constraints and therefore be linked by foreign keys.

Referring back to the objectives outlined in Section 1.3, this chapter addresses Objective 3, i.e., tackling the problem of mapping generation over autonomous sources for a complex target schema subject to constraints. To this end, it reports two contributions: Contribution 3.1 describes an algorithm that populates target attributes that are subject to constraints while avoiding violating them as much as possible; and Contribution 3.2 describes a mechanism for characterizing the generated mappings in terms of the degree of satisfaction of the target constraints. Motivation. In Chapters 3 and 4, we described Dynamap, a mapping generation algorithm that tackles the problem of generating mappings between a set of autonomous, heterogeneous sources and a single-relation target schema without constraints. In Section 4.5, we showed that the generated candidate mappings are of good quality and are produced in a timely manner even when the search space is large and complex. However, Dynamap is designed to handle a target schema
without constraints. For a more complex target, comprising more than one table, Dynamap generates a set of mappings for each target table independently from the other target tables. If the target tables have primary-foreign key relationships, then the output tuples in such mappings populate the target tables disregarding, and hence violating the target constraints. This chapter extends the work on Dynamap as described in Chapters 3 and 4 with techniques that allow mapping generation between a (set of) source schema(s) and a multi-table target schema such that the target constraints are taken into account.

![Target schema with key and foreign key constraints](image)

**Figure 5.1: Target schema with key and foreign key constraints**

### 5.1 Example Scenario

We use one scenario to motivate the need to generate mappings between a repository of autonomous sources and a complex target schema, i.e., with constraints, for which we propose a solution in this chapter. The example is illustrated in Figure 5.1, and it exemplifies two of the schema constraints that can appear in various data integration scenarios.

**Example 5.1.1.** Consider a similar scenario as in Examples 1.1.1 and 1.1.2, previously used. In those examples, a company delivers real-estate information to a business partner, but the information needed by the partner is contained in
different data sources that do not contain the same kind of information, i.e., the sources are schema complementary w.r.t. the required data.

Figure 5.1 depicts the above-described scenario, i.e., different source schemas contain parts of information that is required.

The differences between this example and Examples 1.1.1 and 1.1.2 are that, in the previous examples, all the data was in one target table and there were no target constraints, while, here, the target schema contains two target relations with declared primary keys linked by a primary-foreign key relationship between the Area_ID attributes.

## 5.2 Overview of the Approach

**Considered approaches.** One approach to extending Dynamap to mapping generation for complex targets could be to generate mappings for each target table (separately) and then run a post-processing step that aims to reconcile the data so as to satisfy the target constraints. In practice, detecting mappings that satisfy the target constraints involves the materialization of the candidate mappings. The separate generation of mappings for each target table loses uniqueness and inclusion dependency relationships between the generated sets of mappings, so we can no longer rely on propagated profile data. As a result, this approach involves materializing all the data from candidate mappings, and then selecting from these mappings subsets that meet the constraints. This is potentially expensive and hinders our aim of producing quality results in a timely manner.

Another approach could be to pre-process the complex target schema and transform its tables into a single universal target table that comprises all target attributes. Then, Dynamap can generate a solution for populating the latter from the sources. After the universal target table is materialized, existing mapping generation methods for target schemas with constraints, such as ++Spicy [Mar-nette et al. (2010)], can be run to extract the data from the materialized universal target table transferring it to the tables of the complex target schema while satisfying the constraints on the latter. As described in Section 2.1.4, ++Spicy can take as input target constraints which it tries to satisfy by using target egds to remove redundancy and this could lead to satisfying target constraints. However, we consider this approach infeasible in our setting because, as shown in Sections 4.5.2 and 4.5.3, the source data may only partially merge, so we cannot assume
that after the merge, the output tuples do not violate the target constraints as
the source relations are not necessarily part of a well-behaved source schema, as
assumed by ++Spicy. In our setting, it is not safe to assume that all constraints
are satisfiable, let alone satisfied as the sources are not all designed for integration
and extraction.

In order to tackle this challenge, we describe a method for characterizing
the mappings based on the degree to which they violate the target constraints.
We do this by creating mappings that: (i) generate synthetic data values where
there are attributes with no/scarce source-extracted data and that have key and
foreign key constraints; and (ii) discard subsumed tuples before materialization.
Then, based on the method used to generate the synthetic values, Dynamap\(^X\)
estimates confidence scores for satisfying a set of attribute-level properties, i.e.,
attribute completeness, source-extracted ratio, and key consistency per attribute
(as described in Section 5.5). Using these scores, we can (i) estimate the quality of
the data that the mappings generate, and (ii) detect whether the target constraints
are guaranteed to be satisfied or not. In order to create these confidence scores,
the process of generating the synthetic data needs to be transparent such that
Dynamap\(^X\) is aware of the performed steps, thus, it is able to compute the scores.
Using ++Spicy would mean treating the process of synthetic data generation as a
black box which would not allow Dynamap\(^X\) to consider how the data is created,
thus, it would not be able to generate the confidence scores without materializing
the mappings ++Spicy generates, which is a post-processing step we aim to avoid.
Figure 5.2: DynamapX workflow – approach for target schema with constraints
5.2. OVERVIEW OF THE APPROACH

Chosen approach. We extend Dynamap to Dynamap$^X$, which is capable of taking target constraints into account. Thus, the mappings that Dynamap$^X$ generates aim to comply to the latter. Nevertheless, given that the mappings are over autonomous, heterogeneous sources, one cannot expect any mapping to perfectly align the data, and, although Dynamap$^X$ tries to satisfy the target constraints, some may not be satisfied. Dynamap$^X$ generates a set of indicators for each of the mappings that aim to convey the degree to which the mappings are expected to satisfy the target constraints (as explained in Section 5.5).

The workflow of Dynamap$^X$ is shown in Figure 5.2. Dynamap$^X$ has an additional preprocessing step w.r.t. Dynamap (called Generate UT) that generates a \textit{UTR} for the target schema, i.e., the universal target relation whose columns consist of all the attributes in every target relation as are linked by foreign keys, i.e., in a join chain. After the creation of the \textit{UTRs} (one for each join chain in the target schema), Dynamap$^X$ searches for mappings for each of these \textit{UTRs} as in the previous approach, taking one \textit{UTR} (i.e., only one target table) at a time.

After the \textit{UTR} mappings are created, there is a post-processing step where the algorithm alters the \textit{UTR} mappings in order that they generate labelled nulls (i.e., skolems) when executed (post-processing – step 1). The creation of the labelled nulls helps satisfy candidate key and foreign key constraints in the target schema. Then, in the subsequent post-processing step (step 2), \textit{projections} are applied on the views of the mappings for the \textit{UTR} such that the mappings for the initial target relations that were bundled, so to speak, in that \textit{UTR} are obtained. The last post-processing step (in Figure 5.2, post-processing - step 3) aims to modify the candidate mappings such that, when executed, redundancy is reduced by removing subsumed output tuples. In the post-processing steps, the algorithm alters all the candidate mappings without materializing any data, so, based on the corresponding profile data and metadata information, we contribute a method to characterize the output candidate mappings without executing them.
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

5.3 Universal Target Composition

This section describes the method for composing the universal target. This is formalized in Algorithm 6 and corresponds to Generate UT step in Figure 5.2. 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 with Dynamap, where the UTR is the single target relation for which mappings are sought, whereas the UTG is used in the decomposition of the Dynamap-generated UTR mappings. The method for transforming a UTR mapping into mappings for the target relations it bundles up is described in Section 5.4.

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 universal target graph(s), i.e., the output (output_uts, on line 15) is of the form [(utr1, utg1), (utr2, utg2), ..., (utr_n, utg_n)], where n is the number of (disjoint and exhaustively explored) join paths in the target schema. DynamapX therefore generates a separate set of mappings for each pair (utr, utg) by running Dynamap each time. The algorithm first creates the UTG, then uses it to create the UTR.

5.3.1 Universal Target Graph

Motivation. The purpose of the UTG is to determine how to compose and decompose the UTR mappings. The universal target graph not only preserves
5.3. UNIVERSAL TARGET COMPOSITION

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.

**Directed graph notions** [Aho and Hopcroft (1974)]. For completeness, we outline below the notions for directed-acyclic graphs which we use in the description of our approach.

A *directed graph* is a pair $G = (V, E)$ with the following properties:

1. The first component, $V$, is a finite, non-empty set. The elements of $V$ are called the *vertices* of $G$.
2. The second component, $E$, is a finite set of ordered pairs of vertices, i.e., $(v, w) \in E$, where $v, w \in V$, $v$ is called *tail* and $w$ the *head*. That is $E \subseteq V \times V$. The elements of $E$ are called the *edges* of $G$.
3. In a DAG, for an edge $(v, w) \in E$, we say $w$ is *adjacent* to $v$.
4. The *out-degree* of a vertex $v$ is the number of vertices adjacent to $v$, i.e., for which $v$ is the *tail*.
5. The *in-degree* of a vertex $v$ is the number of edges for which $v$ is the *head*.
6. A *path* is a sequence of edges of the form $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$, where we say that the path is from $v_1$ to $v_n$ and is of length $n - 1$.
7. A *cycle* is a path of length at least 1 which begins and ends at the same vertex.
8. A *direct acyclic graph* (DAG) is a directed graph with no cycles.

**Universal Target Graph as DAG.** In our setting, the *universal target graph* (UTG) is a *direct acyclic graph*, where:

- the nodes are (different) initial target relations,
- the edges are represented by foreign key relationships,
- the direction of an edge is given by the foreign key in the sense that the direction is from the referenced to the dependent relation. The direction of the edges determines the order in which the labelled nulls are created for each initial target relation. The description of the labelled nulls generation is in Section 5.4.2.

**Creation of UTG(s).** The creation of the universal target graphs is part of the Generate UT pre-processing step (Create UTG, in Figure 5.2) and corresponds to lines 4-10 in Algorithm 6. In lines 4-5, Dynamap$^X$ adds all initial target relations to a single graph, target$_-$graph. If the provided input target is well-behaved, i.e., the desired keys and foreign keys are explicitly stated, in lines 6-7, it uses the
declared foreign keys in the target schema to create the directed edges between
the nodes. In line 8, \texttt{FindConnGraphs}\footnote{In this thesis, if a method call is typeset in \texttt{sans-serif}, then it is a helper method. All helper methods are written and described in more detail in Appendix A.} takes \texttt{target\_graph} as input and finds all its connected graphs (there is one connected graph per join path). Notice that each created UTG that is so created does \textit{not} have common nodes with other UTGs because the join paths in each graph are exhaustively explored and included in the graph.

5.3.2 Universal Target Relation

\textbf{Motivation.} The UTR is used as the target in the generation of mappings using Dynamap (as described in Section 3.4). We use a UTR so as to have a single table as target, a requirement for Dynamap. In this way, the data in the sources is first aligned in the format of the UTR and, then, we use the UTR mappings to obtain the mappings for the initial target relations bundled up in the UTR. Relative to the initial target tables, the schema of the UTR represents the schema of the source key-foreign key joins from the join path to which the UTR corresponds. Populating the UTR first, and then \textit{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 the data is split for populating the target tables.

\textbf{Creation of UTR(s).} The creation of the universal target relations is done in a pre-processing step (\textit{Create UTR}, in Figure 5.2) and corresponds to lines 11-13 in Algorithm 6. In order to create the universal target relations, the algorithm uses the already created universal target graph (which is a representation of a join path). By following the join graphs in the UTGs, the algorithm creates each UTR as a single new relation comprising all target attributes in all the relations in the join graph (lines 12-13).

In Algorithm 6, the (UTR, UTG) pairs are created and added to the final output (in line 14) which returns in line 15.

\textbf{Example 5.3.1.} In Figure 5.1, the target schema comprises two target relations: \textit{Area Info} and \textit{UK Realestate} tables. There is only one join path that connects the two relations as there is only one foreign key, viz., \texttt{UK\_Realestate.Area\_ID} $\rightarrow$ \texttt{Area\_Info.Area\_ID}. The corresponding UTR has the following schema:
Universal Target Decomposition

This section describes the method for decomposing the UTRs, i.e., the modification of the UTR mappings as Dynamap$^X$ aims to satisfy the target constraints through the creation of labelled nulls and removal of subsumed tuples. Once the latter tasks are completed, Dynamap$^X$ applies a projection on the modified mappings, thereby creating a new mapping for each initial target relation. The decomposition is done in the post-processing step in Figure 5.2 and is formalized in Algorithm 7.

Algorithm 7 (Decompose) takes as input a UTR mapping (utr_map) and its corresponding universal target graph (utg). The number of output mappings is equal to the number of initial target tables that were used in the creation of the universal target. Each output mapping in output_maps corresponds to one node (i.e., one target table) in the UTG. In line 3, the algorithm sorts the set of nodes in utg in topological sorted order (described in Section 5.4.1). The ordering is important for the generation of labelled nulls as this ensures that no dependent relation is processed before its referenced relations (more details in Section 5.4.2). In lines 4-7, the algorithm sequentially modifies the UTR mapping so that 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). The Alpha method for modifying the mappings to create labelled nulls is explained in Section 5.4.2. In lines 8-11, the algorithm uses the UTR mapping to apply projection on it according to the attributes in the initial target tables (Projection, in line 9). Then, it alters the resulting projected mapping so that it does not produce subsumed tuples when executed (Beta, in line 10). The Beta method for avoiding redundant tuples is explained in more detail in Section 5.4.3. The mappings for the initial target tables are added to the output set (line 11) which is returned (line 12).
Algorithm 7 Decomposition of UTR mappings

1: function decompose(utg, utr_map)
2:     output_maps ← []
3:     sorted_nodes ← TopoSort(utg.nodes)
4:     for each tr in sorted_nodes do
5:         key ← FindPKey(tr)
6:         skolem_atts ← tr.attributes − {key}
7:         utr_map ← Alpha(skolem_atts, utr_map)
8:     for each tr in sorted_nodes do
9:         target_map ← Projection(utr_map, tr.attributes)
10:        target_map ← Beta(target_map)
11:        output_maps.add(target_map)
12:     return output_maps

5.4.1 Topological Sort using Kahn’s Algorithm

Kahn’s algorithm. Kahn’s Algorithm [Kahn (1962)] is an application of breadth-first search for topological sorting in a DAG. Topological sorting is used to determine the sequence in which a set of nodes (representing real-world concepts, e.g., events) can be parsed given that some might depend on others (e.g., that some events cannot occur before other events). The resulting ordered list represents a path such that each node can be reached from a previous node in that order and where (at least) one node is expected to have the in-degree of 0 as such a node is the starting point of the ordered list. A DAG can have one or more topological orderings.

Motivation. For the creation of the labelled nulls, the order of the relations is important so that tables with referenced attributes are processed before the tables with corresponding dependent attributes (Section 5.4.2 describes the method for the generation of skolems). For this purpose, we use Kahn’s algorithm. This method ensures that the (full) inclusion dependencies between pairs of constrained target attributes are satisfied.

Application on our problem. In our case, the UTG contains the target relations as nodes, and the foreign key relationships between them as edges: from the referenced relation to the dependent one. Given that the DAG is built in this manner, when Kahn’s algorithm is run on it, the output will be a node path where the target relations are ordered based on their foreign key dependencies, such that in the first position in the list is a target table with no dependencies, in second position the tables that reference it and so on until all initial target relations that are connected by foreign keys in the DAG are output.
Example 5.4.1. Given three relations: $T_1(a, b)$, $T_2(a, c, x)$, and $T_3(x, y)$, where $T_2.a \rightarrow T_1.a$ and $T_3.x \rightarrow T_2.x$ are foreign keys, the corresponding UTG is: $T_1 \rightarrow T_2 \rightarrow T_3$, where the three tables are nodes, and the topological sort is: $T_1, T_2, T_3$ as $T_1$ has an in-degree equal to 0, i.e., does not reference any other relations, $T_2$ is dependent on $T_1$ and, at the same time, referenced by $T_3$, and $T_3$ is only dependent on $T_2$.

In the example in Figure 5.1, the topological order for the two target tables is Area Info, then UK Realestate.

5.4.2 Labelled Nulls Generation

Prior work. The notion of explicit object identifiers was first proposed in Goldberg and Robson (1983), and was further advanced in the context of relational databases in Khoshafian and Copeland (1986) and Kuper and Vardi (1984). In Hull and Yoshikawa (1990), the problem of generating labelled nulls was studied with the purpose of creating object identifiers, a.k.a. surrogates, using Skolem functors (introduced by Thoralf A. Skolem, 1920s). More recently, Popa et al. (2002); Fuxman et al. (2006); Mecca et al. (2009); Alexe et al. (2012), and Arocena et al. (2013) use labelled nulls in the context of mapping generation to express correlations between attribute values, i.e., merging data, or to create new values where there is missing data in order to help satisfy target constraints on attributes that cannot be null.

Motivation. In the context of mapping generation, creating labelled nulls is driven by the need for a value for an attribute that is not matched but is involved in a schema constraint, e.g., as a candidate key or a foreign key. In Popa et al. (2002), they describe that a target element $E$ is needed if $E$ is (part of) a key or a foreign key or is both not nullable and not optional. Otherwise, if the target attribute is unmatched, but does not have a constraint on it, then it is not crucial for the integrity of the target constraints. The reason we choose to generate skolems is significantly overlapping with the reasons in prior work, i.e., to help to satisfy the target constraints. However, it was shown in Sections 4.5.2 and 4.5.3 that in the context of mapping generation over autonomous sources, not all sources are readily available to merge as in a well-behaved source schema. This requires the generation method (i) to try to cover the case where the target attributes are only partially populated and (ii) to avoid creating redundant tuples. Moreover, in case the mapping cannot be guaranteed to satisfy the target constraints, Dynamap$^X$
outputs an estimation of the degree to which the constraint is expected to be satisfied (more details in Section 5.5).

**Skolem functions.** A *skolem function* is used when a formula of the following format:

\[ \forall x \exists y \phi(x, y) \]

is transformed through *skolemization* into

\[ \exists f \forall x \phi(x, y), \text{ where } y \leftarrow f(x). \]

This means that the existential variables \( y \) are replaced by the values produced by the new existentially quantified function \( f \) (*skolem function*) applied on the set of universally quantified variables in \( x \) [Aroccena et al. (2013)]. We say that the *skolem function* \( f \) depends on the set of variables in \( x \).

**Labelled nulls generation.** We now describe two methods for generating labelled nulls used by prior work on mapping generation. Popa et al. (2002) propose a method for generating labelled nulls for foreign key dependencies by applying a skolem function on target attributes that have a match in the sources, i.e., that are expected to be populated. Mecca et al. (2009) have a similar approach (implemented in ++Spicy) to the one in Popa et al. (2002): they use skolem functions to generate *skolem strings*, which are further used to generate unique integer identifiers.

**Example 5.4.2.** Consider a simplified version of the example in Figure 5.1 where we focus on the source relation of Manchester (i.e., we disregard the matches in the other two source tables). The *Area_ID* attributes are used in the foreign key constraint between the two target tables, but it is not matched by the source table (Manchester) in either of the two target tables. Thus, in order to correctly align the tuples in the two target tables (according to the declared foreign key), labelled nulls can be invented as follows:

1. Using Popa et al. (2002), if the mapping involves only the *Area Info* relation, then a new unique value *Area_ID* (*Aid* below) is created for each combination of values in the mapped attributes, e.g., in *City*:
   
   (a) For a mapping between *Manchester* and *Area Info* of the form:
   
   \[ M(c, le, a, bey, bn, pc, s, pr) \rightarrow \exists Aid, Ir, Co, Cr : AI(Aid, Ir, c, Co, Cr) \]
   
   where \( Aid, Ir, Co, Cr \) are not matched, we need to infer values for \( Aid \) as it is both a key and a referenced attribute in a foreign key. Following Popa et al. (2002) \( Aid \) becomes \( f_{Aid}(c) \), where \( f_{Aid} \) is a skolem function, as *city* is the only matched attribute in the mapping.
(b) For a mapping between Manchester and Area Info with UK Realestate:
\[ M(c, le, a, bey, bn, pc, s, pr) \rightarrow \exists \text{Aid, Ir, Co, Cr, Pid} : \]
\[ \text{AI}(\text{Aid, Ir, Co, Cr}, \text{UKR}(\text{Pid, Aid, pc, s, pr})) \]
where more attributes are matched, \text{Aid} becomes \( f_{\text{Aid}}(c, pc, s, pr) \). This means that the skolem function \( f_{\text{Aid}} \) depends on all matched attributes in the mapping, regardless of the fact that they belong to different target relations.

2. Using Mecca et al. (2009), the skolem functions take three arguments:
   
   (i) the sequence of facts generated by firing the \( \text{tgd} \) (without the existential variables),
   
   (ii) the encoding on the sequence of joins imposed by existential variables, and
   
   (iii) a reference to the specific variable for which the function is used.

For a mapping between Manchester and the \text{AI} and \text{UKR} target tables
\[ M(ci, le, a, bey, bn, pc, s, pr) \rightarrow \exists \text{Aid, Ir, Co, Cr, Pid} : \]
\[ \text{AI}(\text{Aid, Ir, Ci, Co, Cr}), \text{UKR}(\text{Pid, Aid, pc, s, pr}), \]
Mecca et al. (2009) create the following values using skolem strings:
\[ \text{Aid} \leftarrow f_{sk}(\{ \text{AI}(C : ci), \text{UKR}(C : pc, D : s, E : pr) \}) \],
\[ \{ \text{AI.A} = \text{UKR.B} \}, \]
\[ \{ \text{AI.A} = \text{UKR.B} \} \]
Note that the set elements must be encoded in the lexicographic order so that the skolem function is not affected by the order in which the atoms appear in the \( \text{tgd} \).

In both proposals, the generation technique produces unique values for the correlated tuples in the two target relations, but can produce redundant data when there is no evidence that two tuples represent two different entities.

**Example 5.4.3.** Let us consider the example tuples in Table 5.1 for the Manchester source relation (excluding attributes that are not matched).

The target schema comprises the two tables in Figure 5.1:
\[ \text{AI}(\text{Aid, Ir, Ci, Co, Cr}) \] and \[ \text{UKR}(\text{Pid, Aid, Pc, St, Pr}) \],
with a foreign key \( \text{UKR.Aid} \rightarrow \text{AI.Aid} \).
Using either method in Example 5.4.2, the tuples transferred to the target relations are of a similar form as shown in Tables 5.2 and 5.3 (again, excluding the unmatched attributes, which will be padded with nulls).

Note that although, through the join on $Aid$, the data from the two tables reproduces the data from the initial source table, there are redundant tuples in the $Area\ Info$ target table due to the labelled nulls that were created in $Aid$ (as each output of $f_{Aid}$ is unique for each tuple). In this situation, there is no evidence that the city of Manchester is a different city for each entry, e.g., a way to distinguish between them could be to consider the information for the other attributes $Income\ Rank$, $County$, or $Crime\ rank$, but these are populated with nulls in this scenario as they are unmatched. In the mapping generation context that is addressed in this thesis, i.e., over autonomous sources that were not made to merge, it is undesirable to generate data where there is no evidence that these are necessary in the target. Because of this, we aim to generate mappings that populate target schemas avoiding redundant data.

**Chosen approach.** 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 as it is if it comes from merged sources. The generation of labelled nulls does not tamper with source data: it only replaces missing information. If an attribute is only partially populated, then the labelled nulls only replace the missing information on that attribute. Using a generic example, we now describe the behaviour of $\text{Alpha}$ (line 7) in Algorithm 7, i.e., our method for creating labelled nulls:

Consider a generic $UTR$ schema of the form

$$UTR(a_1, a_2, \ldots, a_n, x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_p, z_1, z_2, \ldots, z_q).$$

The $UTR$ comprises
5.4. UNIVERSAL TARGET DECOMPOSITION

Table 5.2: Area Info tuples

<table>
<thead>
<tr>
<th>Aid</th>
<th>Postcode</th>
<th>Street</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>fAi</td>
<td>M3 1NN</td>
<td>Mirabel St.</td>
<td>725pcm</td>
</tr>
<tr>
<td>fAi</td>
<td>M3 1NP</td>
<td>Mirabel St.</td>
<td>800.00</td>
</tr>
<tr>
<td>fAi</td>
<td>M3 1NN</td>
<td>Mirabel St.</td>
<td>950</td>
</tr>
<tr>
<td>fAi</td>
<td>M3 1NN</td>
<td>100 Greengate</td>
<td>1300pcm</td>
</tr>
</tbody>
</table>

Table 5.3: UK Realestate tuples

attributes from four target relations, $T_1$, $T_2$, $T_3$, and $T_4$, some of which form a join path that has both the chain and the star join patterns (shown in Figure 5.3) as follows:

$T_1(a_1, a_2, \ldots, a_n), T_2(x_1^{a_1_{fk}}, x_2, \ldots, x_m), T_3(y_1, x_1^{y_1_{fk}}, y_2, \ldots, y_p),$ and $T_4(z_1, x_1^{z_1_{fk}}, z_2, \ldots, z_q)$

where the (primary) keys are $T_1.a_1$, $T_2.x_1$, $T_3.y_1$, $T_4.z_1$, and the foreign keys are:

- $T_2.a_{1_{fk}} \rightarrow T_1.a_1$,
- $T_3.x_{1_{fk}} \rightarrow T_2.x_1$,
- $T_4.x_{1_{fk}} \rightarrow T_2.x_1$.

In the first post-processing step in Figure 5.2, using Algorithm 7 (lines 4-7), Dynamap$^X$ populates both the (primary/candidate) keys and the foreign key attributes as follows:

1. Compute the topological orderings of the initial target tables:
   (a) $T_1$, $T_2$, $T_3$, and $T_4$;
   (b) $T_1$, $T_2$, $T_4$, and $T_3$.

Notice that, if (at least) one star-join pattern is present, there can be more than one topological order. In this situation, the four relations are in a star-join pattern where $T_2$ is the central relation. In Algorithm 7, the sorting is done on line 3, through TopoSort.

2. Alter the UTR mapping by starting with the first initial table in the topological-ordered list: for computing labelled nulls, add a skolem function that takes as parameters all attributes except the key, $a_1$, in $T_1$, e.g., $T_1.a_1 \leftarrow f_{skT_1}(a_2, a_3, \ldots, a_n)$. Given the foreign key $T_2.a_{1_{fk}} \rightarrow T_1.a_1$, altering the UTR mapping to populate $T_1.a_1$ means that $T_2.a_{1_{fk}}$ is populated with
data from the same set of generated labelled nulls (when materialized). The UTR mapping becomes:

\[
\text{map}_{\text{UTR}}(f_{skT_1}(a_2, ..., a_n), a_2, ..., a_n,
    x_1, x_2, ..., x_m, y_1, y_2, ..., y_p, z_1, z_2, ..., z_q).
\]

3. Alter the UTR mapping by continuing with the next initial table in topological order, viz. \(T_2\): add a skolem function that takes as parameters all attributes except the key, \(x_1\), in \(T_2\), e.g., \(T_2.x_1 \leftarrow f_{skT_2}(a_{1fk}, x_2, ..., x_m)\). Notice that, by sequentially altering the UTR mapping in topological order, \(a_{1fk}\) has labelled null values from the previous step, and these values are used in the creation of the new labelled nulls for \(T_2.x_1\). After this step, the modified UTR mapping is:

\[
\text{map}_{\text{UTR}}(f_{skT_1}(a_2, ..., a_n), a_2, ..., a_n,
    f_{skT_2}(f_{skT_1}(a_2, ..., a_n), x_2, ..., x_m), x_2, ..., x_m,
    y_1, y_2, ..., y_p, z_1, z_2, ..., z_q)
\]

4. Alter the UTR mapping by continuing with the next initial table in topological order, viz. \(T_3\). Let us assume the algorithm proceeds in topological order from 1a): add a skolem function that takes as parameters all attributes except the key, \(y_1\), in \(T_3\), e.g., \(T_3.y_1 \leftarrow f_{skT_3}(x_{1fk}, y_2, ..., y_p)\). Similarly, \(x_{1fk}\) has labelled nulls from the previous step, and these values are used in the creation of the new labelled nulls for \(T_3.y_1\). After this step, the modified UTR mapping is:

\[
\text{map}_{\text{UTR}}(f_{skT_1}(a_2, ..., a_n), a_2, ..., a_n,
    f_{skT_2}(f_{skT_1}(a_2, ..., a_n), x_2, ..., x_m), x_2, ..., x_m,
    f_{skT_3}(f_{skT_2}(f_{skT_1}(a_2, ..., a_n), x_2, ..., x_m), x_2, ..., x_m),
    y_1, y_2, ..., y_p, z_1, z_2, ..., z_q)
\]
5.4. UNIVERSAL TARGET DECOMPOSITION

\[ f_{skT_3}(f_{skT_2}(f_{skT_1}(a_2, \ldots a_n), x_2, \ldots, x_m), y_2, \ldots, y_p), y_2, \ldots, y_p, z_1, z_2, \ldots, z_q) \]

5. Alter the UTR mapping by continuing with the next (and last) initial table in topological order, viz. \( T_4 \): add a skolem function that takes as parameters all attributes except the key, \( z_1 \), in \( T_4 \), e.g., \( T_4.z_1 \leftarrow f_{skT_4}(x_{1jk}, z_2, \ldots, z_q) \).

Again, \( x_{1jk} \) has labelled null values from previous steps, and these values are used in the creation of the new labelled nulls for \( T_4.z_1 \), while computing the nulls for \( T_3.y_1 \) beforehand does not affect the creation of values in \( T_4 \).

After this step, the modified UTR mapping is:

\[ \text{map}_{UTR}(f_{skT_1}(a_2, \ldots a_n), a_2, \ldots a_n, f_{skT_2}(f_{skT_1}(a_2, \ldots a_n), x_2, \ldots, x_m), x_2, \ldots, x_m, f_{skT_3}(f_{skT_2}(f_{skT_1}(a_2, \ldots a_n), x_2, \ldots, x_m), y_2, \ldots, y_p), y_2, \ldots, y_p, f_{skT_4}(f_{skT_3}(f_{skT_2}(f_{skT_1}(a_2, \ldots a_n), x_2, \ldots, x_m), z_2, \ldots, z_q), z_2, \ldots, z_q) \]

6. The algorithm continues creating labelled nulls until it reaches the last initial target relation in the list (which is not referenced by others), e.g., \( T_4 \) (or \( T_3 \)), where it stops adding skolem functions.

Example 5.4.4. Considering the scenario in Example 5.4.3, i.e., with the same source tuples and the same target schema. Following our approach, the resulting populated target relations are in Tables 5.4 and 5.5 (for simplicity, we exclude the other target attributes as they are unmatched so they are padded with nulls).

Note that the initial source data is gathered through the join of the two target tables on \( Aid \) attributes, and that there is only one tuple representing Manchester city in \( Area\ Info \) relation. This is due to the fact that there is no other evidence to help distinguish 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.

After the \( Aid \) attributes are populated with labelled nulls, the algorithm can proceed to create the labelled nulls for \( UKR.Pid \) as this is the primary key in the \( UKR \) target table. For this, the algorithm uses the values already computed for \( UKR.Aid \). The final output tuples for \( UKR \) are depicted in Table 5.6.


5.4.3 Subsumed Output Tuples

Prior work. The problem of identifying subsumed tuples has been a research topic for a while [Galindo-Legaria (1994); Galindo-Legaria and Rosenthal (1997); Bleiholder et al. (2010)]. The aim is to identify redundant tuples in an instance of a relation. In Bleiholder et al. (2010), they study the possible applications of operators which extend the work on subsumption to minimum and complement union where they describe scenarios where data is fused. Through data fusion, source data could be modified through techniques such as replacing missing information with average column values, or use majority voting to find possible values etc. However, in this thesis, our focus is not on data fusion, but on generating mappings that aim to yield quality results without modifying any of the transformed initial data.

Subsumed tuples. Bleiholder et al. (2010) define a subsumed tuple as:

A tuple $t_1 \in T$, where $T$ is a relation, subsumes another tuple $t_2 \in T$ if:

1. $t_1$ and $t_2$ have the same schema,
2. $t_2$ contains more null values than $t_1$, and
3. $t_2$ coincides in all non-null attribute values with $t_1$.

Tuple subsumption is similar to set containment. A tuple $t_1$ subsumes another tuple $t_2$, if $t_2 \subseteq t_1$. Tuple subsumption is a transitive relationship, so if $t_2 \subseteq t_1$ and $t_3 \subseteq t_2$, then also $t_3 \subseteq t_1$. Tuple subsumption is neither symmetric nor reflexive.
Example 5.4.5. Consider the schema of the target relation Area Info in Figure 5.1 with the following tuples in Table 5.7:

<table>
<thead>
<tr>
<th>Aid</th>
<th>IncomeRank</th>
<th>City</th>
<th>County</th>
<th>Crimerank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>131 782</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>Manchester</td>
<td>Greater Manchester</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>925</td>
</tr>
</tbody>
</table>

Table 5.7: Area Info with subsumed tuples

Based on the definition of subsumed tuples, the first tuple $t_1$:

$$
\begin{array}{cccc}
1 & 131 782 & Manchester & Lancashire \\
\end{array}
$$

subsumes the second tuple $t_2$:

$$
\begin{array}{cccc}
1 & ⊥ & Manchester & Lancashire \\
\end{array}
$$

as $t_1$ and $t_2$ have the same schema (belong to the same relation), $t_2$ has equal values for the non-null values in attributes $Aid$, $City$, $County$ as $t_1$ and nulls in the same positions, e.g., in $Crimerank$; while $t_2$ has one more null in $IncomeRank$ than $t_1$, thus $t_2$ is subsumed by $t_1$.

The third tuple $t_3$:

$$
\begin{array}{cccc}
1 & ⊥ & Manchester & Greater Manchester \\
\end{array}
$$

has similar data to $t_1$ and $t_2$, but the value on $County$ attribute is not the same, thus, it cannot be concluded that it is subsumed by either of these.

The fourth tuple $t_4$:

$$
\begin{array}{cccc}
1 & ⊥ & Manchester & Lancashire 925 \\
\end{array}
$$

coincides with the data in $t_1$ on some non-null values, but $t_4$, on $Crimerank$, has a non-null value, e.g., 925, where in $t_1$ it is null, thus $t_1$ and $t_4$ complement each other, so the output mapping will not discard either of these based on this information as this particular fact is not in the scope of this thesis. The tuple complementarity problem is studied in Bleiholder et al. (2010) as a data fusion problem, which can exist as a subsequent step after mapping generation in a data integration work-flow. In this situation, it can be concluded that $t_4$ subsumes $t_2$. 
Motivation. Our aim is to devise an algorithm that generates mappings that satisfy as much as possible the target constraints given the source data. By eliminating subsumed tuples, the generated mappings may come to satisfy candidate key constraints, whose attributes would otherwise contain duplicate values.

Prior work on detecting subsumed tuples, e.g., Galindo-Legaria (1994); Bleiholder et al. (2010), is not conducted in the context of mapping generation, i.e., on non-materialized data, so they make the assumption that the data is already generated and that analysis on the whole relation can be done. In our context of mapping generation, the data is not (yet) materialized. Thus, we propose a method to modify the mappings such that, when executed, the subsumed tuples are not produced for the initial target relations.

Chosen approach. Our algorithm builds on the projections of the modified UTR mappings (for the generation of labelled nulls) such that subsumed tuples are discarded when the mappings for the initial target tables are materialized. This step is shown in Figure 5.2 as the third post-processing step and as Beta in line 10 in Algorithm 7. Here, we describe how to alter the mappings so that they do not produce subsumed tuples. We consider a tuple to be subsumed if it satisfies the conditions in Bleiholder et al. (2010).

Consider the same generic example as in Section 5.4.2, where there are four target tables with the same schema constraints:

\[ T_1(a_1, a_2, ..., a_n), \]
\[ T_2(x_1, a_{1fk}, x_2, ..., x_m), \]
\[ T_3(y_1, x_{1fk}, y_2, ..., y_p), \]
\[ T_4(z_1, x_{1fk}, z_2, ..., z_q). \]

Their UTR mapping is modified to include skolemization, but, for simplicity, we ignore this detail below:

\[ map_{UTR}(a_1, a_2, ..., a_n, x_1, x_2, ..., x_m, y_1, y_2, ..., y_p, z_1, z_2, ..., z_q). \]

To avoid producing subsumed tuples, the following method is used:

For a target table \( T_i, i \in \{1, 2, 3, 4\}, \) with \( T_i.X \) its set of attributes, the mapping is generated following the steps:

1. Retain in a new mapping the projected UTR mapping on the attributes of \( T_i, \) i.e., \( map_{T_i} \leftarrow \pi_{T_i.X}(map_{UTR}) \)
2. Modify the newly created mapping \( map_{T_i} \) such that it does not produce the subsumed tuples, i.e., \( map_{T_i} \leftarrow \beta(map_{T_i}) \). We define \( \beta \) later in this section.

The generation of the mappings for each of the target tables is independent from
one another, thus, the order in which these mappings are generated is not essential and these steps could run in parallel if needed.

**Example 5.4.6.** Consider the example in Figure 5.1, the UTR for the two target tables as expressed in Example 5.3.1, and the projected UTR mapping for Area Info:

\[
\text{map}_{AI} \leftarrow \pi_{\text{Aid,IncomeRank,City,County,Crimerank}}(\text{map}_{\text{UTR}})
\]

For simplicity, let us consider the tuples produced by \(\text{map}_{AI}\) as the ones in Table 5.7. To eliminate the subsumed tuples, the algorithm modifies \(\text{map}_{AI}\) to

\[
\text{map}_{AI} \leftarrow \beta(\text{map}_{AI}), \quad \beta(\text{map}) \text{ is defined as:}
\]

\[
\beta(\text{map}) = \{t \mid t \in \text{map}, X \in \text{schema}(\text{map}), \exists t' \in \text{map} \text{ s.t. } |\text{nulls}(t')| \leq |\text{nulls}(t)| \land t[X] = t'[X] \land t[X] \neq \text{null}\}
\]

In other words, if the mapping resulting from \(\beta(\text{map})\) is executed, then for each tuple \(t\) that it outputs, there are no other tuples \(t'\) with less nulls and with equal values on the non-null attributes.

\(\text{map}_{AI}\) represents the final mapping, whereby Area Info can be populated.

### 5.5 Mapping Characteristics

**Motivation.** Sections 4.5.2 and 4.5.3 show that *mapping generation over autonomous sources* means that the sources might not merge as well as they would in a well-behaved setting because one cannot count on correct and complete source schema constraints for them. Based on the inferred (possibly relaxed) foreign keys, the source data may only partially merge. The consequence is that the target constraints such as (candidate) keys and foreign keys may not be satisfied by the tuples generated by the mappings. In aiming to satisfy target constraints with data from autonomous and heterogeneous sources, the mappings need to be ranked according to how well the data they would yield populates the target schema. In this chapter, we have described two methods for addressing the challenge of generating mappings that satisfy the target constraints. However, these may not be enough and the target constraints can still be violated. We propose, therefore, a set of characteristics for the mappings that aim to reflect the degree to which the mappings are likely to satisfy the target constraints.

**Prior work.** There has been prior work on characterizing mappings [Gottlob and Senellart (2010); Alexe et al. (2011a,b)]. In Gottlob and Senellart (2010),
instance-level data is used in a search for optimal schema mappings based on the structure and occurrences of constants in the instances. They describe a cost function for schema mappings that has different criteria such as validity, explanation, zero-repair, etc, where each criterion is a numerical value. The computation of these characteristics is based on the source and target instances and measures the number of repairs that are needed to correctly transform the data from the source to the target format. Similarly, Alexe et al. (2011a,b) propose an approach that uses two kinds of evidence: instance-level data and user feedback for refinement. They consider a set of mappings expressed as source-to-target \( tgds \) and a set of data examples, and they characterize these mappings in terms of a finite set of positive and negative data examples of what the mapping can generate. These proposals rely on the fact that, in a data integration workflow, the mapping generation has ended and the data their output can be materialized and compared with a set of ground-truth tuples. However, in our setting, mappings need to be characterized without being materialized as Dynamap can generate thousands of mappings making it infeasible to materialize them just for computing the characteristics.

Mapping characteristics. In this thesis, we contribute a set of mapping characteristics that are computed without materializing the data. They are based instead on the propagated profile data, taking into account the methods of generating labelled nulls and discarding subsumed tuples described in this chapter. They are:

1. Attribute completeness is quantified with the percentage of non-nulls in an attribute.
2. Source-extracted values ratio is the fraction of the values of a target attribute that come from the data sources, i.e., non-null values that are not labelled nulls.
3. Key consistency is a score that measures the extent to which the unique constraints are satisfied on the target attributes that are expected to be candidate keys. If the key consistency score indicates that the attribute satisfies the key, then foreign keys are satisfied to the same extent as both the referenced and the dependent attributes draw values from the same \( UTR \) attribute.
5.5. MAPPING CHARACTERISTICS

5.5.1 Attribute Completeness

The purpose of this criterion is to show how complete the attributes are. The attribute completeness score measures the ratio between non-null values and nulls. The non-null values can be either source-extracted or labelled nulls generated through skolemization (as described in Section 5.4.2).

The completeness score $F_C$ is computed by taking into consideration the mapping and its propagated profile and metadata information.

Computing the attribute completeness score for a mapping $m$:

**Input**
- the (estimated) number of tuples in the mapping result ($|m|$)
- the (estimated) number of nulls for attribute $m.X$ ($nulls(m.X)$). For computing the estimated score, the number of nulls is the one estimated before altering the UTR mappings to create labelled nulls.

**Attribute completeness function** for attribute $m.X$

$$F_C(m.X) = \begin{cases} 
1 - \frac{nulls(m.X)}{|m|}, & \text{if } m.X \text{ is not (part of) a candidate key in the target} \\
1.0, & \text{otherwise}
\end{cases}$$

**Output**

Let $c_X \leftarrow F_C(m.X)$, $c_X \in [0.0, 1.0]$, be the attribute completeness score for the input attribute $m.X$. If $c_X = 1.0$, it means that all values are non-nulls, otherwise we say $m.X$ contains $(c_X \times 100)\%$ non-nulls. The score $c_X$ is always equal to 1.0 if $m.X$ is (part of) a candidate key because the remaining nulls in $m.X$ are replaced with labelled nulls (after altering the UTR mapping), so $m.X$ contains mixed non-null data (skolems and source-extracted). Dynamap$^X$ detects this and the score becomes 1.0.

**Example 5.5.1.** For a mapping $m$ of the UTR for relations Area_ID and UK Realestate (see Example 5.3.1), assume the data produced by $m$ has the following metadata information:

1. $nulls(Aid) = 0$
2. $nulls(Pid) = 4$
3. $nulls(IncomeRank) = 3$
4. $|m| = 4$

The example tuples in Table 5.8 could represent the data produced by $m$.

In the target schema, attributes Aid and Pid are primary keys, thus, Dynamap$^X$ detects that these are populated with labelled nulls if there is missing information.
CHAPTER 5. MAPPING GENERATION FOR A COMPLEX TARGET

Aid IncomeRank City County Crime rank Pid Postcode Street Price

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Manchester</th>
<th>Lancashire</th>
<th>⊥</th>
<th>⊥</th>
<th>M3 1NN</th>
<th>Mirabel St.</th>
<th>725pcm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>782</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>⊥</td>
<td>⊥</td>
<td>M3 1NN</td>
<td>Mirabel St.</td>
<td>800.00</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>Manchester</td>
<td>Greater Manchester</td>
<td>⊥</td>
<td>⊥</td>
<td>M3 1NN</td>
<td>Mirabel St.</td>
<td>950</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>925</td>
<td>⊥</td>
<td>M3 7NG</td>
<td>100 Greengate</td>
<td>1300 pcm</td>
</tr>
</tbody>
</table>

Table 5.8: UTR example tuples

Table 5.9: UTR example tuples – with labelled nulls

in them. Hence, it detects that after mapping $m$ is modified, it outputs the tuples in Table 5.9.

For attribute completeness, the algorithm computes:

- for Aid: the number of nulls is 0, so $c_{Aid} = 1.0$.
- for Pid: the number of nulls is 4, so, according to the formula on Pid, $F_C(Pid)$, $c_{Pid} = 0.0$. However, Pid is expected to be a candidate key in the target schema, so the algorithm detects that the modified mapping generates skolems for this attribute. Given these satisfied conditions, $c_{Pid}$ becomes $c_{Pid} = 1.0$.
- for IncomeRank: the number of nulls is 3, so $c_{IncomeRank} = 1 - \frac{3}{4} = 0.25$

The attribute completeness scores for Aid and Pid attributes that are expected to be candidate keys in the target is 1.0, i.e., they are fully populated in the target. On the other hand, IncomeRank is not expected to be (part of) a candidate key in the target, thus, its completeness score remains 0.25, meaning that a quarter of its values are expected to be non-nulls.

5.5.2 Source-extracted Values Ratio

The purpose of this criterion is to show how much source data makes it to the populated target attributes. The source-extracted values ratio measures the fraction of non-null source-extracted values in the attribute. The difference between attribute completeness and source-extracted values ratio is that the measured non-null values can only be source-extracted (i.e., labelled nulls are not counted).

Similarly to attribute completeness, the source-extracted values ratio $F_S$ is computed by considering both the mapping and its propagated profile and metadata information:
5.5. MAPPING CHARACTERISTICS

Input
- the (estimated) number of tuples in the mapping result (|m|)
- the (estimated) number of nulls for attribute m.X (nulls(m.X)). For computing the estimated scores, the number of nulls is the one estimated before altering the UTR mappings to generate labelled nulls.

Source-extracted values ratio function for attribute m.X
\[ F_S(m.X) = 1 - \frac{\text{nulls}(m.X)}{|m|} \]

Output
Let \( s_X \leftarrow F_S(m.X) \), \( s_X \in [0,1.0] \), be the source-extracted values ratio for the input attribute m.X. We say m.X contains \((s_X \times 100)\%\) source-extracted values, e.g., \( s_X = 1.0 \) (all values are source-extracted) if the estimated nulls number is 0.

Example 5.5.2. Continuing Example 5.5.1, the algorithm computes the source-extracted values ratio for each of the attributes in the UTR mapping m as follows:
- for Aid: the number of nulls is 0, so \( s_{Aid} = 1.0 \).
- for Pid: the number of nulls is 4, so \( s_{Pid} = 0.0 \).
- for IncomeRank: the number of nulls is 3, so \( s_{IncomeRank} = 1 - \frac{3}{4} = 0.25 \).

In this case, even if Pid is expected to be a candidate key in the target schema and the algorithm detects that the modified mapping generates skolems for this attribute, the generated labelled nulls are not source data, so the source-extracted values ratio has a lower value than the attribute completeness score. For instance, for Pid, the attribute completeness is \( c_{Pid} = 1.0 \), while the source-extracted values ratio is \( s_{Pid} = 0.0 \). This is due to the fact that Pid only contains labelled nulls (as shown in Table 5.9). On the other hand, for Aid and IncomeRank the two scores coincide.

5.5.3 Key Consistency

The key consistency criterion is not focused on the quality of the data, but on the violation of schema constraints. This criterion shows to which degree the (set of) attribute(s) is expected to violate the key constraints in the target.

In using a UTR, foreign key attributes are represented by the same UTR attribute. Thus, if a UTR attribute has a high score on key consistency and that attribute is part of a foreign key relationship, then it can be concluded that the foreign key relationship is likely to be satisfied given that the referenced attribute is (likely to be) a key and the full inclusion dependency condition is satisfied given
that both attributes draw their values from the same UTR attribute.

The key consistency score \( F_K \) is computed by taking into consideration the mapping and its propagated profile and metadata information:

**Input**
- the (estimated) number of tuples in the mapping result \( |m| \)
- the (estimated) number of distinct values for attribute \( m.X \) \( (V(m.X)) \). For computing the estimated score, the number of distinct values is the one estimated before altering the UTR mappings to create labelled nulls.

**Key consistency function** for attribute \( m.X \)

\[
F_K(m.X) = \begin{cases} 
\text{nulls}(m.X) + V(m.X) \over |m|, & \text{if } m.X \text{ is (part of) a candidate key in the target} \\
V(m.X) \over |m|, & \text{otherwise} 
\end{cases}
\]

**Output**
Let \( k_X \leftarrow F_K(m.X) \), \( k_X \in [0.0, 1.0] \), be the key consistency score for the input attribute \( m.X \). If \( m.X \) is (part of) a candidate key then Dynamap\(^X\) detects that, when the mapping is materialized, the remaining nulls in \( m.X \) are replaced with labelled nulls (after altering the UTR mapping), so \( m.X \) contains mixed non-null data (skolems and source-extracted). The key consistency on the column is \( \frac{\text{nulls}(m.X) + V(m.X)}{|m|} \) as the nulls in \( m.X \) become labelled nulls, e.g., if \( \text{nulls}(m.X) + V(m.X) = |m| \), then \( k_X \) becomes \( k_X \leftarrow 1.0 \) as the non-null distinct (source-extracted) values together with the created skolems fully populate column \( m.X \) with distinct values. If \( m.X \) is not (part of) a candidate key, we say \( m.X \) contains \( (k_X \times 100)\% \) unique values, e.g., \( k_X = 1.0 \) (all values are unique) if the number of estimated distinct values is the same as the estimated cardinality of the mapping \( m \); or \( k_X = 0.0 \) means that the number of distinct (source-extracted) values \( V(m.X) = 0 \) and no labelled nulls populate it.

**Example 5.5.3.** Continuing Examples 5.5.1 and 5.5.2, the algorithm computes the score for key consistency for each attribute of mapping \( m \) that is expected to be a key as follows:

- for \( \text{Aid} \): the number of distinct values is 1. Dynamap\(^X\) detects that the \( \text{Aid} \) column will be padded with labelled nulls so \( k_{\text{Aid}} = \frac{\text{nulls}(\text{Aid}) + V(\text{Aid})}{|m|} = \frac{0 + 1}{4} = 0.25 \).

- for \( \text{Pid} \): the number of distinct values is 0. Dynamap\(^X\) detects that all nulls in \( \text{Pid} \) column become labelled nulls, so \( k_{\text{Pid}} = \frac{\text{nulls}(\text{Pid}) + V(\text{Pid})}{|m|} = \frac{4 + 0}{4} = 1.0 \).

After discarding the subsumed tuples in Table 5.9 for each initial target table, the data in the two target tables will look as in Tables 5.10 and 5.11.
5.6. ALGORITHM EVALUATION

Using key consistency scores, Dynamap\textsuperscript{X} is able to detect whether foreign key constraints are guaranteed to be satisfied or not. In this scenario, there is one foreign key, $\text{UKR.Aid} \rightarrow \text{AI.Aid}$. Given that the key consistency score for $\text{AI.Aid} = 0.25$, it detects that $\text{AI.Aid}$ has duplicates. However, the score shows only a pessimistic approximation of how many duplicates are in the materialized tuples as, through subsumption, some tuples may be discarded. This is the case in this scenario, i.e., one tuple (the second tuple in Table 5.9) from Area Info is discarded as it is subsumed by both the first tuple and the last one (as explained in Example 5.4.5). The algorithm can predict that, although the full inclusion dependency requirement is satisfied, the foreign key constraint is violated as the referenced attribute is not a candidate key.

### 5.6 Algorithm Evaluation

In this chapter, so far we have described Dynamap\textsuperscript{X}, an extension of Dynamap that can handle scenarios where target schemas have multiple tables related by schema constraints. In this section, we evaluate the performance of Dynamap\textsuperscript{X} on two types of scenarios: (i) synthetic scenarios created with a state-of-the-art scenarios generator (\textsc{iBench}); and (ii) two real-world scenarios that are variations on the scenarios in Sections 4.5.2 and 4.5.3.

**Experimental setup.** As with the previous experiments, we compare the results of Dynamap\textsuperscript{X} with the results of ++Spicy. Dynamap\textsuperscript{X} and ++Spicy are run over the same data sources, and the same target schemas. For storage, we used PostgreSQL 9.6. In the case of the real-world scenarios, in order to keep the focus on mapping generation, matches were hand-crafted by a human expert. Profiling

---

### Table 5.10: Area Info tuples (populated initial target table)

<table>
<thead>
<tr>
<th>Aid</th>
<th>IncomeRank</th>
<th>City</th>
<th>County</th>
<th>Crimerank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>782</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>√</td>
</tr>
<tr>
<td>1</td>
<td>√</td>
<td>Manchester</td>
<td>Greater Manchester</td>
<td>√</td>
</tr>
<tr>
<td>1</td>
<td>√</td>
<td>Manchester</td>
<td>Lancashire</td>
<td>925</td>
</tr>
</tbody>
</table>

### Table 5.11: UK Realestate tuples (populated initial target table)

<table>
<thead>
<tr>
<th>Pid</th>
<th>Aid</th>
<th>Postcode</th>
<th>Street</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{id}(1</td>
<td>M31NN</td>
<td>MirabelSt,</td>
<td>725pcm)</td>
<td>1</td>
</tr>
<tr>
<td>$p_{id}(1</td>
<td>M31NN</td>
<td>MirabelSt,</td>
<td>800.00)</td>
<td>1</td>
</tr>
<tr>
<td>$p_{id}(1</td>
<td>M31NN</td>
<td>MirabelSt,</td>
<td>950)</td>
<td>1</td>
</tr>
<tr>
<td>$p_{id}(1</td>
<td>M37NG</td>
<td>100Greengate,</td>
<td>1300pcm)</td>
<td>1</td>
</tr>
</tbody>
</table>
data was generated through two *Metanome* modules, i.e., HyUCC [Papenbrock and Naumann (2017)] for generating candidate keys and Sindy [Kruse et al. (2015)] for generating (partial) 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 a (full) inclusion dependency with an attribute from another relation then a foreign key is inferred. In the case of the synthetic scenarios, the matches, the profile data, the data sources, and the target schema are generated automatically (without human input) by the generator. Tuples in the synthetic scenarios were generated using *Datafiller* [Coelho (2013)]. The experiments were run over an Intel Core i5 with 2×2.7GHz, and 8GB of RAM.

**Evaluation metrics.** As in previous chapters, we compare the output tuples of the mappings over the same input sources. In these scenarios, the focus is on generating mappings for target tables that have constraints. For this reason, generating data that is correlated between the tables is essential. Taking this into consideration, the comparison between the ground-truth tuples and the output mappings is based on the outerjoin of the base target tables. We decided to conduct the comparison in this manner as each algorithm may produce labelled nulls for the purpose of keeping the tuples correlated and we need to avoid comparing differently generated synthetic data (skolems) as each mapping algorithm does it differently and still be correct. Our focus is on comparing correctly correlated source data.

**iBench scenarios.** For the iBench scenarios we used the same metrics as in Section 4.5.1 and previously used in Arocena et al. (2015), viz., the number of constants and the number of nulls produced by the mappings were computed. In Arocena et al. (2015), mappings that output fewer constants and fewer nulls are considered to be desirable. The intuition behind this metric is that if the data is correlated as best as possible, then there are no (or few) redundant tuples, while mappings that do not correlate (i.e., join) tables will create many duplicate values and undesirable nulls. Thus, if the number of constants and nulls is minimal then it means that the data has been correlated.

**Real-world scenarios.** As in the previous chapters, for the two experiments with real-world data, we compare the data output by Dynamap\(^X\) mappings, and that returned by ++Spicy mappings with the data in the ground-truth mappings. We analyze two aspects on the generated data:

1. **Data correlation:** We ascertain whether the source data remains correlated
across the target tables (i.e., the relationships between the data values are not lost once the target is populated). The purpose here is to detect whether the mappings generated by the two systems maintain the source data linked through foreign key values across multiple target tables. In the experiments in Chapters 3 and 4, the target contains one relation only so all the source data that is correctly correlated is transferred in the same target tuple. For a target schema with multiple tables with foreign key constraints, data correlation must be measured across all target relations. We do this by performing a full outer join on the tables that result from each of the three sets of mappings, i.e., Dynamap$^X$, ++Spicy, and the ground-truth mapping. We then perform the comparison between the outer-joined tables of the two systems against the outer-joined ground-truth tables. We define the notions of true positive, true negative, false positive, and false negative attribute/tuple values in the same way as we did for experiments in Sections 4.5.2 and 4.5.3.

2. **Mapping characteristics:** We compare the (estimated) attributes scores (i.e., attribute completeness, source-extracted values ratio and key consistency per each attribute) on the data produced by the Dynamap$^X$, ++Spicy, and the ground-truth mappings. This serves three purposes:
   (a) To determine the accuracy of the attribute scores that Dynamap$^X$ estimates for the non-materialized mappings. We do this by comparing the estimated scores with the real scores for the materialized data.
   (b) To determine how well Dynamap$^X$ performs against the state-of-the-art mapping generation system in terms of populating the target attributes and satisfying the target constraints. We do this by comparing its attributes scores on the materialized data with the scores for the data materialized by the ++Spicy mappings.
   (c) To determine how well Dynamap$^X$ performs against the ground-truth in terms of populating the target attributes and satisfying the target constraints. We do this by comparing its scores on the materialized data with the scores for the ground-truth data. We consider the ground-truth mappings as the best-effort solution that an expert can design without using intermediate materialized relations. Thus, if the ground-truth data cannot satisfy the target constraints, we say that the source data does not allow the generation of mappings that satisfy them.
5.6.1 Benchmark Experiment - iBench

Motivation. As explained in Section 3.8.1, iBench is a tool that generates data integration/exchange scenarios where the sources and target have explicit keys and foreign keys. These scenarios are relevant for our purpose as they make use of a variety of data integration primitives that mapping generation algorithms must (ideally) be able to tackle.

In the experiments in Sections 3.8.1 and 4.5.1, we have shown that Dynamap, as described and evaluated in, respectively, Chapters 3 and 4, is able to tackle input scenarios that are complex on the sources (e.g., multiple join patterns), but have only one target relation. In this chapter, the aim of the experiments is to show that Dynamap can tackle the scenarios that Dynamap was (as expected) not able to tackle, viz., vertical partitioning and variations thereof. As explained in Section 3.8.1, the only type of primitive that Dynamap is not able to handle as expected is when a source relation is split into two target relations that are linked by a foreign key. In such cases, Dynamap would generate mappings that populate but not link the two target relations since it is not designed to consider the foreign keys in the target.

Scenarios. We have tried to reproduce as closely as possible one of the experiments in the iBench paper [Arocena et al. (2015)], where they vary the number of foreign keys in the target schema using different primitives for vertical partitioning: VH, VI, and VNM primitives. In Arocena et al. (2015), they describe these primitives as part of Ontology scenarios because they involve primitives that mimic IS-A, M-to-N, and HAS-A relationships. In the experiments, we use the same primitives that they use for building the Ontology scenario. These are the following:

1. (ADD) Add-attribute transformation:
   \[ R(a, b) \rightarrow T(a, b, f(a, b)) \]

2. (VH) Vertical partitioning in a HAS-A relationship:
   \[ R(a, b) \rightarrow S(f(a), a) \land T(g(a, b), b, f(a)), \text{ where } T.f(a) \rightarrow S.f(a) \text{ is a FK} \]

3. (VI) Vertical partitioning in a IS-A relationship:
   \[ R(a, b, c) \rightarrow S(a, b) \land T(a, c), \text{ where } T.a \rightarrow S.a \text{ and } S.a \rightarrow T.a \text{ are Fks} \]

4. (VNM) Vertical partitioning in N-to-M relationship:
   \[ R(a, b) \rightarrow S_1(f(a), b) \land M(f(a), g(b)) \land S_2(g(b), b), \text{ where } M.f(a) \rightarrow S_1.f(a) \text{ and } M.g(b) \rightarrow S_2.g(b) \text{ are Fks} \]

The scenarios are built as follows.
5.6. ALGORITHM EVALUATION

Figure 5.4: Dynamap compared to ++Spicy on iBench scenarios (rich target schemas)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Transformation</th>
<th>As Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>$R(a,b) \rightarrow T(a,b)$</td>
<td>Yes</td>
</tr>
<tr>
<td>HP</td>
<td>$R(a,b) \rightarrow T_1(b), R(a,b) \rightarrow T_2(b)$</td>
<td>Yes</td>
</tr>
<tr>
<td>ADD</td>
<td>$R(a,b) \rightarrow T(a,b,f(a,b))$</td>
<td>Yes</td>
</tr>
<tr>
<td>DEL</td>
<td>$R(a,b) \rightarrow T(a)$</td>
<td>Yes</td>
</tr>
<tr>
<td>ADL</td>
<td>$R(a,b) \rightarrow T(a,f(a))$</td>
<td>Yes</td>
</tr>
<tr>
<td>ME</td>
<td>$R(a,b) \land S(b,c) \rightarrow T(a,b,c)$</td>
<td>Yes</td>
</tr>
<tr>
<td>MA</td>
<td>$R(a,b) \land S(b,c) \rightarrow T(a,b,c,f(a,b,c))$</td>
<td>Yes</td>
</tr>
<tr>
<td>VP</td>
<td>$R(a,b) \rightarrow S_1(f(a,b),a) \land S_2(f(a,b),b), S_1(f(a,b),b)$ references $S_2(f(a,b),b)$ and vice versa</td>
<td>Yes</td>
</tr>
<tr>
<td>VH</td>
<td>$R(a,b) \rightarrow S_1(f(a,b),a) \land T(g(a,b),b,f(a))$</td>
<td>Yes</td>
</tr>
<tr>
<td>VI</td>
<td>$R(a,b,c) \rightarrow S_2(f(a,b),c)$</td>
<td>Yes</td>
</tr>
<tr>
<td>VNM</td>
<td>$R(a,b) \rightarrow S_1(f(a,b),a) \land M(f(a),g(b)) \land S_2(g(b),b)$</td>
<td>Yes</td>
</tr>
<tr>
<td>SU</td>
<td>$R(a,b) \rightarrow T(f(a,b),b,g(b))$</td>
<td>Yes</td>
</tr>
<tr>
<td>SJ</td>
<td>$R(a,b,c) \rightarrow S(a,b), R(a,b,c) \land R(b,d,e) \rightarrow T(a,b)$, and $R.b$ references $R.a$</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.12: Results for iBench primitives

Target schema. We set the number of target tables to 40, so regardless of the combination of primitives, there are always that many target relations. We set the corresponding iBench parameters to reuse 0% of the target schema, i.e., every primitive creates new target relations. The arities of the target relations vary between 3 and 7 attributes. For each iBench scenario, we varied the number of target constraints (foreign keys) from 0% to 50%. This means that the scenarios shift from an ADD-dominant scenarios (the majority of the primitives are ADD) to VP-dominant scenarios (the majority of the target relations are generated by the VH, VI and VNM primitives).

Input sources. Given that the number of target relations is fixed to 40 and that the primitives vary, the number of source relations will change as well, as the number of primitives causes the variation: from 40 source relations (with 0% target INDs) to 28 relations with (50% target INDs). The arities range from 3 to 7 attributes and each relation has 400-600 tuples that are generated with
Datafiller [Coelho (2013)].

Matches. All sources 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 shown in Figure 5.4, where the output values of the DynamapX mappings are compared with those of the ++Spicy mappings. The two algorithms perform identically in terms of number of constants and nulls (to avoid clutter, there are no figures on the plot, but the numbers are equal in all situations). Their performance 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 DynamapX were identical with the corresponding ones of ++Spicy. The only inessential difference is in the format of labelled nulls.

Analyzing the results in this experiment, along with the results described in Sections 3.8.1 and 4.5.1, one can say that DynamapX can tackle all primitive cases (proposed by iBench) that might occur in mapping generation scenarios where user input is not essential for further guidance. The iBench primitives is presented in Table 5.12. There is one primitive, viz., self-join, that DynamapX does not tackle but this type of scenario is not expected to happen in the context of which DynamapX was developed, viz., in the wild. We have explained in Section 3.8.1 why we believe that this kind of scenario is an isolated case in mapping generation where further human input is essential, input on which one cannot rely when the mapping generation context is over large repositories of autonomous data sources.

5.6.2 Real-world Experiment - Real-estate Domain

Motivation. As in the experiment in Section 4.5.2, DynamapX is run over real-world data from autonomous sources to populate a single-relation target. Here, we evaluate how DynamapX performs on a scenario where the target has many relations subject to multiple key and foreign key constraints that their instances must try to satisfy. Our motivation for this experiment is to show how DynamapX
performs on real-world data, considering the following aspects:

i) *data correlation:* to analyze how well Dynamap$^X$ is able to merge the sources and transfer the information into a target schema with constraints, while maintaining the data correctly correlated across the foreign key relations;

ii) *mapping characteristics:* to use the mapping characteristics (which we define at attribute level, as scores for completeness, source-extracted ratio, and key consistency), as means to measure:

(a) the accuracy of the scores that Dynamap$^X$ computes (as described in Section 5.5, including the likelihood of the constraints’ satisfaction);

(b) how well Dynamap$^X$ performs in terms of populating the target relations in comparison to ++Spicy, and to the ground-truth data, i.e., the best-effort solution constructed by hand for this mapping generation problem.

**Scenario.** The purpose of this scenario is to generate a set of mappings that associate crime statistics with properties web-extracted from real-estate websites, where the target consists multiple relations linked into a chain by foreign keys.

**Target schema.** The target schema is depicted in Figure 5.5, and contains four relations with the following schema constraints:

- P.Pid, CR.postcode, S.street_id, and C.city_id are primary keys for their corresponding relations,
- P.postcode \(\rightarrow\) CR.postcode foreign key
- CR.street_id \(\rightarrow\) S.street_id foreign key
- S.city_id \(\rightarrow\) C.city_id foreign key

**Input sources.** In this section, we investigate how Dynamap$^X$ performs on the same real-world sources, where web-extracted datasets from the real-estate domain are combined with data from the UK open-government data portal, and open-government datasets that contain information about English indices of deprivation that measure the relative deprivation in small areas in England and freely available open-government addresses data. Details about the input datasets are the same

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data for the target</th>
<th>#Sources</th>
<th>Arity</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manchester real-estate</td>
<td>s_name, price, c_name, postcode</td>
<td>5</td>
<td>5-9</td>
<td>20-171</td>
</tr>
<tr>
<td>London real-estate</td>
<td>s_name, price, c_name, postcode</td>
<td>2</td>
<td>6-13</td>
<td>20-35</td>
</tr>
<tr>
<td>Oxford real-estate</td>
<td>price, postcode, s_name</td>
<td>4</td>
<td>10-14</td>
<td>28-192</td>
</tr>
<tr>
<td>Manchester deprivation</td>
<td>postcode, crimerank</td>
<td>1</td>
<td>28</td>
<td>391</td>
</tr>
<tr>
<td>London deprivation</td>
<td>postcode, crimerank</td>
<td>1</td>
<td>28</td>
<td>54</td>
</tr>
<tr>
<td>Manchester &amp; Oxford addresses</td>
<td>postcode, s_name, c_name</td>
<td>1</td>
<td>4</td>
<td>235</td>
</tr>
</tbody>
</table>

Table 5.13: Web-extracted and open-government datasets
as in Table 4.1, but the matched data is different. Table 5.13 provides information on the sources with the updated matches for the changed target schema.

**Matches.** Given that the focus is on mapping generation, the matches that were given as input were created by hand, thus making sure that the matches are correct and that mapping generation is not hindered by faulty matches. Note in Table 5.13 (second column), that not all target attributes are matched by the sources. The target attributes that are not matched are:

1. for relation P: Pid,
2. for relation CR: street_id,
3. for relation S: street_id, city_id,
4. for relation C: city_id, population.

Nevertheless, each source contributes to the target schema by way of at least two matched attributes.

**Profile data.** The profile data on the input sources is the same as for the previous experiment (as the sources do not change). It contains 68 candidate keys, 1,734 partial inclusion dependencies, and 510 full inclusion dependencies.

**Ground truth.** The ground-truth mappings were created by hand as follows. Given that it is difficult to create separate mappings for each target relation and then try to reconcile the data so as to maintain the data correlations according to the target constraints, we decided to group all attributes in a compound relation. Then, a mapping was created such that the sources were merged as best as possible considering their attribute-value overlaps. The mapping created for the compound relation unions the real-estate data from Manchester real-estate and Oxford real-estate, which is then merged through full outer joins with Manchester deprivation data and Manchester & Oxford addresses. The same is done with the London data: the London real-estate data is unioned, and then outer joined with London deprivation. This result is unioned to the data from Manchester and Oxford. Then, for the unmatched attributes that have constraints, e.g., Pid, street_id and city_id (excluding C.population attribute), we created hash values based on the remaining values in the corresponding target relations. Nevertheless,
the procedure for creating these synthetic values is not essential as it can differ from algorithm to algorithm and different methods can be considered as being correct as long as the source data is not tampered with when transferred. The hash values are important, though, for creating common values between foreign key attributes such that the tuples in different tables are still correlated to one another. Lastly, we populated the initial target tables with projections on the view that the mapping for the compound table creates.

Comparison. Dynamap$^X$ and ++Spicy were run over the same mapping task with the same input sources and target schema. The respective output mappings were executed and the output tuples were used to materialize the target tables. As stated above, we compare the two systems on data correlation and on mapping characteristics, and we do so as follows:

i) For data correlation, we ascertain whether the data in the target is still correlated. We performed a full outer join between the four tables on all three materialized versions, i.e., the ground-truth data, the data produced by Dynamap$^X$ mappings, and the data produced by ++Spicy. Then, we compared the outer-joined tuples of the ground truth tables, with the outer-joined tuples in the tables produced by the Dynamap$^X$ and ++Spicy mappings. The quality of the results is evaluated in the same manner as described in Section 4.5, i.e., at attribute level and at tuple level. The labelled nulls that were produced by either approach were not considered in the comparison as their purpose is only to correlate the tuples, i.e., we compare only source data.

ii) For mapping characteristics, we measure how Dynamap$^X$ performs in terms of populating the target attributes and satisfying the target constraints (i.e., attribute completeness, source-extracted values ratio and key consistency per each attribute), as follows:

(a) We determine the accuracy of the attribute scores that Dynamap$^X$ estimates for the unmaterialized mappings by comparing the estimated scores with the real scores for Dynamap$^X$ materialized data.

(b) We determine how well Dynamap$^X$ performs against ++Spicy in terms of populating the target attributes and satisfying the target constraints by comparing the attributes scores on Dynamap$^X$ materialized data with the scores for the data obtained through ++Spicy.

(c) We determine how well Dynamap$^X$ performs against the ground truth
in terms of populating the target attributes and satisfying the target constraints by comparing the attribute scores in Dynamap\textsuperscript{X} materialized data with the scores that we compute for the ground-truth data.

These characteristics can reveal any differences between the strategies the two algorithms pursue when it comes to generating mappings that aim to satisfy target constraints.

Results. The results are described in terms of data correlation and on the mapping characteristics of the output mappings.

For generating the mappings, ++Spicy ran in 10 min, 7 sec, 728 ms, and Dynamap\textsuperscript{X} ran in 0 min, 7 sec, 458 ms. For populating the target tables, ++Spicy materialized 119 intermediate tables, while Dynamap\textsuperscript{X} does not need to materialize any.

Results on data correlation. The results are shown in Figures 5.6(a) and 5.6(b) at attribute level and at tuple level, respectively.

In this scenario, the mapping generated by Dynamap\textsuperscript{X} first merges the deprivation sources from London and Manchester using a union. Then the Oxford data is outer joined with the addresses data set since its result becomes schema-compatible with the real-estate data from Manchester and London. The output of the deprivation datasets is further merged through full outer join with the unioned data from all real-estate agencies, i.e., Oxford, Manchester, and London. Although this mapping is not the same as the ground-truth mapping, it yields almost all tuples as expected. In fact, given that the postcode attribute was used to correlate tuples from the output to tuples in the ground truth, the only output tuples that were not considered to be as expected are those that did not have data (or had labelled nulls) on the postcode values, thus, they were not compared to any ground-truth tuples as no ground-truth tuple could be assigned to them.

On the other hand, ++Spicy uses egds as it detects that postcode needs to be a primary key in crimerank info target relation. The manner in which ++Spicy uses the egds was explained in Section 2.1.4. In this scenario, for generating the mappings, it produces all possible merges on the postcode values for all relations that have a match to that attribute (which is all sources in this situation). Thus, it manages to correlate most of the sources given that they all have postcode attributes and some have overlapping values, e.g., Manchester agencies and Manchester deprivation.

Attribute level. Figure 5.6(a) shows the results at attribute level. In this
5.6. ALGORITHM EVALUATION

(a) Attribute level

(b) Tuple level

Figure 5.6: Performance of Dynamap$^X$ and ++Spicy on real-estate data

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 5.6(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. However, this is not possible without materializing data as one needs to cache different views representing combinations between all sources that share matches to key attributes, e.g., postcode. 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.

Dynamap$^X$ generates a mapping that creates similar data to the ground-truth data, i.e., all true positive tuples are as expected in the ground truth (complete), while ++Spicy generates almost all data as in the ground truth, but there are 20 tuples whose data was not aligned as expected. Out of the 46 false positive
and the 46 false negative tuples, 40 tuples (from both the output and the ground truth) could not be compared because, although full outer join was used to align the target data, the tuples contain null values on postcode, i.e., there are agency properties that do not have a postcode value in the sources, thus, they could not be compared to the ground truth. For the remaining 6 tuples labelled as either false positive or false negative tuples, DynamapX generated skolem values so that it does not lose the correlation between the tuples in the target, i.e., some source data in the UTR was aligned (e.g., price and street_name), but there was no value on postcode, so DynamapX generated a labelled null for it so that the initial source data remains correlated. This problem can be a common problem in mapping generation over autonomous sources, so DynamapX tries to preserve as much information as possible from the sources. ++Spicy makes the assumption that the source data is coming from a well-behaved source schema, thus, it does not pad with labelled nulls any missing information, rather it assumes that all data is present for all matched attributes in the target. For the same reason, ++Spicy outputs 20 false positive tuples, as it does not preserve the correlations between different attributes in the sources, e.g., between price and street_name if the postcode value is missing.

Comparing the populated target tables separately, one major difference between the two approaches is on the labelled nulls they generate. Due to this difference, there are discrepancies between the tuples created by DynamapX and those created by ++Spicy. For instance, the size of the city relation populated by DynamapX has 117 tuples compared to 850 tuples in the case of ++Spicy. Of those ++Spicy tuples, 97 have only the created labelled null for the primary key and then an entirely empty row and others have repeating values on the city_name (e.g., Manchester appears 512 times). The creation of such redundant labelled nulls happens because, even though the city information is missing, there is information for the other attributes, e.g., for streets, so it links a tuple in City, even though there is no information in it. The approach taken by ++Spicy [Martette et al. (2010)] for generating labelled nulls was explained in Section 5.4.2. 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.

Results on mapping characteristics. In our proposal, the mapping characteristics are given by the attribute scores that are computed based on the output
5.6. ALGORITHM EVALUATION

(a) Attribute Completeness  
(b) Source-extracted Values  
(c) Key Consistency

Figure 5.7: Attribute characteristics on real-estate data

data. Using the method presented in Section 5.5, Dynamap\textsuperscript{X} predicts attribute scores based on the propagated profile data and metadata on the mappings (i.e., size) and their attributes (i.e., nulls and distinct values) without materializing the mappings. Figures 5.7(a), 5.7(b), and 5.7(c) show the predictions made by Dynamap\textsuperscript{X} for attribute completeness, source-extracted values ratio, and key consistency (if needed), together with their corresponding real scores computed on the materialized mappings of Dynamap\textsuperscript{X}, ++Spicy, and the ground-truth. The attributes that are expected to be (primary) keys or foreign key in their corresponding tables are marked on the plots with $[K]$ and $[F]$, respectively.

Attribute completeness. Figure 5.7(a) shows the attribute completeness score for each attribute in the target schema. The method for computing these scores was described in Section 5.5.1. The attribute completeness score per attribute represents the ratio between non-null values, i.e., either source-extracted or labelled nulls, in that attribute and the total number of tuples in the table. A (predicted) score of 1.0 means that the attribute is (likely to be) fully populated with non-null data, i.e., either source data or skolems.

In Figure 5.7(a), it can be observed that, for most attributes, Dynamap\textsuperscript{X}’s predicted scores are close to the real values of the scores produced on Dynamap\textsuperscript{X}’s materialized mappings. The three cases where the prediction is lower than the real values are $C.city\_name$, $CR.crimerank$, and $S.street\_name$. This is due to the fact that the size of the base tables obtained by projections on the view produced by the UTR mapping, cannot be derived from the size of the UTR mapping. Thus, for computing the scores, the approximated size of the UTR mapping is used. In
this scenario, the estimated size of the UTR mapping (1,138 tuples) proved to be significantly larger than the sizes of the materialized target tables, e.g., $|C| = 117$, $|CR| = 864$, $|P| = 985$, and $|S| = 670$.

For the foreign key attributes $P.postcode$, $CR.street_id$, $S.city_id$, Dynamap$^X$ predicts a slightly larger score, as it predicts that almost all entries on these attributes will have a value, but they do not, as the real completeness score is smaller than predicted. This is because the mappings that Dynamap$^X$ generates create skolems on foreign key attributes only if the foreign key can be used to link more source-extracted information (from referenced tables). For example, in some tuples in Street Info relation ($S$) it does not create labelled nulls on $S.city_id$ because there are no city names in the referenced table City(C) which could be linked to those tuples. This way, it avoids creating redundant data (both tuples in City table and attribute values in Street Info table) that would not correlate more source-extracted information. The same is the case for attribute $P.postcode$ and $CR.street_id$, i.e., they are all foreign key attributes.

On attribute $P.price$, it can be observed that the ratio of non-null data with which ++Spicy populates the attribute is higher than the one corresponding to the Dynamap$^X$ and the ground-truth mappings. This is an example where egds in ++Spicy’s mapping generation strategy are successful at eliminating redundant information. Using the intermediate materialized tables and the egds corresponding to postcode attributes, it manages to merge the source tables in a variety of ways such that it does not produce tuples in the output that are subsumed by others and whose subsumption can be detected only through combinations of materialized tables.

In some other cases, e.g., $P.prop_id$, the completeness scores for ++Spicy are smaller as it generates nulls on them when the columns that are used for the generation of the skolems are nulls, e.g., $P.postcode$ is null, even though other attributes in the same relation are not nulls, e.g., $P.price$. On the other hand, for the same example, Dynamap$^X$ uses all available data in the tuples to generate a skolem value for the key attributes, e.g., $P.prop_id$ as this avoids the violation of the target constraint.

Source-extracted values. Figure 5.7(b) shows the source-extracted data ratio for each attribute in the target schema. The method for computing these scores was described in Section 5.5.2. The source-extracted values ratio per attribute represents the ratio between non-null source-extracted data (i.e., excluding skolems)
in that attribute and the total number of tuples in the table. The difference between Figure 5.7(a) and Figure 5.7(b) is on the attributes with constraints, i.e., primary key and foreign key attributes (marked on the plots with $\text{[K]}$ and $\text{[F]}$). These show a decreased score compared to the scores from attribute completeness as the labelled nulls are no longer considered.

**Key consistency.** Figure 5.7(c) shows the key consistency score for each attribute in the target schema. The method for computing these scores was described in Section 5.5.3. The key consistency score per attribute represents the ratio between number of non-null distinct values, i.e., either source-extracted or labelled nulls, in that attribute and the total number of tuples in the table. A (predicted) score of 1.0 means that the key constraint is (likely) to be satisfied.

The key consistency score is important for detecting if the primary key attributes contain unique values or if the constraint is expected to be violated. In this scenario, Dynamap$^X$ can accurately predict the outcome on the violation of the constraints on the primary key attributes that are not matched as it relies on the method for generating the labelled nulls to populate these with unique values.

In some cases, it can be observed that there are discrepancies between the key consistency score predicted by Dynamap$^X$ and the actual scores. For example, although not a key, in the case of $\text{C.city\_name}$, the predicted score is approximately 0.1 while the real score is 1.0. This is for the same reason as mentioned above for attribute completeness, i.e., the large difference between the estimated size of the UTR mapping and the real size of the base table led to a decreased predicted value. For $\text{C.city\_name}$ the size of the base table is $|\text{C}| = 117$, the estimated size of the UTR mapping is 1,138, and the number of distinct non-null values on $\text{C.c\_name}$ is 117 (which amounts to approximately 10% in 1,138). For the same attribute, $\text{C.city\_name}$, the key consistency score computed on the materialized data with ++Spicy mappings is rather low as well. This is due to the method they choose to generate the skolem values (as explained in Section 5.4.2). More precisely, they choose to generate labelled nulls even for tuples that do not have a city name and which leads to rows which contain one attribute with a labelled null (the $\text{city\_id}$ value), followed by null values on the rest of the attributes.

Although it is not a primary key, for attributes such as $\text{S.city\_id}$ and $\text{CR.street\_id}$ (which are foreign key attributes), Dynamap$^X$ predicts that the key consistency score is high (equal to 1.0), while the actual score is lower. This is due to the same reason, i.e., after projection, it cannot be detected how data per attribute
CHAPTER 5. MAPPING GENERATION FOR A COMPLEX TARGET

changes in terms of unique values, so it assumes the same key consistency score as for their corresponding referenced attributes which have a key consistency score of 1.0.

Furthermore, for the postcode attributes in \( P \) and \( CR \), ++Spicy manages to achieve higher scores than Dynamap\(^X\) and the ground truth because it uses the egds to remove subsumed tuples (as explained above). Removing subsumed tuples leads to fewer duplicates which leads to a higher score.

For the cases where ++Spicy achieves lower scores than Dynamap and the ground truth, the attributes have nulls even if they are primary key attributes, e.g., \( P.prop_id \). For example, as explained above, for \( P.prop_id \) it generates 42 null values because the property entries are missing postcode values, thus, the skolem functions have null parameters outputting null values as postcode is the only considered attribute.

Using the key consistency scores and given the fact that both the foreign key and the primary key attributes (in the base tables) draw values from the same UTR attribute, it can be ascertained if the foreign key constraints are satisfied as well. The key consistency score is used to detect if the referenced attribute is (likely) to contain unique values, while the second fact shows that the full inclusion dependency condition is satisfied, thus, the two conditions necessary for satisfying foreign key constraints can be checked and a conclusion in this respect can be drawn.

To conclude, in this experiment:

1. For data correlation: Dynamap\(^X\) mapping is close to the data that the ground truth mappings generate, as Figures 5.6(a) and 5.6(b) show that almost all its output tuples are (entirely) correct. ++Spicy produces less redundant tuples as it exhaustively merges all source relations and eliminates duplicate values for postcode attributes by aligning equal values.

2. For mapping characteristics we conclude the following:
   (a) Dynamap\(^X\)’s predicted scores for attribute completeness (Figure 5.7(a)), in most cases, are close to the real scores. Source-extracted ratios and key consistency (Figures 5.7(b) and 5.7(c)) can diverge from the real scores due to the fact that for the estimated size of the UTR mapping can be significantly larger than the size of the target tables.
   (b) Dynamap\(^X\) and ++Spicy perform similarly in terms of populating the target as \((i)\) Figures 5.7(a) and 5.7(b) show that they can populate the
target close to the expected ground truth; and (ii) Figure 5.7(c) shows that, in terms of satisfying key constraints, Dynamap$^X$ outperforms ++Spicy for some key attributes ($S.street_id$, $P.prop_id$) where there is a need for labelled nulls, and, for one target key ($CR.postcode$), ++Spicy outperforms Dynamap$^X$ as it manages to discard subsumed values, e.g., removes duplicates on $CR.postcode$.

(c) Dynamap$^X$ performs similarly to the ground-truth as Figures 5.7(a), 5.7(b) and 5.7(c) show that the scores for Dynamap$^X$’s data are equal (or close) to the scores for the ground truth.

5.6.3 Real-world Experiment - Schools Domain

Motivation. In the experiment in Section 4.5.3, Dynamap is run over real-world data from autonomous sources that can merge with respect to a chosen single-relation target schema. In this experiment, we use a target that, instead, has multiple tables linked by constraints, i.e., primary keys and foreign keys, which Dynamap$^X$ aims to satisfy. Our motivation for this experiment is similar to the one in Section 5.6.2, i.e., to show how Dynamap$^X$ performs on various real-world scenarios, considering the same aspects: data correlation (i.e., how well the data is correlated across multiple foreign key target tables) and mapping characteristics (i.e., Dynamap$^X$’s performance in terms of (i) predicting the attribute scores for the generated data; and (ii) populating the target relations in comparison to ++Spicy’s data and the ground-truth data).

Description of scenario. The purpose of this scenario is to generate a set of mappings between open-government data sources and a target describing schools in the format of multiple relations linked through foreign keys.

Target schema. The target schema is depicted in Figure 5.8 and contains four relations with the following schema constraints:

- $H.h_id$, $SI.s_id$, $DFE.dfe_code$, and $ST.s_type_id$ are primary keys for their corresponding relations,
- $SI.h_id \rightarrow H.h_id$ foreign key
- $SI.dfe_code \rightarrow DFE.dfe_code$ foreign key
- $DFE.s_type_id \rightarrow ST.s_type_id$ foreign key

Input sources. In this section, we investigate how Dynamap$^X$ performs on the same real-world sources as in Section 4.5.3 where datasets from the UK open-government data portal from the schools domain are combined. Table 5.14
provides information on the sources: the first column states what the data represents, the second outlines which information from the dataset is necessary to populate the target (but that are not all available in the sources), 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.

**Matches.** As with the previous experiments, given that the focus is on mapping generation, the matches that were given as input were created by hand, thus making sure that the matches are correct and that mapping generation is not hindered by faulty matches. In Table 5.14 (second column), it can be observed that not all target attributes are matched by the sources. The target attributes that are not matched are:

1. for relation $H$: $h\_id$, phone
2. for relation $SI$: $s\_id$, $h\_id$,
3. for relation $DFE$: $s\_type\_id$,
4. for relation $ST$: $s\_type\_id$.

Each source contributes to the target schema by way of at least two matched attributes.

**Ground truth.** The ground-truth mappings were created by hand following a similar method to the one used in Section 5.6.2: given that it is difficult to create separate mappings for each target relation and then to reconcile the data so as to maintain the data correlations according to the target constraints, we grouped all
attributes in a compound relation. Then, a mapping was created such that the sources were merged as best as possible considering their attribute-value overlaps. The mapping created for the compound relation unions the six relations with *Additional languages* information, which is then sequentially merged through full outer join with the other four source relations. For the unmatched attributes that have constraints, e.g., `h_id`, `s_id`, and `s_type_id` (excluding `H.phone` attribute), we created hash values based on the remaining values in the corresponding target relations. As stated before, we consider that the specific procedure for creating these synthetic values does not necessarily compromise correctness. The hash values are important for creating common values between foreign key attributes such that the tuples in different tables are still correlated to one another. Lastly, the initial target tables were populated using the corresponding projections on the view that the mapping for the compound table creates.

*Profile data.* The same method as in Section 4.5.3 was used to obtain the profile data. The input contains 48 candidate keys, 681 partial inclusion dependencies, and 47 full inclusion dependencies.

*Comparison.* Dynamap$^X$ and ++Spicy are run over the same mapping task. After the respective output mappings are executed and the target tables are populated, we compare the two systems on *data correlation* and on *mapping characteristics*, and follow the same methodology as in Section 5.6.2:

i) For *data correlation*, we ascertain whether the data in the target is still correlated. We performed a full outer join between the four tables on all three materialized versions, i.e., the ground-truth data, the data produced by Dynamap$^X$ mappings, and the data produced by ++Spicy. Then, we compared the outer-joined tuples of the ground truth tables, with the outer-join tuples in the tables produced by the Dynamap$^X$ mappings and the ++Spicy mappings. The quality of the results is evaluated in the same manner as described in Section 4.5, i.e., at attribute level and at tuple level. The labelled nulls that were produced by either approach were not considered in the comparison as their purpose is only to correlate the tuples, i.e., we compare only source data.

In order to correlate the ground truth tuples with the output tuples, the *school name* attribute was used, as this attribute can be found in all source relations, i.e., it is expected that the majority of the generated tuples will have a value for this attribute.
ii) For mapping characteristics, we measure how Dynamap\textsuperscript{X} performs in terms of populating the target attributes and satisfying the target constraints, as follows:

(a) We determine the accuracy of the attribute scores that Dynamap\textsuperscript{X} estimates for the unmaterialized mappings by comparing the estimated scores with the real scores for Dynamap\textsuperscript{X} materialized data.

(b) We determine how well Dynamap\textsuperscript{X} performs against ++Spicy in terms of populating the target attributes and satisfying the target constraints by comparing the attributes scores on Dynamap\textsuperscript{X} materialized data with the scores for the data obtained through ++Spicy.

(c) We determine how well Dynamap\textsuperscript{X} performs against the ground truth in terms of populating the target attributes and satisfying the target constraints by comparing the attribute scores on Dynamap\textsuperscript{X} materialized data with the attribute scores that we compute for the ground-truth data.

These characteristics flush out any differences between the strategies the two algorithms pursue when it comes to generating mappings that aim to satisfy target constraints.

Results. The results are described in terms of data correlation and on the mapping characteristics of the output mappings.

For generating the mappings, ++Spicy ran in 0 min, 3 sec, 999 ms and Dynamap\textsuperscript{X} ran in 0 min, 1 sec, 162 ms. For populating the target tables, ++Spicy materialized 21 intermediate tables, while Dynamap\textsuperscript{X} does not need to materialize any. We explain below why ++Spicy is so much faster in this experiment (compared to the one in Section 5.6.2).

Results on data correlation. The results are shown in Figures 5.9(a) and 5.9(b), at attribute level and at tuple level, respectively. Comparing the results in Figures 5.9(a) and 5.9(b) to the ones in Section 4.5.3, one can say they are similar at both attribute and tuple level. This similarity, given the fact that the information needed in the target is essentially the same as in that experiment, shows that Dynamap\textsuperscript{X} does not detract from the quality of the chosen merges between the sources obtained by Dynamap. Hence, the details on the experimental results in this section focus on the differences brought by the target schema with constraints, not on the complexity of the sources.

In this scenario, the mapping generated by Dynamap\textsuperscript{X} first merges the six
5.6. **ALGORITHM EVALUATION**

(a) **Attribute level**

(b) **Tuple level**

Figure 5.9: Performance of Dynamap$^X$ and ++Spicy on a schools scenario

*Additional languages* sources using union. Then, the output is sequentially merged using full outer join, with the remaining source relations on either the *dfe_code* or the *s_name* attributes (if merges on both attributes are available, Dynamap$^X$ compares the overlaps between pairs of attributes and chooses the one with greater overlap, as explained in Section 3.3). This mapping is almost the same as the ground-truth mapping, the only difference is in the way the relations are joined, e.g., join attributes differ on two conditions in the ground truth: Dynamap$^X$ joins on school name (*s_name*) whereas in the ground truth the join is on *dfe_code*. This is due to the fact that the *school name* may have higher (estimated) overlap than *dfe_code*, or that *dfe_code* is no longer a key as it can be lost through propagation (as explained in Section 3.6). Even so, many of the aligned tuples are correctly aligned where the school names are in the same format in both tables, but in some cases, the names differ even though they refer to the same entity.

On the other hand, ++Spicy has a different approach: for merging some source relations, ++Spicy uses *egds* as it detects that *dfe_code* is the primary key in *DFE* target relation. The manner in which it uses the *egds* was explained in Section 2.1.4. For generating the mappings, it produces all possible merges between the tables that have *dfe_code* attributes, i.e., *All schools*, *Free meals eligibility*, and *Additional languages* (in this scenario, not all sources have this attribute). Thus, it manages to correlate the tuples from sources that have a match to the *dfe_code* primary key, but not the others (e.g., *Road and Safety training*, and *Bikeability courses*). However, these sources could have been correlated using the school name (*s_name* attributes) as they have overlapping values, but the school name is not a key in the target, so ++Spicy is oblivious to this merge opportunity and it does not use *egds* on it.
In this experiment, ++Spicy took approximately 4 seconds, while in the previous scenario (Section 5.6.2) it took approximately 10 minutes. This is due to the fact that, here, it does not merge exhaustively all the sources as not all sources match target keys. This is reflected by the small number of intermediate relations as well, i.e., 21, compared to 119 intermediate relations in the real-estate scenario.

*Attribute level.* Figure 5.9(a) shows the results at attribute level. The two systems behave similarly in terms of precision, but do not achieve the maximum for the same reasons as stated in Section 4.5.3.

In Figure 5.9(a), it can be observed that ++Spicy seems to outperform DynamapX by 0.013 in precision. This is because ++Spicy produces more tuples that have the same school name than DynamapX as ++Spicy fails to do the expected joins with the sources that do not match target keys (*Road and Safety training,* and *Bikeability courses*). 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. For instance, assume the (fragment of) a ground-truth tuple from *School Info* table:

\[ gt_1(5300, \text{Calderdale}, 3, 7, \text{null}, \text{null}) \]

And two output tuples that correspond to \( gt_1 \):

\[ t_1(5300, \text{Calderale}, \text{null}, 7, \text{null}, \text{null}), \]
\[ t_2(5300, \text{Calderale}, 3, \text{null}, \text{null}, \text{null}). \]

Here, both \( t_1 \) and \( t_2 \) are compared to \( gt_1 \) as they have the same value on \( s\_name \) (*Calderdale*). It can be observed that, were \( t_1 \) and \( t_2 \) merged, they would have aligned the data better, instead of having complementary information for the third and fourth attributes. However, in this situation, the tool that counts the results outputs that \( t_1 \) contains (at attribute level) 3 true positives, 1 false negative, 2 true negatives, and that \( t_2 \) contains the same. Nevertheless, the last two null values in each tuple are counted twice as being true negatives. Thus, instead of choosing just one correct output tuple for each ground-truth tuple, we consider that showing the results at both attribute-level and tuple-level can outline such behaviors of the mappings.

In terms of recall, ++Spicy has a poorer performance than DynamapX by 0.276. DynamapX does not achieve the maximum because it produces 249 false
negatives (the reasons are the same as stated above). ++Spicy has a lower recall because it fails to do the expected joins, thus, the number of false negative cell values is 3.6 times higher (907 FNs) than for Dynamap\textsuperscript{X} (249 FNs).

\textit{Tuple level.} Figure 5.9(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\textsuperscript{X} and ++Spicy is caused by the same reason as explained above.

Also, in Figure 5.9(b), it can be observed that Dynamap\textsuperscript{X} 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\textsuperscript{X}, but chosen from another source to represent the same entity. Hence, there are 39 Dynamap\textsuperscript{X} 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 when doing mapping generation over autonomous sources as many sources can contain data about the same entities, but in different formats.

\textbf{Results on mapping characteristics.} Figures 5.10(a), 5.10(b), and 5.10(c) show the predictions made by Dynamap\textsuperscript{X}, i.e., attribute completeness, source-extracted values ratio, and key consistency (if needed), together with their corresponding real scores computed on the materialized mappings of Dynamap\textsuperscript{X}, ++Spicy, and the ground truth. The attributes that are expected to be (primary) keys or foreign keys in their corresponding tables are marked on the plots with |K| and |F|, respectively.

\textit{Attribute completeness.} Figure 5.10(a) shows the attribute completeness score for each attribute in the target schema. The method for computing these scores was described in Section 5.5.1.

In Figure 5.10(a), it can be observed that, for most attributes, the predicted scores are close to the real values on materialized mappings. The four cases where the prediction is significantly lower than the real scores are for attributes \texttt{H.ht\_name}, \texttt{SI.bikeability\_courses}, \texttt{SI.pupils\_FSM\_eligible}, and \texttt{ST.type\_name}. The reason is the same as explained in Section 5.6.2: the size of the base tables obtained by projections on the view produced by the UTR mapping cannot be derived from the size of the UTR mapping. Thus, for computing the scores it is used the estimated size of the UTR mapping. In this scenario, the estimated
size of the UTR (539 tuples), proved to be significantly larger than the sizes of the materialized base tables, e.g., $|DFE| = 121$, $|SI| = 358$, $|HT| = 99$, and $|ST| = 8$.

For foreign key attributes such as $DFE.s\_type\_id$, $SI.dfe\_code$, and $SI.ht\_id$, Dynamap$^X$ predicts a larger score, as it predicts that almost all entries on these attributes will have a value, but they do not, as the real completeness score is smaller than predicted. This is because the mappings that Dynamap$^X$ generates create skolems on foreign key attributes only if the foreign key value can be used to link more source-extracted information (from referenced tables). For example, in some tuples in the $School\ Info$ relation ($SI$) it does not create labelled nulls on $SI.ht\_id$ because there is no source-extracted data for headteacher names ($H.ht\_name$) in the referenced table $Headteacher(H)$ which could be linked to those tuples. This way, it avoids the creation of redundant data (under the form of either tuples in $H$ table and attribute values in $SI$ table) that would not correlate more source-extracted information. The same is the case for attributes $SI.dfe\_code$, and $DFE.s\_type\_id$, i.e., the labelled nulls do not help correlating more extracted data, thus, they are not created in the foreign key attributes for some tuples. Moreover, because of this strategy, it can be observed that, on attributes $SI.dfe\_code$ and $SI.ht\_id$, the ratio of non-null data with which ++Spicy populates is higher than the one corresponding to the Dynamap$^X$ and the ground-truth mappings. This is due to the fact that ++Spicy generates a labelled null for the foreign key attributes regardless of whether there is source-extracted information across multiple target tables that need to be correlated or not, thus, the foreign key attributes have more labelled nulls than the attributes
populated by Dynamap\textsuperscript{X}.

Also, ++Spicy populates the key attributes in the same manner as previously explained, thus, it produces a labelled null for primary key attributes in referenced tables (e.g., \textit{ST.s\_type\_id}) where there is no source-extracted data (e.g., \textit{ST.type\_name} are nulls), thus, it produces many tuples which are empty. This strategy leads to an increase in size of the materialized base tables, e.g., the size of the \textit{ST} relation materialized with ++Spicy mappings is $|\textit{ST}| = 247$, out of which 201 tuples have nulls on \textit{ST.type\_name} and a skolem on \textit{ST.s\_type\_id}, whereas the materialized \textit{ST} tables using the ground truth and Dynamap\textsuperscript{X} mappings contain 8 tuples with the same number of source-extracted data for \textit{type\_name}.

\textit{Source-extracted values.} Figure 5.10(b) shows the source-extracted data ratio for each attribute in the target schema. The method for computing these scores was described in Section 5.5.2. The difference between Figure 5.10(a) and Figure 5.10(b) is on the attributes with constraints, i.e., primary key and foreign key attributes (marked with \([K]\) and \([F]\) on the plots). These show a decreased score compared to the scores from \textit{attribute completeness} as the labelled nulls are no longer considered.

On the \textit{DFE.dfe\_code} attribute (where there is a primary key constraint) there are both labelled nulls and source-extracted data using both Dynamap\textsuperscript{X} and ++Spicy strategies. The difference between the two scores comes from the fact that Dynamap\textsuperscript{X} does not generate a tuple in \textit{DFE} relation (with a labelled null on \textit{DFE.dfe\_code}) unless there is a tuple in \textit{SI} relation that needs to be linked to a source-extracted value in \textit{ST.type\_name}. Otherwise, it considers it to be redundant to create such tuples. This strategy is different from ++Spicy’s, which creates a new labelled null for each entry in \textit{SI} that does not have a \textit{dfe code} value. The creation of such skolems is done regardless of whether there is data in \textit{ST.type\_name} or not (leading to the increased size in \textit{ST} relation as explained for \textit{attribute completeness}).

Moreover, it can be observed that the dependent attribute of \textit{DFE.dfe\_code}, i.e., \textit{SI.dfe\_code}, has a similar outcome: the real score for the source-extracted ratio on Dynamap\textsuperscript{X}’s data is higher than the score for ++Spicy data, although, for attribute completeness, ++Spicy’s score is higher. This reflects the large number of skolems that ++Spicy generates and which Dynamap\textsuperscript{X} avoids generating, unless they are necessary for data linkage.

Besides the \textit{dfe code} attributes in relations \textit{DFE} and \textit{SI}, the other constrained
attributes are populated with skolems, thus, the source-extracted ratios become 0.

**Key consistency.** Figure 5.10(c) shows the key consistency score for each attribute in the target schema. The method for computing these scores was described in Section 5.5.3.

It can be observed that Dynamap\(^X\) was not able to accurately predict the violation of some of the target constraints. This is due to the fact that the key constraint consistency is achieved only after the projection is applied on the view of the UTR mapping. This (pessimistic) prediction is used because the algorithm cannot detect if projecting several attributes from a table will achieve uniqueness for the new projected tuple values. For example, on DFE.dfe\_code, where there is a primary key, Dynamap\(^X\) predicts a lower score (approximately 0.22) while the key constraint is actually satisfied by the materialized data (the real score is equal to 1.0). This is because of the same reason mentioned above for attribute completeness, i.e., the large difference between the estimated size of the UTR mapping and the real size of the base table led to a decreased predicted value.

For DFE.dfe\_code the size of the base table is \(|DFE| = 121\), the estimated size of the UTR mapping is \(|map_{UTR}| = 539\), and the number of distinct non-null values on DFE.dfe\_code is \(V(DFE.dfe\_code) = 121\) (which amounts to approximately 22% in \(|map_{UTR}| = 539\) – as shown in Figure 5.10(c)). The reason for the difference of the scores on attributes H.ht\_name, SI.pupils_FSM_eligible, and ST.type\_name is the same.

For foreign key attributes such as SI.ht\_id, and DFE.s\_type\_id Dynamap\(^X\) predicts that the key consistency score is high (equal to 1.0), while the actual score is lower. This is due to the same reason, i.e., after projection, it cannot be detected how data per attribute changes in terms of unique values, so it assumes the same key consistency score as for their corresponding referenced attributes which have a key consistency score of 1.0. The lower key consistency score can be caused by the duplicate values on the foreign key attributes (which can be expected), but, also, because of the nulls that are kept unchanged by Dynamap\(^X\) as there is no source-extracted data to link to the referenced tables. The predicted score indicates that there is always a source-extracted value that will be linked to foreign key tuples, thus, for which Dynamap\(^X\) predicts it generates a labelled null.

To conclude, in this experiment:

1. For data correlation: Figures 5.9(a) and 5.9(b) show that the data generated by Dynamap\(^X\) mappings is close to the data that the ground-truth
mappings generate. The differences come from the different join conditions that Dynamap$^X$ chooses w.r.t. the ones used in the ground truth. On the other hand, ++Spicy is not able to find some joins with two data sources if those data sources do not match target key attributes (to enable egds), thus, it is not able to correlate all source data as expected.

2. For mapping characteristics, we conclude the following:

(a) The accuracy of Dynamap$^X$’s predicted scores for attribute completeness (Figure 5.10(a)), is, in most cases, close to the real scores. Source-extracted ratios and key consistency (Figures 5.10(b) and 5.10(c)) can diverge from the real scores due to the fact that the estimated size of the UTR mapping can be significantly larger than the size of the target tables.

(b) Dynamap$^X$ outperforms ++Spicy in terms of populating the target as (i) Figures 5.10(a) and 5.10(b) show that Dynamap$^X$ can populate the target with fewer redundant attribute values by creating labelled nulls only if they are necessary for correlating source-extracted data; and (ii) Figure 5.10(c) shows that Dynamap$^X$ and ++Spicy perform similarly in terms of satisfying key constraints.

(c) Dynamap$^X$ performs similarly to the ground-truth as Figures 5.10(a), 5.10(b) and 5.10(c) show that the scores for Dynamap$^X$’s data are equal (or close) to the scores for the ground truth.

5.7 Conclusions

In this chapter, we have extended the work presented in Chapters 3 and 4 by adding two components to the mapping generation process that enable Dynamap$^X$ to tackle scenarios where the target schema is complex, i.e., it has schema constraints. One component is added to the pre-processing step. It relies on join paths in the target schema to create a universal target relation (UTR) that comprises all relations linked by foreign key relationships. Then the universal target relation is used as a target in the mapping generation process (which remains unchanged from the version presented in Chapter 4). After the UTR mappings are generated, a set of post-processing steps are performed where the UTR mappings are further refined as the algorithm aims to satisfy the target constraints by creating labelled nulls and removing subsumed tuples where necessary.
In Section 5.6.1, we have evaluated Dynamap^X against a set of complex scenarios generated by the state-of-the-art generator, i.e., iBench, where we show that Dynamap^X is able to tackle scenarios that are expected to be handled by mapping generation algorithms. The scenarios were created following the methodology in Arocena et al. (2015), where several primitives are used to create complex scenarios where the target is rich with constraints. The results show that Dynamap^X can generate the expected mapping for all the scenarios but the one centered on self-join, which, we have argued, cannot be automated in the wild.

In Sections 5.6.2 and 5.6.3, we have shown how Dynamap^X performs on real-world scenarios where the target comprises multiple relations with different primary key and foreign key constraints. In all experiments, we compared Dynamap^X with ++Spicy. In Marnette et al. (2011), it is described how ++Spicy relies on target constraints in the form of egds to align source data for target attributes that are primary keys. Hence, in these scenarios where many source and target constraints are explicitly stated, one can say that ++Spicy performs fairly well as the scenarios became quite close to well-behaved scenarios. On the other hand, there are source relationships that ++Spicy is oblivious to, causing it to fail to produce all expected merges, while Dynamap relies less on explicit schema constraints and more on source profile data to detect merge opportunities.

The problem of mapping generation for a complex target has been studied before, building on Clio [Miller et al. (2000)], by works such as Popa et al. (2002); Bonifati et al. (2008); Fagin et al. (2009), and Mecca et al. (2009). For tackling the problem of generating mappings for target schemas with constraints, Popa et al. (2002) introduce the notion of semantic translations, which are interpretations of value correspondences for satisfying target constraints. These interpretations are named logical mappings. Their approach is discussed in Section 2.1.4. Also, Clio was at the basis of (+++)Spicy which is described in Bonifati et al. (2008)(Spicy), Mecca et al. (2009)(+Spicy), and Marnette et al. (2010)(++Spicy). The approach of ++Spicy is to rewrite the s-t tgds so that the tuples generated by subsumed mappings are avoided, i.e., they use egds that rely on the existence of primary keys in the target. These approaches were discussed in Section 2.1.4. These mapping generation algorithms produce reasonable results as long as there are explicit source and target schema constraints. However, under the setting of mapping generation over autonomous sources, it is not always possible that the source data will comply to them. Dynamap^X was developed under this setting
and although it aims to satisfy the target constraints, the source data might not allow it. Hence, we propose a set of mapping characteristics based on which the user can understand which target constraints are guaranteed to be satisfied and which are not guaranteed (but might be satisfied). The characteristics that we propose rely on the profile data and the metadata of the generated mappings.

Characterizing schema mappings has been studied by Gottlob and Nash (2008); Alexe et al. (2011a,b). Their approaches rely on the fact that the output data of the generated mappings can be materialized and compared with a set of ground-truth tuples which were given as input. In our setting, the mappings need to be characterized before the user chooses to materialize them as Dynamap\textsuperscript{X} can generate thousands of mappings, i.e., it becomes unfeasible to materialize all of them in order to perform the comparison. Moreover, in the context of mapping generation over autonomous sources, it might not be obvious what is a suitable set of ground-truth tuples given that the pool of sources can be large, and without the ground-truth tuples, the characteristics cannot be computed with the previously proposed approaches.

To conclude, there have been works on generating mappings for scenarios where the target is complex in terms of constraints and they produce reasonable results on well-behaved source schemas where there are explicit join paths. However, the focus in these works is not to transform source data that was not meant to merge, and in such scenarios the target constraints might not be satisfied. Given this problem, in this chapter, we describe a method for creating mappings that aim to satisfy the target constraints, for which we propose set of characteristics that determine to which degree the target constraints are expected to be satisfied. If a user is interested in selecting mappings based on the degree to which the target constraints are satisfied, Dynamap\textsuperscript{X} is able to detect which are more suited to the given priorities. Such priorities can become essential when one wants to merge multiple data sources that were not designed to be merged and bring them in a format which needs to satisfy keys and foreign keys, thus, constraint violations can be expected and this can become a criterion in filtering the mappings further.
Chapter 6

Conclusions and Open Issues

"We can't solve problems by using the same kind of thinking we used when we created them."
– Albert Einstein (1879–1955)

6.1 Concluding Discussion

Schema mapping generation is the data management task that, given a set of sources and a target schema, generates a set of executable transformations, i.e., schema mappings, that transform source data into the format of the target schema. In this thesis, we propose a new mapping generation system, Dynamap\(^{(X)}\), which enables the creation of mappings over repositories of autonomous sources that are assumed to not have been, on the whole, created with a view to being integrated. The thesis has been structured in terms of three major contributions, which we summarize.

**Contribution 1.** We contribute a technique for generating mappings in settings where there are no explicitly declared relationships between the sources. This contribution was described in Chapter 3 where we have proposed three methods for solving different facets of the problem:

1.1) We described a method for merging sources that match the same single target table and that share schema properties as inferred from profiling data, e.g., candidate keys and (partial) inclusion dependencies. The profiling data is used to infer (relaxed) foreign key constraints by using candidate keys and (partial) inclusion dependencies, where inclusion dependencies with higher overlap are preferred.
1.2) We described a dynamic programming algorithm for exploring the search space of mappings that are combined using the method in (1.1). The algorithm searches for various ways to merge the input sources based on profiling data. A dynamic programming approach was adopted because of the need to devise a technique that explores all possible merges (as there are no explicitly declared join paths, i.e., foreign keys). Because of this, most of the time, there is no obvious way of merging the sources so the algorithm needs to explore a variety of ways for doing so. Moreover, the mapping generation problem that we have tackled is a difficult problem to solve without relying on a methodology that builds solutions bottom-up. Mappings are combined in a bottom-up fashion in the sense that the algorithm starts by creating mappings for small subsets of initial relations and, then, based on those mappings, builds other mappings that merge larger subsets of sources. The merging process stops when all initial sources are merged (if possible). However, the combinatorial nature of the search comes with performance challenges that need to be addressed through pruning. We addressed this challenge in Contribution 2 (below).

1.3) The method in (1.1) needs profiling data in order to make informed decisions for combining mappings. However, the bottom-up fashion in which the solutions are built creates intermediate mappings for which the same type of profile data that existed for the input mappings is needed, since they would not be able to be merged further without it. We describe a method for assigning profiling data to the intermediate mappings created during the mapping generation search process. This is done by propagating the already-known profiling data from the inputs to output, i.e., to the intermediate mapping. The propagation takes into consideration (i) the type of merge operator that was chosen for the creation of the new mapping, and (ii) the type of profile data that needs to be propagated, i.e., (partial) inclusion dependencies or candidate keys.

Other proposals view mapping generation as a search problem, using either generic or bespoke strategies. In relation to generic strategies, Fletcher and Wyss (2006) describe Tupelo, a mapping discovery algorithm that performs search within the transformation space of example instances based on a set of mapping operators. These are operators which help create more 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. The search is done using best-first search which is a strategy that evaluates the most promising step to follow in solving a problem (instead of considering all steps equally promising). Clio [Miller et al. (2000)] and (+++)Spicy [Mecca et al. (2009); Marnette et al. (2010)] 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 merge 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 Chapter 3, the experiments show that Dynamap is able to tackle a set of mapping generation scenarios generated with the state-of-the-art integration-scenario generator, iBench. However, this version of Dynamap is not able to tackle vertical partitioning scenarios, i.e., where the target contains foreign key constraints. This challenge is tackled in Contribution 3.

Contribution 2. In order for the dynamic programming technique applied to our problem to scale to repositories with tens/hundreds of sources, we described a set of strategies for pruning the search space and the profile data. The proposed strategies on the search space rely on (possibly propagated) profile data. For the purpose of pruning, the profiling data is treated as an indicator of how promising a mapping is for further merges, as the merge operator is chosen based on profile data. By keeping the search space from growing too fast, the algorithm no longer needs to explore paths that do not promise to lead to (better) solutions. The second type of pruning is on profiling data. This is done by considering the usefulness of the profiling data, i.e., if the data is likely to be used in further merge conditions or if it is kept without the prospect that it will be used. The two reasons behind pruning the profiling data are: (i) Dynamap no longer needs to explore merge opportunities that are unsuitable, and (ii) the propagation step infers only profiling data when doing so seems promising for the creation of new potentially good quality intermediate mappings.

The experiments in Chapter 4 show the impact that the proposed pruning strategies have. We run Dynamap against stress tests that can comprise scenarios with up to 556 source relations capable of being merged through either union or
For a scenario with 556 sources, without any active pruning strategies, in the second iteration of the dynamic programming algorithm, if each two relations can be merged pairwise, the number of created intermediate mappings can reach up to $(C_{556}^2 = 154,290)$, which then need to be merged further in the next 554 iterations to create other mappings. The search space can therefore grow quite rapidly. The proposed pruning strategies performed reasonably in the test with 556 sources, Dynamap keeps only 304 intermediate mappings for iteration 2. The results on real-world data were reasonable in terms of both data quality and processing time and, using the real-world data, we have shown that the accuracy of the propagated profile data is preserved for further merges, i.e., the possible uncertainty caused by the approximated overlap values did not hinder the mapping generation process and the desired mappings were still generated.

Addressing the problem of integrating heterogeneous sources from large repositories, the problem of finding related sources at scale has been the focus of work by Das Sarma et al. (2012); Zhu et al. (2016); Castro Fernandez et al. (2018), and Nargesian et al. (2018), where methods for detecting relationships between the sources are proposed, e.g., whether they are joinable or unionable. Das Sarma et al. (2012) contribute a method for detecting various types of relatedness. Their experiments show that their proposed methods can run over a million sources from Wikipedia. Zhu et al. (2016) detect join paths between the sources based on the domains of the values in the attributes. Their experiments were run on relational data comprising 10,635 relations. A similar idea can be found in Nargesian et al. (2018), where the idea of table unionability (based on pairs of attributes possibly from different sources that have similar domains) is proposed. Their experiments were performed on repositories of up to 165,236 tables.

These approaches are complementary to the work we present in this thesis, in the sense that they can provide information regarding the necessary evidence for deciding when to merge two relations and in which manner they could be merged. Their focus is on discovering relationships between materialized sources which means that their content can be analyzed. Our contributions go beyond relationship discovery to perform fully fledged mapping generation replacing the need for materialized data with techniques for profiling data propagation.

**Contribution 3.** Chapters 3 and 4 described Dynamap as a mapping generation technique that can handle scenarios where there is a single-relation target schema without constraints. In Chapter 5, we extend that work by contributing a method
for integrating multi-relation targets taking into consideration target constraints giving rise to what we refer to as $\text{Dynamap}^{(e)}X^{(tended)}$.

The extensions serve complementing purposes: one component creates a universal target relation ($\text{UTR}$) comprising all attributes in all relations linked through primary-foreign keys, and the second decomposes the $\text{UTR}$ mappings into mappings to populate the target tables. The $\text{UTR composition}$ is a pre-processing step and $\text{decomposition}$ is a post-processing step. For the $\text{decomposition}$ step, we proposed a methodology for generating labelled nulls and removing subsumed tuples with a view to satisfying the target constraints. Given that the mapping generation context that we focus on implies autonomous sources that might not (always) satisfy the target constraints, we must accept that it may not be possible to satisfy all constraints. For this reason, we proposed a set of mapping characteristics that rely on measuring the quality of the data at attribute level, viz., $\text{attribute completeness}$ and $\text{source-extracted ratio}$, and the degree to which the target constraints might be satisfied, i.e, $\text{key consistency}$ from which $\text{foreign key consistency}$ can be derived as well.

In the experiments in Chapter 5, we have shown that $\text{Dynamap}^X$ is now able to tackle the $\text{IBench}$ scenarios that involve $\text{vertical partitioning}$, i.e., with foreign keys in the target schema. Using real-world data, we have shown that the proposed techniques for creating mappings that generate labelled nulls and discard subsumed tuples yield reasonable results in terms of correctly preserving the information correlated across multiple target tables that are linked through foreign keys and in terms of satisfying target constraints. The experiments have shown that:

i) $\text{Dynamap}^X$ mappings produce results that are close to the results generated through the mappings designed by hand (the ground truth).

ii) $\text{Dynamap}^X$ mappings produce results that satisfy the target constraints as often as possible.

iii) The predicted attribute scores are often close to the real values, showing that they could be used for selecting which mappings to prefer.

The problem of generating mappings for a target schema with constraints has been the focus of work by Popa et al. (2002); Bonifati et al. (2008); Fagin et al. (2009), and Mecca et al. (2009). This body of work advances the seminal insights that resulted in Clio [Miller et al. (2000)]. One difference between their approach and ours is that $\text{Dynamap}^X$ does not rely on or assume the existence of target
6.1. CONCLUDING DISCUSSION

Constraints and, therefore, sources are integrated using a best-effort ethos built upon opportunities that may emerge from the profiling data. In other words, our mapping generation approach is focused on the complementarity of the sources and merge opportunities with respect to the required target and not on enforcing target constraints. In our approach, after the sources are combined in \( UTR \) mappings, the post-processing step takes into consideration the target constraints and decomposes these mappings for separate target tables. This ensures best-effort merges between the sources. For instance, in Marnette et al. (2011) they use \textit{egds} that enforce target keys. Their method relies on exhaustively merging all source relations that match the same target keys, thus obtaining unique values for the key attributes. However, if the sources do not match any of the target keys, then the sources are merged by using the declared source foreign keys. Nevertheless, over autonomous sources, one cannot assume that the target constraints exist or that their attributes are always matched and satisfied by the sources.

6.1.1 Impact of Dynamap

Dynamap has been integrated into the VADA (Value-Added DAta Systems) system [Konstantinou et al. (2017, 2019)], for data wrangling. VADA is an automated data preparation tool that is the result of a joint collaboration of several pieces of research work that tackle different wrangling tasks, e.g., data discovery, schema matching, mapping generation [Mazilu et al. (2019)], mapping selection [Abel et al. (2018)], format transformation [Bogatu et al. (2018, 2019)], and data repair based on data context [Koehler et al. (2017)]. The project started with the purpose of creating a tool that does not require extensive information about the sources and only asks the user to define the desired target. The tool fully automates the production of data products tailored to a set of user preferences that can be set before-hand. The mapping generation component is responsible for merging the relevant relations in the source repository (typically a data lake) using only profiling data. The output data products can then be further refined through user feedback, or the user can revise the preferences, for which VADA computes a revised product.

The work on VADA evolved into a commercial product named \textit{Data Preparer} where Dynamap is used as the prototype for the mapping generation component that lies at the core of the data wrangling process.

\footnote{https://www.thedatavaluefactory.com/}
6.2 Open Technical Challenges

The experiments on Dynamap\(^{(X)}\) against different scenarios reveal a set of directions for outstanding issues and future work that are now described and discussed.

6.2.1 Mapping Fitness

The fitness function for mapping generation plays an important role in steering the process into generating a set of tuples with certain characteristics.

The fitness function described in Section 3.5, which was used across all experiments, performed reasonably on the data used for evaluation. However, as mentioned there, the fitness function can be changed so as to prefer mappings with different characteristics. Using the profiling data available, other options could favour mappings with

(i) the lowest ratio of estimated nulls, which would prefer results with as few sparse tuples as possible; or

(ii) the highest number of distinct values on matched attributes, which would rank higher the mappings that gather data from sources that are as disparate as possible; or

(iii) the highest cardinality, which would favor mappings that merge data from (possibly) many sources; or

(iv) the best coverage for the chosen target, which aims to populate as many attributes as possible; or, finally, (v) the degree to which the target constraints are satisfied, thus, driving the mapping generation towards this aim. Other options may, of course, exist.

We consider that further work in this direction could explore whether a more complex fitness function could be used, e.g., one that could incorporate a combination of the above suggestions and adapt to a set of user preferences (if any). Also, other profiling data, e.g., functional dependencies, could be passed to the algorithm so that the fitness function could consider the likelihood of satisfaction of target constraints into the fitness value.

6.2.2 Characterizing Mappings without Materialization

Mapping characteristics can influence the way a target is populated as the mappings that have a certain set of characteristics may be filtered and ranked higher.

As noticed in the experiments in Sections 5.6.2 and 5.6.3, the formulas for computing the quality of the data on the attributes can, in some cases, diverge from the real scores. This is due to the fact that, in this thesis, the scores for
attribute completeness, source-extracted values ratio and key consistency scores are computed using the estimated size of the UTR table, which, in many situations, differs from the actual size of the resulting base target tables, thus leading to discrepancies between the predicted and the real scores. In such situations, in order to achieve accurate scores, materializing the data would be the solution, however, this may not always be feasible as it can become costly if the number of mappings is large.

Further work on computing the proposed scores could consider adding a set of conditions to bring the predicted scores closer to the real values. An example of an added condition could be as follows. Assume the schema of a UTR table \( U(X, a_1, a_2, \ldots, a_n) \) and one of its base tables \( T(a_1, a_2, \ldots, a_n) \), where \( U.X \) is a set of attributes from other base tables, \( T.a_1 \) is a key, and where \( T.a_n \) is the only attribute matched by the sources in \( T \). In this setting, Dynamap\(^X\) computes the attribute scores using the estimated size of the UTR table. Assuming that the estimated size of \( U \) is significantly larger than the real size of \( T \), \(|U| > |T|\), then the predicted scores of the attributes will be smaller than the real values. However, the predicted values could be brought closer to the real scores, if, with the estimated number of distinct values on \( T.a_n \) in hand, check if other attributes besides \( T.a_n \) in \( T \) are matched. In our example, \( T.a_n \) is the only matched attribute. This leads to the conclusion that the size of the materialized table \( T \) is (close to or) equal to the estimated \( V(T.a_n) \). One can draw this conclusion because \( T.a_n \) is the only matched attribute, i.e., the only one that can have source-extracted data, and the skolems the key attribute \( T.a_1 \) are computed using those values only (as explained in Section 5.4.2). Thus, it can be concluded that \( T \) cannot have more tuples than the number of distinct values in \( T.a_n \).

The above example condition is safe to draw conclusions from only if the base table does not share any foreign keys in which it is the dependent relation. Being a dependent relation means that some of the attributes will be populated with skolems (from the referenced attributes), thus, taking into consideration just matched attributes is no longer enough. For instance, assume that, for the above example, there is another target relation \( R(y_1, y_2, \ldots, y_m) \), where \( T.a_2 \rightarrow R.y_1 \) is a foreign key. Accordingly, the UTR table is of the following form: \( U(X, y_1, y_2, \ldots, y_m, a_1, a_3, \ldots, a_n) \), where \( a_2 \) is now omitted as it is represented by its referenced key, viz., \( U.y_1 \). Given that \( U.y_1 \) (representing both \( R.y_1 \) and \( T.a_2 \)) is a non-matched key attribute, skolem values will be generated on it. The skolem
values are created with the method described in Section 5.4.2. After the skolems for \( U.y_1 \) are created, the algorithm proceeds to generating labelled nulls for the key \( U.a_1 \) (representing \( T.a_1 \)): the function takes as parameters both \( T.a_n \) and \( T.a_2 \) as now, \( T.a_2 \) contains labelled nulls from the previous step for generating skolems (from \( U.y_1 \)). The challenge of correlating the possible number of distinct values with the estimated sizes of the base tables would improve the accuracy scores computed by Dynamap\(^X\).

### 6.2.3 Finding Merge Opportunities

The current method for finding the merge opportunities between intermediate mappings is based on matches and profile data, i.e., (partial) inclusion dependencies and (at least) one candidate key. However, in some situations, the conditions that the algorithm is guided by are not enough to decide whether to merge or not because there may be an inadvertent overlap that could lead to semantic inconsistencies. For instance, there may be an overlap between Location and City. Although some tuple alignment on these might make sense, it could also lead to erroneous correlations as they do not mean the same thing. This observation leads to the idea of a possible extension of Algorithm 1 by the use of semantic information, e.g., ontologies. The added information could make possible more complex comparisons between attributes that have value overlaps, i.e., comparisons at the semantic level.

In Sections 4.6 and 5.7, we briefly described the work done in this direction by Zhu et al. (2016); Castro Fernandez et al. (2018), and Nargesian et al. (2018), all of whom aim to find related sources based on the domain of the attribute values. A possible step further could be to adapt (one of) these techniques so that choosing the merge operator within Dynamap is done based on more informative descriptions of the relations and their attributes, where these descriptions are generated by (one of) their proposed methods.

### 6.3 Future Work

#### 6.3.1 Adapting to More Data Formats

Dynamap\(^{(X)}\) was developed assuming tabular data, but it includes only a subset of the possible relational transformations. Further work could extend the capabilities
6.3. FUTURE WORK

of Dynamap$^X$ to tackle scenarios that need more complex transformations on the sources. For instance, one future direction would be to adapt the mapping generation algorithm to detect the necessity of pivoting, i.e., transforming the unique values from one column into multiple columns in the output. This type of transformation would allow more complex merges between two sources become possible through pivoting. Also, a natural extension to Dynamap$^X$ would be the addition of new relational operators, e.g., aggregators, selections. For example, selection could be used to filter out the tuples that do not match a potential set of target requirements, e.g., output real-estate entries that have at least 2 bedrooms, excluding studios and 1-bedroom properties. With the current approach, for such a scenario, Dynamap$^X$ outputs mappings that correlate all sources (with all their information), and then, the output tuples generated with the mappings need post-processing to filter out the unnecessary information.

In a real-world data lake, the sources can have various formats, i.e., semi-structured, unstructured, not only relational. Probably one of the most natural directions in which the work in this thesis can be moved further is to adapt the algorithm to data models at the sources and/or the target. Tackling new data formats and adapting the current mapping generation approach would mean generating the same or similar types of merges between the sources of the same format (e.g., semi-structured), but also finding a way to merge two sources that have different data formats as this would mean bringing them in a common format so that a merge operation can be applied between them. Another possible path would be to abstract over the merge operators and introduce a translation methodology that would translate the abstract mappings to executable query languages according to the underlying data sources.

6.3.2 Mapping Generation Reuse

As seen in the results of the pruning experiments, the search space can grow quite rapidly so regenerating the mappings from scratch for each run can become time-consuming. A natural idea would be to cache mappings that can be reused in other mapping generation runs (e.g., with target schemas that are different, but similar, to what has been run before). This could improve with processing time as (parts of) sub-solutions could be loaded from cache, instead of regenerated. However, given that the mappings are generated with respect to a target schema and on the basis of profiling data computed for one particular state of dataset, caching the mapping
would mean that these should be stored under a set of characteristics which would define them as either suitable or unsuitable for reuse in certain scenarios with different (but similar) target schemas. Otherwise, reusing mappings that are not suitable for populating a target would lead to merge choices between sources that would erroneously populate the target. Also, the identification of the most effective and manageable extensions to this search space might be considered in future work, i.e., build a search space comprising reusable mappings that are easier to parse and to combine instead of searching through specific mappings (as it is done now).

This has already been the focus of recent work by Atzeni et al. (2019), where a method for generating meta-mappings is proposed. Meta-mappings are generic mappings over specific initial mappings. They propose a fitness function that characterizes and checks when a meta-mapping is suitable for a reuse scenario. An idea worth investigating could be to study the possibility of integrating the meta-mappings [Atzeni et al. (2019)] with the contributions of this thesis.

6.3.3 Mapping Generation Refinement

The open challenges mentioned in Sections 6.2.1 and 6.2.3 show a focus on keeping the mapping generation process automated, but with the addition of more informed decisions. However, it may not always be possible to fully automate the process in a setting where the input may not be correct, e.g., the matches are erroneous, or the profiling data is misleading with semantically incorrect attribute overlaps. In such a situation, the mapping generation would yield erroneous results and two ways of identifying the errors are through user input or a form of reference data (that is considered to be correct by definition). Feedback could be obtained in two ways: at the end of the mapping generation process, on the final output; or on intermediate results, thus, suppressing wrong decisions earlier in the process.

In the case of user feedback, the challenges brought by this research direction imply designing a suitable visualization of the mappings on which the feedback can be given; a method for providing user feedback on the mappings (or on the output data) considering the expertise level of the user; and a method of incorporating the feedback into the refinement process of the mappings, e.g., reusing the mappings that satisfy the user criteria and discarding the mappings that are build on merge opportunities that the user annotated as being incorrect.
6.3.4 Processing Optimization

Given the fact that the dynamic programming paradigm is at the core of the algorithm, Dynamap (without active pruning) does not scale over large repositories. In Chapter 4, we proposed a set of pruning strategies to address this challenge. However, this is only one type of optimization that can be done to improve the processing time. Some Dynamap components can run in parallel. For instance, the generation of the intermediate mappings (per iterations) can be done in parallel as long as other active threads (running for other iterations) do not depend on the one processed. The current algorithm would need a refactoring in terms of parallelism so as to improve the processing time, but without hindering the mapping generation.


of the 29th International Conference on Scientific and Statistical Database Management, SSDBM ’17, New York, NY, USA, pp. 41:1–41:4. ACM.


Appendices

A Helper Functions

A.1 CommonAncestors

This method is used in Algorithm 1 on line 3. Its purpose is to return the number of common ancestor relations for two input mappings. Two mappings can merge 2 or more initial source relations. CommonAncestors takes as input two mappings and checks how many initial relations have in common.

Considering the mappings as sets, CommonAncestors returns the number of elements in the intersection of the two sets.

Input: two mappings \( m_1, m_2 \)

Output: integer

A.2 DiffMatches

This method is used in Algorithm 1 on line 11. Its purpose is to return a boolean which is:

- true if the two input mappings match different target attributes
- false if the two input mappings match the same target attributes

Two mappings can match one or more target attributes. DiffMatches takes as input two sets of matches corresponding to two different mappings and checks if they match different target attributes.

Considering the matches of the two mappings as sets, DiffMatches returns true if the number of elements in the difference of the two sets is not equal to 0, and false otherwise.
A. HELPER FUNCTIONS

**Input:** two mappings \( m_1, m_2 \)

**Output:** boolean

*This method is the opposite of SameMatches method.*

### A.3 FindConnGraphs

This method is used in Algorithm 6 on line 8. Its purpose is to return a set of connected graphs from the input graph which may be disconnected.

For this method, we consider the input graph as an undirected graph (although it is a DAG - as it is the monolithic target graph), then a connected graph is a graph where there is a path from any point to any other point in the graph.

If the input graph is a connected graph, then the input and output will contain the same entity. Otherwise, the result is set of objects where an object is a connected graph.

**Input:** a (undirected) graph

**Output:** set of connected graphs

### A.4 FindMatchesAttr

This method is used in:
- Algorithm 9 on lines 3-4, and
- Algorithm 1 on lines 5-6.

Its purpose is to return the set of matches of the input mapping to the input target relation.

**Input:** one mapping \( m \), a target relation \( t \)

**Output:** the set of matches of mapping \( m \) to the target \( t \)

### A.5 FindMatchedKeys

This method is used in Algorithm 2 on lines 26-27. Its purpose is to return the set of keys for one input mapping where the source attributes match target attributes.

Each candidate mapping can have associated candidate keys that were assigned to it either through propagation or read from the input (on the base mappings). This method loops through the profile data, more specifically, through the set of
candidate keys for all mappings, and retrieves the ones that correspond to the input mapping.

**Input:** a mapping $m$, a target relation $target\_relation$

**Output:** set of keys in $m$ that have a matches to the input target relation

*The difference between this method and FindKeys is that in the latter, the algorithm does not check if the returned keys have matches to the target or not, where in the former, the returned keys need to be matching to the target.*

### A.6 FindPKey

This method is used in Algorithm 7 on line 5. Its purpose is to return the set of primary key for one input relation. If there are no primary keys in the input relation, the method returns null.

**Input:** a (target) relation $tr$

**Output:** the primary key $key$

### A.7 FindKeys

This method is used in Algorithm 2 on lines 8-9. Its purpose is to return the of keys for one input mapping. Each candidate mapping can have associated candidate keys that were assigned to it either through propagation or read from the input (on the base mappings). This method loops through the profile data, more specifically, through the set of candidate keys for all mappings, and retrieves the ones that correspond to the input mapping.

**Input:** sets of keys (part of profile data), and a mapping $m$

**Output:** set of keys in $m$

### A.8 IsFittest

Algorithm 8 is used to decide if the intermediate mapping has the highest fitness among the mappings that stem from the same initial relations, i.e., with exactly the same ancestor relations.

IsFittest is used in MergeMappings (on line 9) to determine whether to discard the newly generated mapping or not. This method takes as input the newly generated mapping and other mappings that stem from the same input
Algorithm 8 Check if a mapping has the highest fitness compared to mappings that stem from the same initial source relations.

1: function IsFittest(map, old_maps)
2: \ \
3: isFittest ← true
4: fitness ← Fitness(map, t)
5: memoizedMaps ← GetMemoizedMaps(map)
6: \ 
7: for each imap in old_maps do
8: \ if fitness(imap) > fitness then
9: \ isFittest ← false
10: \ return isFittest

relations and that were generated in the same iteration as the new map. First, on line 4, the fitness of the intermediate mapping is computed according to the fitness function. Then the intermediate mappings that stem from the same source relations, but were already memoized are retrieved (line 5). The fitness of each mapping is compared to the fitness of the new mapping, and if at least one previous mapping has a fitness higher than the new mapping (line 8), then we can decide that the new mapping is not the fittest (line 9) and it will not be memoized, i.e., will not be kept for further merges.

A.9 IsSubsumed

Algorithm 9 Predict subsumption between two mappings

1: function IsSubsumed(map1, map2,t_rel)
2: \ 
3: matchAttr1 ← FindMatchesAttr(map1,t_rel)
4: matchAttr2 ← FindMatchesAttr(map2,t_rel)
5: \ 
6: if ind.overlap = 1.0 then
7: \ subsumedMap ← map1
8: \ else
9: \ ind ← MaxInd(pd, matchAttr2, matchAttr1)
10: \ if ind.overlap = 1.0 then
11: \ subsumedMap ← map2
12: \ return subsumedMap

IsSubsumed (Algorithm 9) is used by ChooseOperatorDiff to
predict whether one mapping is subsumed by another mapping. First, in lines 3 and 4, it retrieves those attributes from the given mappings that have a match to the target, i.e., $\text{matchAttr}_1$ and $\text{matchAttr}_2$ for $\text{map}_1$ and $\text{map}_2$, respectively. In line 5, the algorithm searches in the profile data for inclusion dependencies that establish that the values of $\text{matchAttr}_1$ attributes are (partially) included in the corresponding values of $\text{matchAttr}_2$. If the overlap is equal to 1.0 (line 6), we conclude that the matched attributes of $\text{map}_1$ are subsumed by the matched attributes of $\text{map}_2$ (line 7), otherwise, in lines 8 to 11, the same process is repeated for the reverse case. The output of the algorithm is either the subsumed mapping, or null if neither is subsumed. Note that the definition is approximate, as table subsumption is assumed to exist where each attribute is subsumed by its counterpart.

A.10 MaxInd

This method is used in:
- Algorithm 9 on lines 5 and 9, and
- Algorithm 2 on line 10, and lines 17-18.

Its purpose is to return the inclusion dependency with maximum overlap between the two batches of attributes.

MaxInd loops through the input profile data, more specifically through the pool of inclusion dependencies, and searches for one inclusion dependency whose referenced attribute is in one input batch, e.g., $\text{attributes}_1$, and the dependent attribute is from the other input batch, e.g., $\text{attributes}_2$.

**Input:** profile data (containing inclusion dependencies), two sets of attributes $\text{batch}_1$ and $\text{batch}_2$

**Output:** one inclusion dependency, e.g., $I = S \subset_\theta P$, where $S \in \text{batch}_1$, and $P \in \text{batch}_2$, or vice versa, i.e., where $S \in \text{batch}_2$, and $P \in \text{batch}_1$, where $\theta$ is the maximum value out of overlaps of the set of inclusion dependencies that have both the referenced and dependent attributes in the two input batches.

A.11 SameMatches

This method is used in Algorithm 1 on line 8. Its purpose is to return a boolean which is:
- **true** if the two input mappings match exactly the same target attributes
- **false** if the two input mappings do not match the same target attributes.

Two mappings can match 1 or more target attributes. **SAME MATCHES** takes as input two sets of matches corresponding to two different mappings and checks if they match the same target attributes.

Considering the matches of the two mappings as sets, **SAME MATCHES** returns **true** if the number of elements in the difference of the two sets is equal to 0, and **false** otherwise.

**Input:** two mappings $m_1, m_2$

**Output:** boolean

This method is the opposite of **DIFF MATCHES** method.

## B Synthegrate

**Synthegrate**\(^2\) approach. The methodology behind Synthegrate is a top-down approach that starts with the creation of the target and then uses the target table(s) to create the source table(s) – which, at the end, are grouped in one or more source schemas. The target attributes are annotated such that they are populated with specifically created synthetic data using **Datafiller** [Coelho (2013)].

The algorithm starts the creation of the sources by using a target relation and its annotated target attributes. The relation is divided into two tables which, when merged, can recreate the table that was split. The resulting tables from the divisions become source relations in the integration scenarios. Having the top-down approach we can be sure that the generated ground-truth mapping, when executed on the source tables, recreates the same tuples as in the target table(s).

**Datafiller.** Datafiller [Coelho (2013)] is a tool that is used to generate synthetic data for relational instances. The method that Datafiller uses for creating synthetic data is by annotating the relations and/or the attributes in a relational schema, and based on those annotations, it creates synthetic data in a specific manner. In our settings, we use the annotations such that the data created on each attribute does not violate any required candidate keys, or any inclusion dependencies. Moreover, for each (partial) inclusion dependency, the annotations help in creating data with the indicated overlap over the attribute values involved.

\(^2\)https://github.com/MLacra/Synthegrate.git
in the inclusion dependency. The annotations that we use in defining the rules for the creation of the synthetic data are:

- **relation annotation:**
  - specify relation size:
    - `--df: size=<input size>`

- **attribute annotations:**
  - use dictionary (this is a file with a list of words):
    - `--df: use=<input file>`
  - how to parse the dictionary:
    - specify offset in the dictionary, i.e., the offset of the first word used to populate the attribute:
      - `offset=<input offset>`
    - the step with which the input dictionary is parsed, i.e., this will determine with which number the index of the words is modified:
      - `step=<input step>`
    - size of the dictionary – this helps when the parser reaches the end of the dictionary and it needs to circle back to the beginning of the dictionary list:
      - `size=<input dictionary size>`
  - prefix of data values in an attribute;
  - in order to control the creation of the attribute values complying to the inclusion dependencies constraints, all attributes involved in inclusion dependencies will have *unique* and *not null* constraints assigned. Otherwise, it becomes difficult to accurately compute values that respect the set overlaps for the required inclusion dependencies.

The above annotations are used in the step where the profile data is generated for the created source relations.

### B.1 Synthegrate Workflow

In this section we describe the workflow in Synthegrate for creating an integration scenario. Figure 1 shows the main steps in the generation of the components that comprise a scenario.

**Input parameters.** The parameters that Synthegrate expects are the ones that are used for varying the types of integration scenarios:

1. number of target relations (minimum of 1);
2. maximum number of source relations;
3. number of source schemas (minimum of 1 and a maximum equal to the number of source relations as we avoid creating empty schemas);
4. number of keys in the target schema;
5. cardinality ranges: minimum/maximum number of tuples;
6. number of join operations in the ground-truth mapping;
7. number of union operations in the ground-truth mapping;
8. maximum number of explicit foreign keys in the source schemas as foreign keys can be inferred between source relations of different schemas, but those are not explicitly stated;
9. arity range: minimum/maximum number of attributes in a (source or target) relation;
10. number of expected union candidates which are disjoint;
11. whether to reuse attributes for creating join conditions (in a split);
12. in a nested mapping, whether the union operations are expected to be performed first;
13. in a nested mapping, whether the join operations are expected to be performed first;

Target schema generation. For generating the target schema, Synthegrate uses the parameters specific to creating the target and the tables, i.e., cardinality and arity ranges, the number of target relations, and the number of foreign keys and primary keys in the target schema.

The first step is to create a target relation that has i) the arity equal to or greater than the number of required join operations, and ii) the size equal to or greater than the number of required union operations. The number of required join operations is equal to the number of join operations desired to appear in the ground-truth mapping (each target attribute might become a join condition attribute, for a foreign key), together with the number of foreign keys in the target schema. The number of required union operations is the number of union
operations desired to appear in the ground-truth mapping. After the first table is created, the algorithm loops into iteratively splitting a target table which is randomly chosen from the pool of created tables (at first it will be just one table) to create the target tables linked through foreign key constraints, i.e., it will create the target tables for a chain join comprising the required number of target foreign keys. After the required number of foreign keys is created, then Synthegrate incorporates them into a schema which represents the target schema.

In our setting, Synthegrate was used to create only single-table target schemas, thus, each scenario comprises a target schema that contains one target relation with a number of attributes equal to or greater than the number of required join operations in the ground-truth mapping. The created target relation is then used to create the source relations by splitting it, such that the source relations reconstruct it using the desired merge operations (we explain this step below). Synthegrate annotates the target relation and its attributes with parameters that are required by Datafiller for creating synthetic data. Given that we are fully controlling the creation of the synthetic data in the sense that we make sure all required profile data is created without any violations, Synthegrate will generate annotations that can accommodate the creation of the desired ground-truth mapping. For example, if the number of union operations is a large one, and we set the scenarios to create only disjoint union tables, then the cardinality of the created target relation cannot be smaller than the number of union operations as each table needs to comprise at least one tuple, otherwise it will create empty relations after splitting a table, e.g., as splitting one table with one tuple into two tables then one resulting table is empty and the other has the same tuple as the split table.

Source schema(s) generation. For the generation of the sources, Synthegrate has a similar approach to the one for creating target relations linked through foreign keys, i.e., chooses an already created table and splits it. For this step, it takes into consideration the parameters that determine the number and type of operations, i.e., joins or unions, how these should appear in the ground-truth mapping, i.e., whether it is expected for the sources to union first and then join or vice versa; how many sources to create, their arity and cardinality ranges; number of explicit foreign key constraints, i.e., the ones that are not declared explicitly are expected to be inferred through profile data even if the relations are part of different schemas; if the unionable relations are disjoint or not; and if
the attributes in the join conditions are already created attributes or if new ones need to be created.

At the beginning of this step, if both union first and join first parameters are false, then Synthegrate will randomly pick the next operation to be created; if either join or union are desired to be performed first in the ground-truth mapping, then the algorithm will choose the next operation accordingly. After the type of operation is chosen, it needs to pick the next relation to split. At the beginning, there are no source relations to choose from, but Synthegrate will use the target relation(s) to split as the target relations are the starting point for the creation of the sources. In our setting, there will be only one target relation to split, so, in the first loop, the target is split into two relations which, if merged through the chosen operation, recreate the target relation. Before starting to split the relation, the algorithm checks if the chosen relation is suitable for that type of split:

- the relation is join-extensible if the relation is not referenced by other tables (in a foreign key relationship) as it might preclude the inference of the (already created) foreign key,
- the relation is union-extensible if its size is at least two, as having just one tuple means one of the resulting relations will have no tuples.

After the relation is split, the attributes of the two relations are created and annotated such that they have the desired attribute overlaps, e.g., full inclusion dependencies for foreign keys, 0 overlap for disjoint unions, partial for overlapping unions. The annotations are done with the parameters specified in Section 4.4.2 for Datafiller. The values for these Datafiller parameters are computed to obtain the desired overlaps as following:

- for disjoint unions: randomly choose two numbers that add up to the cardinality of the split relation to be the cardinalities of the two newly created tables, while the offset for the first remains the same as the parent’s and the offset of the second becomes offset plus the new size of the first relation; the step and dictionary remain the same.
- for overlapping unions: randomly choose two numbers that add up to more than the cardinality of the split relation to be the cardinalities of the two newly created tables, while the offset for the first remains the same as the parent’s and the offset of the second becomes offset plus the new size of the first relation minus the difference between their sum and the parent’s cardinality (as that difference represents the number of overlapping values);
the step and dictionary remain the same.

- for inferred/explicit foreign key constraints: the cardinality of the split relation will be equal to the cardinality of the foreign key relation, while the cardinality of the referenced relation has a minimum value equal to the same number and maximum equal to the maximum of the cardinality range given as a parameter. The offset, step, and dictionary remain unchanged, the only thing that changes is the cardinality of the second relation.

The set of source relations in the output scenario is the set of relations that were not split. These relations are randomly spread across the specified number of source schemas (in the input parameters), provided that tables with explicit foreign keys are kept in the same schema.

Unmatched attributes creation. This step follows the creation of the source relations as all the attributes in the source relations are either needed in the target or needed to merge the relations. In this step, a random number of source attributes is added to the source relations, provided that their arity is within the range specified in the input parameters.

Profile data generation. In this step, Synthegrate generates profile information corresponding to the sources, i.e., candidate keys and (partial) inclusion dependencies.

The candidate keys are the primary keys or the attributes used in the creation of the join condition attributes, while the source attributes that do not have a match to the target or are not used in the merge conditions will not be candidate keys.

For inclusion dependencies, the overlaps between attributes need to be computed, and to this end, we use the relation and attribute annotations for Datafiller as these will define which type of synthetic data is generated for the attributes. In computing the overlaps, these annotations are used as described below:

Given two relations \( r_1 \) and \( r_2 \) with their attributes \( X \) and \( Y \), respectively, and their annotations:

For \( r_1.X \):
- \( \text{dictionary} = \text{dict} \)
- \( \text{offset} = o_1 \)
- \( \text{step} = st_1 \)
- \( \text{size} = ds_1 \) (the size of \( \text{dict} \))
- \( \text{size} = rs_1 \) (the size of the relation \( r_1 \) — this will correspond to the total
number of distinct values for \( r_1.X \)).

For \( r_2.Y \):
- \( \text{dictionary} = \text{dict} \)
- \( \text{offset} = o_2 \),
- \( \text{step} = st_2 \),
- \( \text{size} = ds_2 \) (the size of \( \text{dict} \)),
- \( \text{size} = rs_2 \) (the size of the relation \( r_2 \) – this will correspond to the total number of distinct values for \( r_2.Y \)).

One can assume that the two attributes might have some overlapping values as they draw values from the same dictionary \( \text{dict} \) and they have the same \( \text{step} \) value, thus, we compute their (possible) overlap:

1. compute the number of common values between \( r_1.X \) and \( r_2.Y \):
   \[
   cv = \min(o_1 + rs_1*st_1, o_2 + rs_2*st_2) - \max(o_1, o_2)
   \]
2. compute the overlap:
   \[
   \theta_{X,Y} = \frac{cv}{rs_1}
   \]

**Example B.1.** Given two annotated relations:

```sql
CREATE TABLE schema_1.relation_1 ( --df: size=150
  attribute_10 text,--df: use=dictionary_26 offset=299 step=1 shift=0 size=1102
  attribute_11 text,--df: use=dictionary_64 offset=299 step=1 shift=0 size=1102
  attribute_20 text--df: prefix=p7
 );

CREATE TABLE schema_1.relation_2 ( --df: size=599
  attribute_1 text,--df: use=dictionary_26 offset=0 step=1 shift=0 size=1401
  attribute_7 text,--df: use=dictionary_119 offset=0 step=1 shift=0 size=601
  attribute_18 text,--df: prefix=FkAn
  attribute_19 text--df: prefix=Cb
 );
```

Characteristics of the relations:
- the sizes of the relations are \(|\text{relation}_1| = 150 \) and \(|\text{relation}_2| = 599 \),
  given by the first annotation, e.g., \(--df: \text{size} \). These will correspond to the number of distinct values in all matched attributes of the two relations.
- it can be deduced that attributes \( \text{attribute}_20 \), \( \text{attribute}_18 \), and \( \text{attribute}_19 \) are attributes that do not match the target as they will have random values.
attached to the set prefixes, i.e., their values are not drawn from a dictionary, hence, they are not manipulated not to violate any constraints that are necessary for recreating the attribute values in the relation that was split for their creation.

- two inclusion dependencies will be generated between \textit{attribute\_1} and \textit{attribute\_10} as they have values drawn from the same dictionary, e.g. \textit{dictionary\_26}, and they have the same step value, e.g., \textit{step=1}. The overlaps are computed as following:

\[
\theta_{\text{attribute\_1,attribute\_10}} = \frac{\text{cv}}{599} = \frac{150}{599} = 0.2504
\]

Based on the two resulting overlaps, it is concluded that \textit{attribute\_1} is partially included in \textit{attribute\_10} as approximately a quarter of it is included, while the values of \textit{attribute\_10} are fully included by the values of \textit{attribute\_1}.

\textbf{Output scenario.} The output scenario comprises:

- two SQL scripts for the creation of the target schema and source schema(s);
- two SQL scripts annotated with the Datafiller parameters. These are the input to the Datafiller executable script, based on which Datafiller automatically populates the empty schemas.
- one SQL script with the ground-truth mapping (used to evaluate the correctness of any other mappings);
- matches between the sources and the target;
- profile data on the source relations, i.e., (partial) inclusion dependencies and candidate keys.

Given that the output scenario contains scripts, the source and target relations can be materialized in a relational database in order to run queries on them. After they are materialized, it can be checked that the output of the ground-truth mapping is exactly the same as the materialized target relation(s).