UNIVERSITY OF CALIFORNIA, IRVINE

Analysis-Aware Approach To Entity Resolution

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Hotham Altwaijry

Dissertation Committee:
Professor Sharad Mehrotra, Chair
Professor Dmitri Kalashnikov
Professor Chen Li
Professor Deva Ramanan

2015
DEDICATION

To my beloved parents, Abdulaziz and Aljawharah.
To my loving wife, Hana.
# TABLE OF CONTENTS

| LIST OF FIGURES                          | v    |
| LIST OF TABLES                          | vii  |
| ACKNOWLEDGMENTS                        | viii |
| CURRICULUM VITAE                       | x    |
| ABSTRACT OF THE DISSERTATION           | xi   |
| 1 Introduction                         | 1    |
| 2 ER Preliminaries and Related Work    | 5    |
| 2.1 ER Problem                         | 5    |
| 2.2 Standard Phases of ER              | 7    |
| 2.3 Related Work                       | 10   |
| 3 QDA: A Query Driven Approach To ER for Selection Queries | 14   |
| 3.1 Introduction                       | 14   |
| 3.2 Motivating Example                 | 16   |
| 3.3 Notation and Problem Definition    | 22   |
| 3.3.1 Standard Notation                | 22   |
| 3.3.2 Approach-Specific Notation       | 26   |
| 3.4 Vestigiality                       | 29   |
| 3.4.1 Triple \((p, \oplus, a_t)\) Categorization | 30   |
| 3.4.2 Multi-Predicate Selection Queries | 31   |
| 3.4.3 Creating and Labeling the Graph  | 35   |
| 3.4.4 Vestigiality vs. Minimality      | 38   |
| 3.4.5 Vestigiality Testing Using Cliques | 39   |
| 3.5 Query-Driven Solution              | 43   |
| 3.5.1 Overview of the Approach         | 43   |
| 3.5.2 Vestigiality Testing             | 45   |
| 3.5.3 Computing Answer of Given Semantics | 47   |
| 3.5.4 Answer Correctness               | 49   |
| 3.5.5 Optimizations of Equality and Range Queries | 49 |
3.5.6 QDA for Different Clusterings ........................................ 52
3.6 Experimental Evaluation .................................................. 54
3.6.1 Google Scholar Dataset Experiments ............................... 55
3.6.2 Hotels Dataset Experiments .......................................... 65
3.6.3 Discussion ............................................................... 69
3.7 Conclusions ................................................................. 70

4 QuERy: A Framework for Integrating ER with Query Processing 71
4.1 Introduction ................................................................. 71
4.2 Problem Setup ............................................................. 73
4.3 ER Preliminaries ............................................................ 76
4.3.1 Entity-sets .............................................................. 76
4.3.2 Standard ER Phases .................................................. 77
4.4 ER and Query Processing ................................................ 80
4.4.1 Queries and Query Trees ............................................ 81
4.4.2 Evaluating Set Values .............................................. 81
4.4.3 Standard Solution .................................................... 82
4.4.4 Problem Definition ................................................... 83
4.5 Polymorphic Query Trees ................................................ 84
4.5.1 Polymorphic Operators .............................................. 84
4.5.2 Equivalent Polymorphic Query Trees ............................. 87
4.6 Lazy-QuERy Solution ........................................................ 89
4.6.1 Creating Sketches ..................................................... 90
4.6.2 Query Plan Execution ............................................... 92
4.6.3 Discussion ............................................................ 95
4.7 Adaptive-QuERy Solution ................................................. 96
4.7.1 Sampling Phase ........................................................ 97
4.7.2 Adaptive Cost-based Cleaning .................................... 99
4.8 Experimental Evaluation ................................................ 102
4.8.1 Experimental Setup .................................................. 102
4.8.2 Performance Factors ................................................. 103
4.8.3 Products Dataset Experiments ...................................... 105
4.8.4 Synthetic Dataset Experiments .................................. 111
4.9 Conclusions ................................................................. 114

5 Conclusions and Future Work ............................................. 115
5.1 Conclusions ................................................................. 115
5.2 Future Work ............................................................... 116

Bibliography ................................................................. 120
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Standard Phases of ER</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Graph $G$ for $R$ (Table 3.1) Without $p_T$</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>A Symmetric Graph</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>CREATE-GRA PH(.) Function</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of Edge Labeling in Graph $G$. Unlabeled Edges Are unresolved</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Vestigiality vs. Minimality</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>IS-VESTIGIAL(.) Function</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>VESTIGIALITY-TESTING(.) Function</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>CHECK-POTENTIAL-CLIQUE(.) Function</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>COMPUTE-ANSWER(.) Function</td>
<td>48</td>
</tr>
<tr>
<td>3.10</td>
<td>QDA vs. TC [Time]</td>
<td>56</td>
</tr>
<tr>
<td>3.11</td>
<td>QDA vs. TC [#$R()$]</td>
<td>56</td>
</tr>
<tr>
<td>3.12</td>
<td>Answer Semantics</td>
<td>56</td>
</tr>
<tr>
<td>3.13</td>
<td>QDA Speed Up</td>
<td>58</td>
</tr>
<tr>
<td>3.14</td>
<td>$#R()$ Cost</td>
<td>60</td>
</tr>
<tr>
<td>3.15</td>
<td>Blocking [Time]</td>
<td>61</td>
</tr>
<tr>
<td>3.16</td>
<td>Blocking [%$R()$]</td>
<td>61</td>
</tr>
<tr>
<td>3.17</td>
<td>Edge Picking Strategy</td>
<td>61</td>
</tr>
<tr>
<td>3.18</td>
<td>QDA-CC vs. CC [#$R()$]</td>
<td>62</td>
</tr>
<tr>
<td>3.19</td>
<td>Algorithms’ Quality</td>
<td>63</td>
</tr>
<tr>
<td>3.20</td>
<td>QDA-CC Deviation</td>
<td>64</td>
</tr>
<tr>
<td>3.21</td>
<td>Q1: $price \leq t \land \text{stars} \geq 1 \land \text{country} = \text{US}'</td>
<td>66</td>
</tr>
<tr>
<td>3.22</td>
<td>Q2: $price \leq t \land \text{stars} \geq 2 \land \text{country} = \text{US}'</td>
<td>66</td>
</tr>
<tr>
<td>3.23</td>
<td>Q3: $price \leq t \land \text{stars} \geq 3 \land \text{country} = \text{US}'</td>
<td>66</td>
</tr>
<tr>
<td>3.24</td>
<td>Q4: $price \leq t \land \text{stars} \geq 4 \land \text{country} = \text{US}'</td>
<td>66</td>
</tr>
<tr>
<td>3.25</td>
<td>Q5: $price \geq t \land \text{stars} \leq 1$</td>
<td>67</td>
</tr>
<tr>
<td>3.26</td>
<td>Q6: $price \geq t \land \text{stars} \leq 3$</td>
<td>67</td>
</tr>
<tr>
<td>3.27</td>
<td>Q7: $\text{stars} \leq t \land \text{country} = \text{US}'</td>
<td>68</td>
</tr>
<tr>
<td>3.28</td>
<td>Q8: $\text{stars} \leq t \land \text{country} = \text{FR}'</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>Process Diagram of the Standard Solution (See Section 4.4.3)</td>
<td>75</td>
</tr>
<tr>
<td>4.2</td>
<td>Process Diagram of the QuERy Solution (See Section 4.6)</td>
<td>76</td>
</tr>
<tr>
<td>4.3</td>
<td>Query 1</td>
<td>82</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Relation $R$ – Some Papers Are Duplicates. $C_1 = {p_1, p_7}$, $C_2 = {p_2, p_3, p_4}$ and $C_3 = {p_5, p_6}$ Are 3 Clusters</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Clustering $C$ – Relation $R$ After Being De-duplicated Using TC</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Triple Categorization for Numerical Attributes</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Triple Categorization for Categorical Attributes</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Triples Generalization</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>A Collection of Raw Records from Two Distinct Data Sources</td>
<td>73</td>
</tr>
<tr>
<td>4.2</td>
<td>Cellphones Entity-set ($C$)</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Manufacturers Entity-set ($M$)</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Cellphones Object-set ($O^C$)</td>
<td>80</td>
</tr>
<tr>
<td>4.5</td>
<td>Manufacturers Object-set ($O^M$)</td>
<td>80</td>
</tr>
<tr>
<td>4.6</td>
<td>Relational and Polymorphic Relational Algebra Operators</td>
<td>88</td>
</tr>
<tr>
<td>4.7</td>
<td>Sketch $K^C_1$ for Block $C_1$ in Table 4.2</td>
<td>92</td>
</tr>
<tr>
<td>4.8</td>
<td>Decision Plane</td>
<td>100</td>
</tr>
<tr>
<td>4.9</td>
<td>Sketches for Entity-set $C$</td>
<td>106</td>
</tr>
<tr>
<td>4.10</td>
<td>Sketches for Entity-set $M$</td>
<td>106</td>
</tr>
<tr>
<td>4.11</td>
<td>Execution Time Breakdown</td>
<td>109</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Sharad Mehrotra for his support, guidance, and mentorship throughout my studies at University of California, Irvine (UCI). Prof. Mehrotra has been an invaluable source of ideas and inspiration. He has taught me how to identify new interesting problems and how not to fear tackling them. His confidence in me, support, and guidance was an important factor in my success.

I would like to offer my thanks to Prof. Dmitri Kalashnikov for his insightful support and suggestions throughout this research, and specifically on the third chapter of my thesis. Prof. Kalashnikov has helped me improve my writing skills through very useful feedback and for that I am grateful.

I am thankful to Prof. Michael J. Carey for his help and guidance while I was working under his supervision on the AsterixDB project during my first two years at UCI. The time and effort he spent with me were instrumental in my progress.

An additional gratitude is due to Prof. Chen Li and Prof. Deva Ramanan for finding the time to join my dissertation committee.

I would like to thank everyone in the ISG group. Specifically, I thank my wonderful (previous) and current colleagues in the Data Quality and Privacy Group at UCI, Yasser Altowim, Stylianos Doudalis, Kerim Oktay, Mehdi Sadri, Jie Xu, and Liyan Zhang. A special thank you goes to Yasser and Mehdi for being great officemates.

I extend my thankfulness to King Abdulaziz City for Science and Technology (KACST), Riyadh, Saudi Arabia, for providing me with generous Master and PhD Fellowships that allowed me to pursue my graduate studies.

I would also like to thank VLDB Endowment for permissions to include materials from “Query-Driven Approach to Entity Resolution” (VLDB 2013) into my thesis.

At a more personal level, I would like to thank my friends during my studies at UCI namely; Hashem Alayed, Mubarak Alhajri, Abdulaziz Alhussien, Mishari Almishari, Ali Alrajhi, Abdulrahman Alzaid, and Mohammed Alzaid for their moral help and humor.

A very special thanks to my good friends Sattam Alsubaiee and Yasser Altowim for their moral support and all the laughs and jokes that we have shared during the program. I can not think of better ones to have shared my graduate years with.

I would also like to thank my in-laws for their encouragement and support. I especially thank my brother-in-law Abdulaziz for all the fun times that we spent in Irvine.

I would also like to thank my brothers and sisters, Hesham, Haithem, Najla, Najwa, Nojoood, Hani, and Najd for being the best brothers and sisters ever. Many thanks to Najwa for
proofreading my papers and correcting my grammatical and punctuation errors, and to Hani for all the great times that we had while he was in San Diego.

Above all, I would like to thank my parents. I express my sincere gratitude to my father Abdulaziz who has always been a tremendous source of inspiration and support, and to my mother Aljawharah for her endless source of unconditional love and unwavering encouragement; without them I would not be the person that I am today.

Last, but certainly not least, I am also indebted to my loving wife Hana for her endless love and constant support; without her this work would never come to light. I would like to thank her for always taking care of me and standing beside me. Hana, I love you!
CURRICULUM VITAE

Hotham Altwaijry

EDUCATION

University of California– Irvine, Irvine, CA, USA 2015
Doctor of Philosophy, Information and Computer Science.

University of Southern California, Los Angeles, CA, USA 2008
Master of Science, Computer Science.

King Saud University, Riyadh, Saudi Arabia 2005
Bachelor of Science, Computer Science.

PUBLICATIONS


SELECTED AWARDS

2008-2015: PhD Fellowship, King Abdulaziz City for Science and Technology.
2014: Travel Award Fellowship, VLDB.
2007-2008: Master Fellowship, King Abdulaziz City for Science and Technology.
2005: Second Class Honors, King Saud University.
2004: Distinction Award Certificate, King Saud University.
2003: Distinction Award Certificate, King Saud University.
In the era of big data, in addition to large local repositories and data warehouses, today’s enterprises have access to a very large amount of diverse data sources, including web data repositories, continuously generated sensory data, social media posts, clickstream data from web portals, audio/video data capture, and so on. As a result, there is an increasing demand for executing up-to-the-minute analysis tasks on top of these dynamic and/or heterogeneous data sources by modern applications. Such new requirements have created challenging new problems for traditional entity resolution, and data cleaning in general, techniques. In this thesis, we respond to some of these challenges by developing an analysis-aware approach to entity resolution.

First, we explore the problem of analysis-aware data cleaning in the context of selection queries. Specifically, we propose an “on-the-fly” data cleaning framework in the context of SQL-like selection queries. The objective of this framework is to perform the minimal number of cleaning steps that are required to answer a user query correctly. Our approach leverages the concept of vestigiality to reduce cleaning overhead. We conducted a comprehensive empirical evaluation of the proposed solution to demonstrate its significant advantage in terms of efficiency over the traditional techniques for the given problem settings.

Subsequently, we study analysis-aware data cleaning for the more general case where queries...
can be complex SQL-style selections and joins. In particular, we develop a framework for integrating entity resolution techniques with query processing. The aim of this framework is to utilize the query semantics to reap the benefits of early predicate evaluation while still minimizing redundant computation in the form of data cleaning. This framework relies on the notion of polymorphic operators, which are analogous to the common relational algebra operators with one exception: they know how to test the query predicates on the dirty data prior to cleaning it. We conducted extensive experiments to evaluate the effectiveness of our approach on real and synthetic datasets.

Overall, our experiments demonstrate outstanding results – that is our analysis-aware approaches are significantly better compared to traditional ER techniques, especially when the query is very selective.
Chapter 1

Introduction

This thesis addresses the problem of analysis-aware data cleaning, wherein the needs of the analysis task dictates which parts of the data need to be cleaned. Analysis-aware cleaning is emerging as a new paradigm for data cleaning to support today’s increasing demand for (near) real-time analytical applications of big data. Modern enterprises have access to potentially limitless data sources such as web data repositories, social media posts, clickstream data from web portals, and so on. Analysts usually wish to integrate one or more such data sources (possibly with their own data) to perform joint analysis and decision making. Likewise, a user can dynamically find data in external data sources and connect it with her data. For example, a small store owner may discover an online source (e.g., a web table) containing Amazon’s product pricing and may wish to compare that pricing with her own pricing.

Several systems have been developed to empower analysts to dynamically discover and merge data sources. For instance, Microsoft Power Query [75] provides features to dynamically find, combine, visualize, share, and query data across a wide variety of online and offline sources. Another example is Trifacta [9], a data transformation platform that employs a predictive
interaction framework [45] to enable users to transform raw data into structured formats. However, to the best of our knowledge, such systems have not yet incorporated data cleaning mechanisms.

As a result of merging data from a variety of sources, a given real-world object may often have multiple representations, resulting in data quality challenges. In this thesis, we focus on the Entity Resolution (ER) challenge [21, 26, 31, 77, 81], the task of which is to discover duplicate entities that refer to the same real-world object and then to group them into a single cluster that uniquely represents that object.

Traditionally, entity resolution, and data cleaning in general, is performed in the context of data warehousing as a well-engineered offline preprocessing step prior to making data available to analysis – an approach that works well under standard settings. Such an offline strategy, however, is not viable in emerging applications that deal with big data analysis. First, the need for (near) real-time analysis requires modern applications to execute up-to-the-minute analytical tasks, making it impossible for those applications to use time-consuming standard back-end cleaning technologies. Another reason is that in the data analysis scenarios that motivate our work, an analyst (or a user) may discover and analyze data as part of a single integrated step. In this case, the system will know “what to clean” only at analysis time (while the user is waiting to analyze the data). Last, given the volume and the velocity of big data, it is often infeasible to expect that one can fully collect or clean the data in its entirety.

Recent work on analysis-aware data cleaning seeks to overcome the limitations of traditional offline data cleaning techniques [17, 22, 23, 36, 49, 68, 70, 73]. While these existing solutions address analysis-aware data cleaning to a degree, they are limited to only simple queries (viz., mention queries and/or numerical aggregation queries) executed on top of dirty data. Data analysis, however, often requires a significantly more complex type of queries requiring SQL-style selections and joins. For example, consider a user searching for well-cited publica-
tions, e.g., those with a citation count above or equal to, say, 45 of a researcher named “Alon Halevy” on Google Scholar. As another example, suppose that a user interested in comparative shopping may wish to find cellphones that are listed on two distinct data sources: Best Buy and Walmart to compare their ratings and reviews. Clearly, the query that corresponds to the user’s first interest will require a SQL-like selection while her second interest will require a SQL-like join query. In contrast to this thesis, the existing analysis-aware approaches cannot exploit the semantics of such selection and/or join predicates to reduce the amount of cleaning.

In this thesis, we explore the problem of analysis-aware data cleaning for complex SQL-style selections and joins spanning single and/or multiple dirty entity-sets. In this thesis, we develop analysis-aware approaches that leverage the query semantics to reduce the amount of cleaning and thus minimize the total execution time of the user’s query for two different classes of SQL queries: selections and joins. A key concept driving our approaches is that of vestigiality. A cleaning step is vestigial (viz., unnecessary) if our techniques can guarantee that they can still compute a correct final answer without knowing the outcome of this cleaning step.

In particular, in Chapter 3, we propose QDA [15], a query-driven approach to entity resolution that systematically exploits the semantics of the selection predicates to reduce the data cleaning overhead. The objective of QDA is to issue the minimum number of cleaning steps that are necessary to answer a given query correctly. Moreover, we formalize the concept of vestigiality in the context of SQL selection queries. Finally, we provide a comprehensive empirical evaluation to present QDA’s considerable advantage over traditional entity resolution techniques.

While QDA, described in Chapter 3, provides a strong foundation for analysis-aware data cleaning, it is limited to only selection queries executed on top of a single entity-set containing duplicates. Data analysis, however, often requires a significantly more complex type of
queries requiring SQL-style joins, e.g., the query that corresponds to the user’s second interest presented above. As a result, in Chapter 4, we propose QuERy, a novel framework that integrates ER with query processing to efficiently and accurately answer complex SQL-style selection and join queries issued on top of dirty data. The predicates in those queries may be associated with any attribute in the entity-sets being queried. In this chapter, we introduce and formalize the notion of polymorphic operators – a key concept in QuERy. We finally conduct extensive experiments to evaluate the effectiveness of our solutions on real and synthetic datasets.

Currently, the main traditional approach for improving the efficiency of ER is that of blocking\(^1\) [46,50,60]. QDA as well as QuERy are entirely new complementary paradigms for improving the efficiency of ER: they are different from blocking and, as we will show, are typically much more effective in conjunction with blocking. However, unlike blocking, QDA and QuERy are not generic as they are meant primarily for analysis-aware tasks. Both approaches (i.e., QDA and QuERy) exploit the specificity and the semantics of the given SQL query to significantly reduce the cleaning overhead by only cleaning those entities that may influence the answer of the query. They compute answers that are equivalent to those obtained by first using a standard cleaning algorithm such as Swoosh [21] or DepGraph [31], and then querying on top of the cleaned data. However, we will see that in many cases QDA as well as QuERy can compute these answers much more efficiently.

The rest of this thesis is organized as follows. Chapter 2 covers the ER preliminaries and related work. Our QDA approach is described in Chapter 3. Subsequently, Chapter 4 presents our QuERy framework. Finally, we conclude this thesis and discuss future extensions in Chapter 5.

---

\(^1\)Blocking is the main divide-and-conquer style efficiency technique in ER. It utilizes blocking functions to partition the dataset into (possibly overlapping) smaller blocks. See Chapter 2 for more details.
Chapter 2

ER Preliminaries and Related Work

2.1 ER Problem

Before we describe our solutions on enabling analysis-aware data cleaning, in this section, we briefly review the entity resolution problem (which is the focus of this thesis) to provide context to our research.

In 1959, Newcombe et al. [62] defined the entity resolution (ER) problem. Ten years later, it was formalized by Felligi and Sunter [35]. Ever after, the ER problem has been studied under various names (e.g., deduplication [66], entity matching [57], record linkage [81], reference reconciliation [31], merge/purge [46], etc.) by different fields of Computer Science including the Database field, the Natural Language Processing field, and the Artificial Intelligence field to name a few.

Entity resolution [21, 26, 35, 46, 57, 62, 77, 81] is a very common data quality challenge that arises when real-world objects are referred to using entities or descriptions that are not always a unique identifier of the objects. Researchers often distinguish between two major
types of entity resolution: *Lookup* and *Grouping* [24,61]. In the case of lookup, there exists a set of known-to-be-clean objects (often referred to as Master Data) and the objective of ER is to validate (incoming) dirty entities by looking them up against this clean set of objects. In the case of grouping, no set of clean objects exists and the task of ER is to first discover duplicate entities that refer to the same real-world object and then to group them into a single cluster that uniquely represents that object. In entity resolution, the word *entity* usually describes a real-world object (e.g., a human-being, an organization, a place, etc.) and the word *resolution* is used since the objective of ER is to *resolve* the question “do the entities represent the same real-world object or not?”. In this thesis, we are primarily interested in instances of the grouping problem. (See Sections 3.3 and 4.4 for definitions of our problems.)

In the Database field, the ER problem has been identified as one of the critical data cleaning challenges, both (i) in the context of traditional data warehousing, where ER is performed during the Extract-Transform-Load (ETL) process as a well-engineered offline step prior to making the data available to analysis [16], and (ii) in the context of big data analytics\(^2\), where the three V’s of big data (viz., Volume, Velocity, and Variety) have profound influence on every phase of the data analysis pipeline – from data acquisition, cleaning, integration, analysis, to interpretation [58].

It is well-recognized that the ER problems are pervasive in data extraction and integration contexts. They arise due to variety of reasons: it could be a result of human error during data entry – resulting in misspelling, partially provided information, different acronyms, differences in representation, etc. ER problems are even more severe in the context of automated

\(^2\)Data cleaning in the context of big data is beginning to be recognized as an important challenge [58]. Many popular media as well as academic articles have highlighted challenges such as entity resolution as amongst the most important and immediate roadblocks for big data analytics. For instance, a recent blog posted in May 2014 on http://www.datasciencecentral.com calls out entity resolution as one of the biggest roadblocks for big data analytics. Moreover, recent database conferences such as the International Conference on Data Engineering 2014 includes a special tutorial entitled *Data quality: the ‘other’ face of big data* that highlights some of the issues when cleaning big data [65].
information extraction (e.g., entity extraction). In such scenarios, different sources may de-
scribe the same real-world object in very different styles and/or describe different real-world
objects in a similar fashion. Furthermore, complexity builds due to extractor errors that
may not be able to fully extract contextual information hidden in the source necessary to
uniquely identify the real-world object.

2.2 Standard Phases of ER

A typical ER cycle consists of several phases of data transformations, which can be inter-
mixed. Taken together, these phases are shown in Figure 2.1. These phases include:

- *Data preparation*, herein, the ER framework restructures the raw data into a form that
  is more convenient for data consumption. This step consists of different mini-steps
  including *encoding* that translates the input data from one encoding to a different en-
  coding, *conversion* that transforms the data format from one representation to another,
  *standardization* that normalizes the representation of the data into a common format,
Recently, various systems have been developed to enable analysts to (interactively) restructure their data. For instance, Trifacta [9] is a data transformation platform that employs a predictive interaction framework [45] to enable users to transform raw data into structured formats. In addition, Data Wrangler [54] is an interactive tool that has a visual programming interface which allows users to restructure input datasets prior to analysis. Another system is C4, developed at SourceThought [5] – C4 provides the analysts with the ability to shape, ingest, and join their data on Hadoop. In addition to these systems, other data transformation tools exist such as OpenRefine (formerly Google Refine) [2] and SnapLogic [8].

- **Blocking** is the main divide-and-conquer technique used to improve the efficiency of ER approaches. The objective of blocking is to reduce the computationally expensive quadratic pairwise comparisons of entities in the dataset (viz., $O(n^2)$ comparisons where $n$ is the dataset size) by applying such pairwise comparisons on much smaller blocks (i.e., $O(b^2)$ comparisons per cleaning block of size $b$, note that $b \ll n$).

There exists a wealth of literature on blocking techniques including standard blocking techniques such as [35,50], the sorted neighborhood (SN) method [46], and the canopy clustering approach [60]. A comparison of various blocking techniques can be found in [20,27].

A standard blocking technique divides the dataset into (possibly overlapping) smaller blocks. Such a technique uses relatively cheap blocking functions, which are applied on one (or more) attribute(s) called blocking key(s), to partition the entities such that (i) if two entities might co-refer, they will be placed together into at least one block and (ii) if two entities are not placed together into at least one block, they are highly unlikely to co-refer. A blocking function that divides entities based on the first $x$ characters of a blocking key is an example of a standard blocking technique.

The SN method starts by sorting the entities in the dataset based on a sorting key
assuming that entities that are close (in the sorted list) are more likely to be duplicates. After that, it moves a fixed-size window sequentially over the sorted list and compares all pairs of entities that are within the same window at any point.

In canopy clustering, two threshold distances (viz., the loose distance threshold and the tight distance threshold) are utilized to partition the entities into canopies (i.e., blocks). In the first step, the approach randomly removes an entity \( e_i \) from the dataset and inserts it into a new canopy \( C \). In the second step, for each entity \( e_j \) that is still in the dataset, the method puts \( e_j \) in \( C \) if the distance between \( e_i \) and \( e_j \) is less than the loose distance threshold. In the third step, if the distance between \( e_i \) and \( e_j \) is less than the tight distance threshold, then the technique removes \( e_j \) from the dataset.

Next, the technique proceeds iteratively from step one. It terminates when the dataset is empty.

In this thesis, we utilize standard blocking techniques to partition our datasets (See Sections 3.6 and 4.8).

- **Problem Representation** that maps the data into an internal data representation of the ER algorithm such as a graph. For instance, the approach in [52] views the database as a graph of interconnected entities (modeled as vertices) that are linked to each other via relationships (modeled as edges). Moreover, the technique in [31] employs a graph-based model for iterative ER. Specifically, it uses a dependency graph, whose vertices represent the possibility that a pair of entities may be duplicates and whose edges represent the dependencies between the matching decisions, to propagate the similarity/dissimilarity results. Similarly, many other ER approaches such as [14,15,25] represent their ER problem as a graph.

- **Similarity Computation** determines for each pair of entities within the same block whether they co-refer or not. This phase is often computationally expensive as it might require comparing every pair of entities in the same block (i.e., \( O(b^2) \) pairs of entities per cleaning block of size \( b \)) using a compute-intensive application-specific
resolve/match function.

Traditional resolve/match functions analyze the similarity of entities to determine if they co-refer [35, 46, 62]. Recently new approaches exploit new information sources such as analyzing context [16, 26, 83], exploiting relationships between entities [52], domain/integrity constraints [34], behaviors of entities [81], external knowledge bases such as ontologies and web search engines [32, 53, 63], and human intelligence in the form of crowdsourcing [72, 74, 79] to determine if two entities refer to the same real-world object or not.

• **Clustering** attempts to accurately group duplicate entities into a set of non-overlapping clusters based on the values computed by resolve/match functions such that each cluster represents one real-world object and each real-world object is represented by one cluster.

  Several clustering techniques have been proposed in the literature including transitive closure-based algorithms [21], the center algorithm [44], the correlation clustering algorithm [19], the Markov cluster algorithm [71], etc.

• **Merging** consolidates entities of each individual cluster into a single representative object that will represent the cluster to the end-user or application in the final result. For instance, the approach in [21] assumes the existence of a black-box merge function to consolidate duplicate entities. Moreover, in [15], we merge the duplicate entities by utilizing a combine function per attribute.

### 2.3 Related Work

Entity resolution is a well-recognized data quality problem that has received significant attention in the literature over the past few decades, e.g., [21, 26, 35, 46, 57, 62, 77, 81]. A thorough overview of the existing work in this area can be found in several surveys [30, 33, 57].
and workshops/tutorials [37,38,65].

Efficiency, along with quality, has always been key traditional challenges of entity resolution. On the one hand, research on efficiency has focused on improving the runtime of the ER algorithms by exploiting blocking techniques [11,35,39,46,50,60,64] and/or parallel computing frameworks such MapReduce or Pregel [56,59,67]. On the other hand, the research on quality has concentrated on how to exploit additional information sources (e.g., relationships between entities, domain/integrity constraints, behaviors of entities, ontologies, web search engines, etc.) to improve the quality of ER [16,26,32,52,53,63,72,74,79,81,83].

Given the increasing demand of (near) real-time analytical applications, recent research has begun to consider new ER approaches like analysis-aware ER, progressive ER, incremental ER, and so on.

**Analysis-aware ER.** The work on analysis-aware ER has been proposed in [17,22,23,36,49,68,70,73]. The approach in [22] answers mention-matching queries (e.g., retrieve all papers written by author “J. Smith”) collectively using a two-phase expand/resolve algorithm. It retrieves the related records for a query using two expansion operators and then answers the query by only considering the extracted records. Likewise, reference [70] uses an unsupervised online ER method to identify duplicates in the results of a mention matching query issued on multiple Web databases. The authors in [23] proposed a technique (which utilizes batching) to speedup the computations of top-k entity recognition queries against a dictionary. Furthermore, a more recent analysis-aware ER approach has been developed in [73], where the authors presented a framework that applies data cleaning to only a small sample of the data and then uses the results of the cleaning process to approximately answer aggregate numerical queries. Unlike this thesis, these approaches do not consider optimizing for other complex types of SQL queries (e.g., selections and joins).

Even though the ER techniques in [17,36,49] are analysis-aware, they solve different prob-
lems. For example, the approach in [36] considers answering queries over databases that may violate a set of integrity constraints by resolving such inconsistencies on-the-fly at query time. Moreover, the technique proposed in [17] attempts to answer queries over inconsistent databases in the presence of probabilistic information about potential duplicates. In contrast to our work, where we utilize ER algorithms to resolve the dirty data, these two works use query rewriting algorithms to return consistent answers. A different probabilistic approach is studied in [49] – it resolves queries under data uncertainty by connecting ideas of record linkage and probabilistic databases. The term query refers to a combination of key/value pairs and each entity returned as an answer is accompanied by a probability that this entity will be selected amongst all possible worlds. Other probabilistic solutions that deal with duplicate records and data uncertainty are Dataspaces [42] and Trio [10]. These two techniques create database systems that support uncertainty along with inconsistency and lineage.

Reference [68] handles entity uncertainty at query time for OLAP applications. Unlike ours, this work assumes the existence of a record-to-cluster mapping table and its goal is to answer group-by OLAP queries by returning results in the form of strict ranges.

Other New ER Approaches. Several approaches, e.g., [14, 51, 80], are considering how to clean the data progressively, while interactively analyzing the partially cleaned data to compute better results. Such approaches aim to prioritize the data parts that need to be cleaned first in such a way that if cleaning is stopped, then they can get the best possible quality for that cleaning time. Furthermore, various approaches have begun to consider incremental cleaning techniques [40, 78]. Such techniques address the problem of maintaining an up-to-date ER result when data updates arrive quickly. Instead of recomputing the ER result each time from scratch. These approaches provide mechanisms that leverage the previous ER result to efficiently compute the updated one.
In addition to such approaches, the ER research community is exploring other novel directions. For example, Data Tamer [69] is an end-to-end data curation system that entails machine learning algorithms with human input to perform schema integration and entity resolution. Another system is NADEEF [29], which is a general-purpose data cleaning and repair system that provides appropriate programming abstractions for users to specify data cleaning transformations.
Chapter 3

QDA: A Query Driven Approach To ER for Selection Queries

3.1 Introduction

In this chapter, we propose QDA, a query-driven approach to entity resolution that systematically exploits semantics of selection predicates to reduce the data cleaning overhead. The key insight is that the knowledge of the query can be exploited to significantly reduce the amount of cleaning required.

A key concept driving the QDA approach is that of vestigiality. A cleaning step (i.e., call to the resolve function for a pair of records) is called vestigial (viz., unnecessary) if QDA can guarantee that it can still compute a correct final answer without knowing the outcome of this resolve. We formalize the concept of vestigiality in the context of a large class of SQL selection queries and develop techniques to identify vestigial cleaning steps. Technical challenges arise since vestigiality, as we will show, depends on several factors, including the specifics of the merge function used if two entities are duplicates, the predicate associated
with the query, and the query answer semantics of what the user expects as the result of the query. We show that determining vestigiality is NP-hard and we propose an effective approximate solution to test for vestigiality that performs very well in practice.

The main contributions of this chapter are:

- Introduction of QDA, a query-driven approach to entity resolution that systematically exploits semantics of selection predicates to reduce overhead of data cleaning (Sections 3.2 and 3.3).

- Introduction of the concept of vestigiality of certain computations in the context of a solution for SQL selection queries (Section 3.4).

- Development of query-driven techniques that leverage the concept of vestigiality to reduce computation (Section 3.5).

- Demonstrating that, interestingly, even though we explain QDA in the context of transitive closure-based algorithms, this technique is generic and could be applied to different types of clustering techniques as well (Section 3.5.6).

- Extensive empirical evaluation of QDA (Section 3.6).

The rest of this chapter is organized as follows. A motivating example is presented in Section 3.2. The problem definition is provided in Section 3.3. Section 3.4 explains the concept of vestigiality. Our solution is described in Section 3.5 and tested in Section 3.6. Finally, we conclude this chapter in Section 3.7.
3.2 Motivating Example

Before formalizing the notion of vestigiality and developing QDA, in this section we describe the main idea behind our query-driven solution using an illustrative example.

**Setup.** Consider a user searching for bibliographic information of a researcher named “Alon Halevy” in Google Scholar, the results of which are shown in Table 3.1. Let us further assume that all the publications are written by the same author, but some of the papers returned could be duplicates. Such duplicates possibly arise as a result of the method Google uses to get its information: Google crawls research publications and then extracts citations. Since the same paper might be referred to differently in different publications, paper clusters could be split in parts, leading to duplication. In Table 3.1, papers \( p_1, p_7 \), \( p_2, p_3, p_4 \), and \( p_5, p_6 \) are duplicates and refer to the same real-world entities. Hence, they should be clustered into three clusters \( C_1, C_2 \) and \( C_3 \) by an ER algorithm.

Suppose that the user is actually *not* interested in all papers of “Alon Halevy”, but only in well-cited ones, e.g., those with a citation count above or equal to, say, 45. The following query represents the user’s area of interest:

\[
\text{Query 1. } \text{SELECT } * \text{ FROM } R \text{ WHERE cited } \geq 45
\]

When Query 1 is issued on Table 3.1 prior to cleaning it, the results are \( p_1 \) and \( p_7 \), corresponding to cluster \( C_1 \). This is incorrect since the second paper cluster \( C_2 \) has a citation count equal to 60 \( \geq 45 \) and should also be returned.

**Standard Solution.** The standard way to answer Query 1 is to first deduplicate relation \( R \) to create merged profiles of each paper and then compute the query over this clustering. Suppose that we use a variant of the transitive closure (TC) algorithm for this purpose,

---

3 This running example is synthetically generated and is only used for illustration purposes.
Table 3.1: Relation $R$ - Some Papers Are Duplicates. $C_1 = \{p_1, p_7\}$, $C_2 = \{p_2, p_3, p_4\}$ and $C_3 = \{p_5, p_6\}$ Are 3 Clusters

similar to [21]. TC uses a pairwise resolve function to compare records and a pairwise merge function to consolidate two matching records. It merges two records $p_i$ and $p_j$ as soon as resolve returns $\text{true}$ to produce a new combined record $p_i \oplus p_j$, but it does not merge them when resolve returns $\text{false}$.

Let the order in which the TC algorithm invokes resolves be:

1. $\mathcal{R}(p_1, p_7) = \text{true}$,
2. $\mathcal{R}(p_1 \oplus p_7, p_2) = \text{false}$,
3. $\mathcal{R}(p_1 \oplus p_7, p_3) = \text{false}$,
4. $\mathcal{R}(p_1 \oplus p_7, p_4) = \text{false}$,
5. $\mathcal{R}(p_1 \oplus p_7, p_5) = \text{false}$,
6. $\mathcal{R}(p_1 \oplus p_7, p_6) = \text{false}$,
### Table 3.2: Clustering $\mathcal{C}$ – Relation $R$ After Being De-duplicated Using TC

<table>
<thead>
<tr>
<th>cluster</th>
<th>p_id</th>
<th>p_title</th>
<th>cited</th>
<th>venue</th>
<th>authors</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$p_1 \oplus p_7$</td>
<td>Towards efficient entity resolution</td>
<td>110</td>
<td>Very Large Data Bases</td>
<td>Alon Halevy</td>
<td>2000</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$p_2 \oplus p_3 \oplus p_4$</td>
<td>Entity-Resolution on dynamic data</td>
<td>60</td>
<td>Proc of ACM SIGMOD Conf</td>
<td>Alon Halevy, Jane Doe</td>
<td>2005</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$p_5 \oplus p_6$</td>
<td>Entity-Resolution for census data</td>
<td>15</td>
<td>Proc of ICDE Conf</td>
<td>Alon Halevy</td>
<td>2002</td>
</tr>
</tbody>
</table>

7. $\mathcal{R}(p_4, p_5) = \text{false}$,

8. $\mathcal{R}(p_2, p_3) = \text{true}$,

9. $\mathcal{R}(p_2 \oplus p_3, p_5) = \text{false}$,

10. $\mathcal{R}(p_2 \oplus p_3, p_6) = \text{false}$,

11. $\mathcal{R}(p_4, p_6) = \text{false}$,

12. $\mathcal{R}(p_5, p_6) = \text{true}$,

13. $\mathcal{R}(p_2 \oplus p_3, p_4) = \text{true}$,

14. $\mathcal{R}(p_2 \oplus p_3 \oplus p_4, p_1 \oplus p_7) = \text{false}$,

15. $\mathcal{R}(p_2 \oplus p_3 \oplus p_4, p_5 \oplus p_6) = \text{false}$,

16. $\mathcal{R}(p_5 \oplus p_6, p_1 \oplus p_7) = \text{false}$.

Above, “$\mathcal{R}(p_i, p_j) = \text{true}$” or “$\mathcal{R}(p_i, p_j) = \text{false}$” refers to the resolve function and its outcome (i.e., either true or false). Table 3.2 shows clustering $\mathcal{C} = \{C_1, C_2, C_3\}$ of relation $R$ after applying TC. In $\mathcal{C}$ the duplicate publication records are merged into clusters, where the notation $p_i \oplus p_j$ denotes the merged representation of papers $p_i$ and $p_j$. The clusters $C_1$, $C_2$, and $C_3$ have citation counts of 110 ($= 65 + 45$), 60 ($= 25 + 20 + 15$), and 15 ($= 10 + 5$),

---

4In this example, we use a more traditional version of TC which treats false “softly”, that is, $p_i$ and $p_j$ are not merged at the moment due to a lack of evidence supporting a merge, but clusters containing $p_i$ and $p_j$ may be merged later on. Other versions of TC might treat a false, in a “hard” fashion, meaning that clusters containing $p_i$ and $p_j$ are not merged, now or in the future.
respectively.

In the above execution, 16 calls to the (potentially expensive) resolve function are made by TC. Query 1 on Table 3.2 returns the clusters $C_1$ and $C_2$ corresponding to $p_1 \oplus p_7$ and $p_2 \oplus p_3 \oplus p_4$, respectively.

**Returned Answer Semantics.** Before we illustrate how the knowledge of a query can be exploited to reduce the number of resolves invoked by the original cleaning algorithm, we first need to discuss the guarantees the QDA system can provide regarding the answer returned to the user. The system provides a trade-off between the strictness of the chosen query answer semantics and the efficiency of query processing, as explained below.

QDA is said to follow *exact* semantics if the returned results match (in terms of both the clusters returned and their representations) the results returned by first cleaning the data (using TC) and then querying it (as in the standard solution above). For instance, when QDA follows *exact* semantics it must return $p_1 \oplus p_7$ (representing $C_1$) and $p_2 \oplus p_3 \oplus p_4$ (representing $C_2$) as the answer for Query 1. Note that such an answer will not have duplicates since each returned cluster will have a single representation.

An alternate answer might be $p_1 \oplus p_7$ (representing $C_1$) and $p_2 \oplus p_3$ (representing $C_2$). Note that the representation of $C_2$ is different (e.g., the value of the *cited* attribute is $25 + 20 = 45$, instead of 60). Here, the system does not provide the canonical merged representation as the TC algorithm; it does, however, return a semantically equivalent answer, since both $C_1$ and $C_2$ are represented in the returned answer. Yet again, each cluster is returned exactly once without duplication. In this case, QDA is said to follow *distinct* semantics.

Another possible returned answer, which provides more savings in terms of cleaning, is where duplicates are allowed to appear in the answer. For example, an answer such as $p_1$, $p_7$, and

---

5 Assume that citations are split across different representations of the same paper and the cleaning algorithm sums them up to get the correct count.
$p_2 \oplus p_3$ clearly contains duplicates since $p_1$ and $p_7$ are returned as separate clusters although they represent $C_1$. This answer semantics is indeed consistent with the default SQL semantics of **SELECT** queries which return a *bag* instead of a *set* to prevent the expensive duplicate elimination process. Similarly, in standard search (e.g., web, video, people search, etc.) end-applications and users are already accustomed to tolerating duplicates in the answer. In this case, we say that **QDA** follows *representative* semantics.

Note that in all three semantics, the returned answer does not omit any cluster that is returned by the original ER algorithm. It also does not return any extra cluster that is not returned by the original merge algorithm. That is, the returned answer is always equivalent to an answer generated by the original TC algorithm. For instance, an answer such as $p_1 \oplus p_7$ will not be acceptable since $C_2$ is not represented in the answer. Also, the answer $p_1 \oplus p_7$, $p_2 \oplus p_3$, and $p_5$ is not acceptable since $p_5$, which represents $C_3$, is returned while in fact $C_3$ does not satisfy the query.

As discussed above, the *exact* semantics is the most restrictive, followed by the *distinct* semantics, then the *representative* semantics. Thus, and as shown next, the *representative* semantics provides the most opportunities for savings, followed by the *distinct* semantics, then the *exact* semantics.

**QDA in Action.** We next illustrate how **QDA** exploits query semantics to reduce the number of resolves. Assume *representative* semantics. Before any data cleaning step is invoked, we observe that both $p_1$ and $p_7$ have citation counts of 65 and 45, both of which satisfy the citation count criteria (i.e., $\geq 45$), and thus should be present in the answer. As a result, depending on the order in which resolves are invoked, from 6 (viz., $\mathcal{R}(p_1,p_7)$ and $\mathcal{R}(p_1 \oplus p_7,p_j)$) to 11 (viz., $\mathcal{R}(p_1,p_j)$, $\mathcal{R}(p_7,p_j)$, and $\mathcal{R}(p_1,p_7)$), where $j = 2, 3, 4, 5, 6$, calls to resolve can be eliminated. They are *vestigial*, as cluster $C_1$ will be represented in the answer set by both $p_1$ and $p_7$, regardless of the outcome of these calls. Furthermore, such resolves
do not influence whether any additional clusters satisfy the query.

Next, suppose that \textbf{QDA} calls the following two resolves:

1. \( R(p_4, p_5) = \text{false} \),
2. \( R(p_2, p_3) = \text{true} \).

These two calls results in merging \( p_2 \oplus p_3 \) with citation count 45. Now, the query result would return answers \( p_1, p_7 \) and \( p_2 \oplus p_3 \) that represent clusters \( C_1 \) and \( C_2 \) respectively. Note that at this stage of the execution, all the remaining resolve function calls can be eliminated. The reason is that the remaining unresolved papers \( p_4, p_5, \) and \( p_6 \), whose citation counts are 15, 10, and 5, respectively, even if merged together, cannot form a new cluster \( C_3 \) distinct from \( C_1 \) and \( C_2 \), that satisfies the query predicate (since \( 15 + 10 + 5 = 30 < 45 \)). Thus, after only two calls to the resolve function, \textbf{QDA} can safely and confidently return the answer \( p_1, p_7, \) and \( p_2 \oplus p_3 \), as all clusters matching the query have been found. Note that while the clusters returned by \textbf{QDA} and the original algorithm are exactly the same (viz., \( C_1 \) and \( C_2 \)), their representations, \( \{ p_1 \oplus p_7, p_2 \oplus p_3 \oplus p_4 \} \) versus \( \{ p_1, p_7, p_2 \oplus p_3 \} \), are not the same. Also, note that the answer returned by \textbf{QDA} contains duplicates \( (p_1 \text{ and } p_7) \), while the answer returned by TC does not since \( p_1 \) and \( p_7 \) are merged into cluster \( C_1 \).

To generate a distinct answer we can try a strategy to isolate (i.e., disconnect) \( p_1 \) and \( p_7 \) from the remaining papers. Such a strategy would result in the following calls:

3. \( R(p_1, p_7) = \text{true} \),
4. \( R(p_1 \oplus p_7, p_2 \oplus p_3) = \text{false} \),
5. \( R(p_1 \oplus p_7, p_4) = \text{false} \),
6. \( R(p_1 \oplus p_7, p_5) = \text{false} \),
7. \( R(p_1 \oplus p_7, p_6) = \text{false} \).
At this stage, QDA can guarantee that \( p_1 \oplus p_7 \) and \( p_2 \oplus p_3 \) are distinct clusters in the answer of the original TC.

Now, to get the exact answer we need to add calls:

8. \( R(p_2 \oplus p_3, p_4) = \text{true} \),

9. \( R(p_2 \oplus p_3 \oplus p_4, p_5) = \text{false} \),

10. \( R(p_2 \oplus p_3 \oplus p_4, p_6) = \text{false} \).

Note that the original cleaning algorithm required 16 resolves, while QDA with representative semantics needed 2 resolves, distinct semantics required 7 resolves, and exact semantics needed 10 resolves, leading to savings in all three cases. In our experiments (Section 3.6), we will show that such a query-driven solution for all three semantics is significantly better compared to cleaning the whole dataset, especially when the query predicate is very selective.

### 3.3 Notation and Problem Definition

We start this section by introducing common ER notation in Section 3.3.1. Then, we discuss new QDA-specific notation and formally define the problem in Section 3.3.2.

#### 3.3.1 Standard Notation

**Relation and Clustering.** Let \( R = \{r_1, r_2, \ldots, r_{|R|}\} \) be a relation in the database, where \( r_k \) represents the \( k^{th} \) tuple of \( R \) and \( |R| \) is its cardinality. Relation \( R \) is considered dirty if at least two of its records \( r_i \) and \( r_j \) represent the same real-world entity, and hence \( r_i \) and \( r_j \) are duplicates. The attributes in \( R \) can be represented as \( \langle a_1, a_2, \ldots, a_n \rangle \), where \( n \) is the
Figure 3.1: Graph $G$ for $R$ (Table 3.1) Without $p_7$

arity of $R$. Thus, the $k^{th}$ record in $R$ is defined as $r_k = (\nu_{k1}, \nu_{k2}, \ldots, \nu_{kn})$, where $\nu_{k\ell}$ is the value of the $\ell^{th}$ attribute in the $k^{th}$ record (such that $1 \leq k \leq |R|$ and $1 \leq \ell \leq n$).

As mentioned in Chapter 2, the goal of traditional ER is to partition records in $R$ into a set of non-overlapping clusters $C = \{C_1, \ldots, C_{|C|}\}$ such that each cluster corresponds to a single real-world entity. That is, any two records $r_i$ and $r_j$ from the same cluster should co-refer, and simultaneously, any two records $r_k$ and $r_\ell$ from two distinct clusters $C_m$ and $C_n$ should not co-refer.

**Graphical View of the Problem.** The clustering problem can be represented graphically, as in [25, 52], where records in $R$ are encoded as a labeled graph $G = (V, E)$, where $V$ is a set of nodes interconnected by a set of edges $E$. Each record $r_i \in R$ is represented by a node $v_i \in V$, hence $|V| = |R|$. Each edge $e_{ij} = (v_i, v_j)$ represents the possibility that $r_i$ and $r_j$ may be duplicates. In the simplest case, $G$ is a complete graph with $|E| = \frac{|R|(|R|-1)}{2}$ edges. However, as we will explain in Section 3.4.3, our QDA approach will create a much simplified version of this graph. Figure 3.1 shows the encoding for $R$ from Table 3.1 without $p_7$ for clarity. The numbers outside the nodes represent the cited count for each paper.

**Resolve Function.** A pairwise resolve function $\mathcal{R}(r_i, r_j)$ operates on any two records $r_i, r_j \in R$ to try to decide whether they co-refer, that is, refer to the same real-world entity
or not. Resolve is a “black-box” function that may be cheap or very expensive – e.g., a web query. The algorithms we develop, in this chapter, are meant for the cases where the resolve function is not very cheap and calling resolves is in fact the bottleneck of an ER approach. The resolve function may return a classification, a binary answer, or a numeric similarity value (viz., confidence). For the purpose of embedding resolve within an ER algorithm, the outcome of the resolve function is mapped into the following three decisions:

1. \( R(r_i, r_j) = \text{MustMerge} \), if resolve is highly confident \( r_i \) and \( r_j \) are the same and hence, must be merged,

2. \( R(r_i, r_j) = \text{MustSeparate} \), if resolve is highly confident \( r_i \) and \( r_j \) are different and hence, must be separated,

3. \( R(r_i, r_j) = \text{Uncertain} \), otherwise.

By controlling when (i.e., for which similarity/dissimilarity levels) a resolve maps its outcome to each of these three decisions, the degree of eagerness can be controlled\(^6\). Naturally, the resolve may output decisions that are incorrect and that could lead to errors in the entity resolution process.

**Merge and Combine Functions.** If \( R(r_i, r_j) \) returns \text{MustMerge}, then the two records are declared to be duplicates and a merge function \( r_i \oplus r_j \) will consolidate them to produce a new record \( r_m = r_i \oplus r_j \). To merge two duplicate records \( r_i \) and \( r_j \), a combine function is used for each attribute \( a_\ell \) such that \( 1 \leq \ell \leq n \).

We assume that the WHERE-attribute combine function \( \nu_{i\ell} \oplus \nu_{j\ell} \) takes two values of attribute \( a_\ell \) and outputs a single value \( \nu_{m\ell} = \nu_{i\ell} \oplus \nu_{j\ell} \). Such combine functions perform different operations depending on the type of \( a_\ell \).

\(^6\)For example, conceptually, R-Swoosh [21] merges \text{MustMerge} entities eagerly, but does not actually use \text{MustSeparate} – only \text{Uncertain}.  

24
If $a_{\ell}$ is a numeric attribute then we consider:

- **ADD** semantics: $\nu_{i_{\ell}} \oplus \nu_{j_{\ell}} = \nu_{i_{\ell}} + \nu_{j_{\ell}}$,
- **MAX** semantics: $\nu_{i_{\ell}} \oplus \nu_{j_{\ell}} = \max(\nu_{i_{\ell}}, \nu_{j_{\ell}})$,
- **MIN** semantics: $\nu_{i_{\ell}} \oplus \nu_{j_{\ell}} = \min(\nu_{i_{\ell}}, \nu_{j_{\ell}})$.

The **ADD** semantics are used when records are obtained from the same data source, yet their entities are split in parts. The number of citations in duplicate publications in Google Scholar is an example of such a case. In this chapter, **ADD** semantics are used as the default semantics to illustrate various examples, unless stated otherwise.

The **MAX** semantics are used, for instance, when records are retrieved from different data sources where some copies of the record are obsolete. It is typically applied to attributes that monotonically increase over time, such as *age*. The **MIN** semantics are similar to **MAX**, except that they are applied to attributes that monotonically decrease over time, such as *days to expire* attribute in a product table.

If $a_{\ell}$ is a categorical attribute then we consider:

- **EXEMPLAR** semantics: $\nu_{i_{\ell}} \oplus \nu_{j_{\ell}}$ chooses either $\nu_{i_{\ell}}$ or $\nu_{j_{\ell}}$ according to some policy,
- **UNION** semantics: $\nu_{i_{\ell}} \oplus \nu_{j_{\ell}} = \nu_{i_{\ell}} \cup \nu_{j_{\ell}}$.

The **EXEMPLAR** semantics are used when, for example, one value holds more information than the other value. For instance, in the *authors* attribute, the value “Alon Halevy” dominates “A. Halevy”. In contrast, the **UNION** semantics are utilized when the system needs to retain all possible values for some attribute, e.g., the application needs to retain all *email* addresses for an author.

Note that the aforementioned combine functions have the commutativity and associativity
properties defined as:

1. Commutativity: \( \nu_{i\ell} \oplus \nu_{j\ell} = \nu_{j\ell} \oplus \nu_{i\ell} \).

2. Associativity: \((\nu_{i\ell} \oplus \nu_{j\ell}) \oplus \nu_{k\ell} = \nu_{i\ell} \oplus (\nu_{j\ell} \oplus \nu_{k\ell})\).

Since these properties hold regardless of the merge order, the representation of the merged cluster will always be the same.

### 3.3.2 Approach-Specific Notation

**Queries.** We will consider SQL selection queries. For clarity of presentation, our discussion will focus on queries with a single predicate \( p \), with the syntax:

\[
\text{SELECT} \ [\text{DISTINCT}] \ast \text{ FROM } R \text{ WHERE } a_{\ell} \text{ op } t
\]

\[
op \text{ is } \begin{cases} 
<, \leq, >, \geq, \text{ or } = & \text{ if } a_{\ell} \text{ is a numeric attribute;} \\
= & \text{ if } a_{\ell} \text{ is a categorical attribute.}
\end{cases}
\]

We will discuss the multi-predicate case in Section 3.4.

**Returned Answer Equivalence.** Before we formally define various answer semantics, we need to introduce several auxiliary concepts. Recall that a cluster \( C \) can be viewed as a set of records \( C = \{r_1, r_2, \ldots, r_{|C|}\} \). We will say:

**Definition 1.** Record \( r_k \) represents cluster \( C \), if \( r_k \in C \), or \( r_k = r_i \oplus r_j \) where \( r_i \) and \( r_j \) represent \( C \).

Assume an entity resolution algorithm \( \mathcal{A} \), such as TC, is applied to \( R \) and generates a clustering \( \mathcal{C}_{\mathcal{A}} \) as its answer. That is, \( \mathcal{C}_{\mathcal{A}} \) is the set of clusters that partitions \( R \). Let \( \mathcal{C}_{\mathcal{A},Q} \)
denotes the set of clusters from $C_A$ that satisfy query $Q$. Let $C_{QDA,Q}$ be the set of clusters returned by QDA as the answer to $Q$.

To make our definitions formal, we also must account for the following observation: in general, the same algorithm $A$ might produce a different clustering $C'_A$ of $R$, where $C'_A \neq C_A$, if $A$ changes the order in which it invokes resolves [76]. Let $\bar{C}_A$ denotes the set of all possible output clusterings that $A$ may produce as a result of changing its resolves order.

Now we can define when $C_{QDA,Q}$ is exactly, distinctly, or representationally equivalent to an answer of $A$ to query $Q$:

**Definition 2.** Answer $C_{QDA,Q}$ generated by QDA for query $Q$ is exactly equivalent to that of algorithm $A$ iff there exists $C_A \in \bar{C}_A$ such that: (i) for each cluster $C_i \in C_{A,Q}$ there exists exactly one cluster $C_j \in C_{QDA,Q}$ such that $C_i \equiv C_j$, and (ii) for each cluster $C_j \in C_{QDA,Q}$ there exists exactly one cluster $C_i \in C_{A,Q}$ such that $C_j \equiv C_i$.

In other words, there is a one-to-one mapping between clusters in $C_{QDA,Q}$ and $C_{A,Q}$ and the content of clusters in $C_{QDA,Q}$ is identical to those in $C_{A,Q}$.

We further define the less restrictive distinct semantics as:

**Definition 3.** Answer $C_{QDA,Q}$ generated by QDA for query $Q$ is distinctly equivalent to that of algorithm $A$ iff there exists $C_A \in \bar{C}_A$ such that: (i) for each cluster $C_i \in C_{A,Q}$ there exists exactly one cluster $C_j \in C_{QDA,Q}$ such that $C_i \supseteq C_j$, and (ii) for each cluster $C_j \in C_{QDA,Q}$ there exists exactly one cluster $C_i \in C_{A,Q}$ such that $C_j \subseteq C_i$.

That is, there is still a one-to-one mapping between clusters in $C_{QDA,Q}$ and $C_{A,Q}$, but now clusters in $C_{QDA,Q}$ are allowed to be subsets of clusters from $C_{A,Q}$.

We define the least restrictive representative semantics as:

**Definition 4.** Answer $C_{QDA,Q}$ generated by QDA for query $Q$ is representationally equivalent
to that of algorithm $A$ iff there exists $C_A \in \bar{C}_A$ such that: (i) for each cluster $C_i \in C_{A,Q}$ there exists at least one cluster $C_j \in C_{QDA,Q}$ such that $C_i \supseteq C_j$, and (ii) for each cluster $C_j \in C_{QDA,Q}$ there exists exactly one cluster $C_i \in C_{A,Q}$ such that $C_j \subseteq C_i$.

The representative semantics go one step further on top of the distinct semantics and do not require the one-to-one mapping by allowing for duplicates. Namely, they ask for one-to-many mapping from $C_{A,Q}$ to $C_{QDA,Q}$ and one-to-one mapping from $C_{QDA,Q}$ to $C_{A,Q}$.

Problem Definition. Let $A$ be an original entity resolution algorithm, e.g., TC, whose query-driven version is being developed. Then, given a query $Q$, we can formally define our problem as an optimization problem as follows:

Minimize: Number of $\text{R}(\cdot)$
Subject to:
1. $\forall C \in C_{QDA,Q}, C$ satisfies $Q$; // Query satisfaction
2. $C_{QDA,Q} \equiv C_{A,Q}$; // User-defined equivalence

It can be trivially shown that achieving an optimal solution that generates the least number of resolves is infeasible in practice, as it requires an “oracle” that knows which pair of nodes to resolve next. To illustrate this, consider the symmetric graph shown in Figure 3.2. Its (unknown to the algorithm) ground truth is that the three nodes $v_1, v_2, \text{ and } v_3$ correspond to two clusters: $C_1 = \{v_1, v_2\}$ and $C_2 = \{v_3\}$. The numbers outside the nodes represent the values of the nodes.

Suppose that we are interested in nodes with values greater than or equal 30 (viz., $p : val \geq 30$). The algorithm must call resolve at least once, since $v_1 \oplus v_2, v_1 \oplus v_3, v_2 \oplus v_3, \text{ or } v_1 \oplus v_2 \oplus v_3$ can potentially co-refer and thus form a new cluster that would satisfy $val \geq 30$. The algorithm can start by resolving any of the three edges. The optimal solution will call resolve only once: on edge $e_{12}$, which will return MustMerge. From that it will learn that $v_1$
and $v_2$ co-refer, and it can stop and output its final answer as $v_1 \oplus v_2$. In contrast, calling resolve first on $e_{13}$ or $e_{23}$, instead of $e_{12}$, would result in resolve returning MustSeperate. This will force the algorithm to call resolve at least one more time to check whether the remaining nodes can form a cluster that satisfy $p: \text{val} \geq 30$.

However, observe that the graph is symmetric and since the ground truth is unknown, the algorithm has no way of knowing that choosing edge $e_{12}$ first would result in the minimal number of calls to resolve for this case. Hence, without an oracle an optimal solution is not possible, and the goal translates into finding a good solution by developing algorithms which attempt to reduce the number of calls to the resolve function by exploiting vestigiality, and by implementing a good edge selection policy, as explained in Section 3.5.

### 3.4 Vestigiality

In this section, we introduce the notion of *vestigiality*, which is the key concept in our query-driven solution. Before we can formally define it, we have to introduce several auxiliary concepts. We first define a way to categorize a triple $(p, \oplus, a_\ell)$ (where $p$ is the query predicate, $\oplus$ is the combine function defined over $a_\ell$’s domain) into three categories: *in-preserving*, *out-preserving*, and *neither* as explained in Section 3.4.1. Then, we discuss how to deal with multi-predicate selection queries in Section 3.4.2. The construction of the labeled
graph is explained in Section 3.4.3. In Section 3.4.4, we discuss the difference between vestigiality and minimality. Finally, we explain how this triple categorization as well as the new notions of relevant clique and minimal clique can be used to test for vestigiality of an edge in Section 3.4.5.

### 3.4.1 Triple \((p, \oplus, a_\ell)\) Categorization

QDA exploits the specificity of a query predicate \(p\) and the semantics of a combine function \(\oplus\) defined on attribute \(a_\ell\) to significantly reduce the cleaning overhead by resolving only those edges that may influence the answer of \(Q\). For that goal, we will classify any triple \((p, \oplus, a_\ell)\) into three generic categories: in-preserving, out-preserving, and neither. These broad categories are important as they allow us to develop generic QDA algorithms instead of developing specific algorithms for each small case.

**Definition 5.** Triple \((p, \oplus, a_\ell)\) is in-preserving, if for all possible values \(\nu_{i\ell}, \nu_{j\ell} \in a_\ell\), if \(p\) is true for \(\nu_{i\ell}\), then \(p\) is also true for all \(\nu_{i\ell} \oplus \nu_{j\ell}\).

This property means that once a record is in the answer, it will remain so, even if it is merged with other records. For instance, triple \((\text{cited} \geq 45, \text{ADD}, \text{cited})\) is in-preserving, since any tuple with a citation count above or equal to 45 will continue to be above or equal to 45 even if merged with other tuples. In contrast, triple \((\text{cited} \leq 45, \text{ADD}, \text{cited})\) is not in-preserving.

**Definition 6.** Triple \((p, \oplus, a_\ell)\) is out-preserving, if for all possible values \(\nu_{i\ell}, \nu_{j\ell} \in a_\ell\), if \(p\) is false for \(\nu_{i\ell}\), then it is also false for all \(\nu_{i\ell} \oplus \nu_{j\ell}\).

This property means that once a record is out of the answer, it will remain so, even if it is merged with other records. For example, triple \((\text{cited} \leq 45, \text{ADD}, \text{cited})\) is out-preserving.
Tables 3.3 and 3.4 show a classification of different common triples for numeric and categorical attributes, respectively.

### 3.4.2 Multi-Predicate Selection Queries

Our discussion so far has focused on the case where the WHERE-clause contains a single predicate. The overall solution, however, applies to more complex selection queries with multiple predicates connected via logical connectives, such as \textit{AND}, \textit{OR}, and \textit{NOT}. This is since such combinations of triples can also be categorized into the same three categories – based on the categories of the basic triples it is composed of, as illustrated in Table 3.5. For instance, consider the following range query:

\textit{Query 2. SELECT * FROM R WHERE cited \geq 45 AND cited \leq 65}

This range query consists of two basic predicates $p_1 : \textit{cited} \geq 45$ and $p_2 : \textit{cited} \leq 65$. Hence, it consists of two basic triples: an in-preserving triple $\tau_1 = (\textit{cited} \geq 45, \textit{ADD}, \textit{cited})$ and an out-preserving triple $\tau_2 = (\textit{cited} \leq 65, \textit{ADD}, \textit{cited})$. From Table 3.5, we can see that the resulting combination $\tau_1 \land \tau_2$ is neither in- nor out-preserving. To see why, consider record $r_i$ with

![Table 3.3: Triple Categorization for Numerical Attributes](image)

<table>
<thead>
<tr>
<th>⊕, domain</th>
<th>$a_\ell \geq t$ or, $a_\ell &gt; t$</th>
<th>$a_\ell \leq t$ or, $a_\ell &lt; t$</th>
<th>$t_1 \leq a_\ell \leq t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ADD}, $a_\ell \in \mathbb{R}^+$</td>
<td>in-preserving</td>
<td>out-preserving</td>
<td>neither</td>
</tr>
<tr>
<td>\textit{MAX}, $a_\ell \in \mathbb{R}$</td>
<td>in-preserving</td>
<td>out-preserving</td>
<td>neither</td>
</tr>
<tr>
<td>\textit{MIN}, $a_\ell \in \mathbb{R}$</td>
<td>out-preserving</td>
<td>in-preserving</td>
<td>neither</td>
</tr>
</tbody>
</table>

Table 3.4: Triple Categorization for Categorical Attributes

<table>
<thead>
<tr>
<th>⊕, domain</th>
<th>$a_\ell = t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{EXEMPLAR}, $a_\ell \in \textit{enum}$</td>
<td>in-preserving</td>
</tr>
<tr>
<td>\textit{UNION}, $a_\ell \in \textit{enum}$</td>
<td>in-preserving</td>
</tr>
</tbody>
</table>
\[ \tau_i = (p_i, \oplus_i, a_i) \quad \tau_j = (p_j, \oplus_j, a_j) \quad \tau_i \land \tau_j \quad \tau_i \lor \tau_j \quad \neg \tau_i \]

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
in-preserving & in-preserving & in-preserving & in-preserving & out-preserving \\
\hline
in-preserving & out-preserving & neither & neither & out-preserving \\
\hline
out-preserving & in-preserving & neither & neither & in-preserving \\
\hline
out-preserving & out-preserving & out-preserving & out-preserving & in-preserving \\
\hline
in-preserving & neither & neither & neither & out-preserving \\
\hline
neither & in-preserving & neither & neither & neither \\
\hline
out-preserving & neither & neither & neither & in-preserving \\
\hline
neither & out-preserving & neither & neither & neither \\
\hline
neither & neither & neither & neither & neither \\
\hline
\end{tabular}
\end{center}

Table 3.5: Triples Generalization

cited = 40. Initially \( r_i \) is out of the answer of Query 2. If \( r_i \) merges with another record \( r_j \) with \( cited = 10 \), the new record \( r_i \oplus r_j \) will have \( cited = 40 + 10 = 50 \) and will be in the answer. If now \( r_i \oplus r_j \) can merge again with another record \( r_k \) with \( cited = 20 \), then \( r_i \oplus r_j \oplus r_k \) will have \( cited = 50 + 20 = 70 \) which clearly does not satisfy Query 2. Hence, \( \tau_1 \land \tau_2 \) is neither in-preserving nor out-preserving.

We can prove a lemma for each combination (in Table 3.5) and following are the most important cases which extend the concepts of in and out-preserving to more general predicates:

**Lemma 1.** If triple \( \tau_i = (p_i, \oplus_i, a_i) \) is in-preserving and triple \( \tau_j = (p_j, \oplus_j, a_j) \) is in-preserving then, triple \( \tau \) that results from the combination \( \tau_i \land \tau_j \) is also in-preserving.

**Proof.** By definition, to show that triple \( \tau \) is in-preserving we need to show that, \( p : p_i \land p_j \) is true if and only if \( p_i \) is true and \( p_j \) is true. If \( p_i \) is true for \( \nu_{ki} \), then \( p_i \) is true for all possible values \( \nu_{ki} \oplus_i \nu_{li} \) since \( \tau_i \) is in-preserving. Likewise, if \( p_j \) is true for \( \nu_{kj} \), then \( p_j \) is true for all possible values \( \nu_{kj} \oplus_j \nu_{lj} \) since \( \tau_j \) is in-preserving. As a result, triple \( \tau \) is in-preserving when triples \( \tau_i \) and \( \tau_j \) are in-preserving. \qed
Lemma 2. If triple $\tau_i = (p_i, \oplus_i, a_i)$ is out-preserving and triple $\tau_j = (p_j, \oplus_j, a_j)$ is out-preserving, then triple $\tau$ that results from the combination $\tau_i \land \tau_j$ is also out-preserving.

Proof. By definition, to show that triple $\tau$ is out-preserving we need to show that, $p : p_i \land p_j$ is false if and only if (i) $p_i$ is false and $p_j$ is true, (ii) $p_i$ is true and $p_j$ is false, or (iii) $p_i$ is false and $p_j$ is false. If $p_i$ is false for $\nu_{ki}$, then $p_i$ is false for all possible values $\nu_{ki} \oplus_i \nu_{li}$ since $\tau_i$ is out-preserving. Similarly, if $p_j$ is false for $\nu_{kj}$, then it does not follow that $p_j$ is true for all possible values $\nu_{kj} \oplus_j \nu_{lj}$ since $\tau_j$ is out-preserving. Therefore, triple $\tau$ is out-preserving when triples $\tau_i$ and $\tau_j$ are out-preserving.

Lemma 3. If triple $\tau_i = (p_i, \oplus_i, a_i)$ is in-preserving and triple $\tau_j = (p_j, \oplus_j, a_j)$ is out-preserving, then triple $\tau$ that results from the combination $\tau_i \land \tau_j$ is is neither in- nor out-preserving.

Proof. By definition, to show that triple $\tau$ is not in-preserving we need to show that, $p : p_i \land p_j$ is true if and only if $p_i$ is true and $p_j$ is true. If $p_i$ is true for $\nu_{ki}$, then $p_i$ is true for all possible values $\nu_{ki} \oplus_i \nu_{li}$ since $\tau_i$ is in-preserving. However, if $p_j$ is true for $\nu_{kj}$, then it does not follow that $p_j$ is true for all possible values $\nu_{kj} \oplus_j \nu_{lj}$ since $\tau_j$ is not in-preserving. Hence, $\tau$ is not in-preserving.

Similarly, by definition, to show that triple $\tau$ is not out-preserving we need to show that, $p : p_i \land p_j$ is false if and only if (i) $p_i$ is false and $p_j$ is true, (ii) $p_i$ is true and $p_j$ is false, or (iii) $p_i$ is false and $p_j$ is false. If $p_j$ is false for $\nu_{kj}$, then it does not follow that $p_j$ is true for all possible values $\nu_{kj} \oplus_j \nu_{lj}$ since $\tau_j$ is out-preserving. Nevertheless, if $p_i$ is false for $\nu_{ki}$, then it does not follow that $p_i$ is false for all possible values $\nu_{ki} \oplus_i \nu_{li}$ since $\tau_i$ is not out-preserving. Consequently, $\tau$ is not out-preserving.
As a result, triple $\tau$ is neither in- nor out-preserving when triple $\tau_i$ is in-preserving and triple $\tau_j$ is out-preserving.

**Lemma 4.** If triple $\tau_i = (p_i, \oplus_i, a_i)$ is in-preserving and triple $\tau_j = (p_j, \oplus_j, a_j)$ is neither in- nor out-preserving, then triple $\tau$ that results from the combination $\tau_i \wedge \tau_j$ is neither in- nor out-preserving.

**Proof.** By definition, to show that triple $\tau$ is neither in- nor out-preserving we need to show that, $p : p_i \wedge p_j$ is true if and only if $p_i$ is true and $p_j$ is true. If $p_j$ is true for $\nu_{kj}$, then it does not follow that $p_j$ is true for all possible values $\nu_{kj} \oplus_j \nu_{kj}$ since $\tau_j$ is not in-preserving. Also, if $p_j$ is false for $\nu_{kj}$, then it does not follow that $p_j$ is false for all possible values $\nu_{kj} \oplus_j \nu_{kj}$ since $\tau_j$ is not out-preserving. Hence, triple $\tau$ is neither in- nor out-preserving when triple $\tau_j$ is neither in- nor out-preserving.

**Lemma 5.** If triple $\tau_i = (p_i, \oplus_i, a_i)$ is in-preserving and triple $\tau_j = (p_j, \oplus_j, a_j)$ is in-preserving then, triple $\tau$ that results from the combination $\tau_i \lor \tau_j$ is also in-preserving.

**Proof.** By definition, to show that triple $\tau$ is in-preserving we need to show that, $p : p_i \lor p_j$ is true if and only if (i) $p_i$ is true and $p_j$ is false, (ii) $p_i$ is false and $p_j$ is true, or (iii) $p_i$ is true and $p_j$ is true. That is, it is sufficient to make sure that either $p_i$ is true or $p_j$ is true. Herein, if $p_i$ is true for $\nu_{ki}$, then $p_i$ is true for all possible values $\nu_{ki} \oplus_i \nu_{ki}$ since $\tau_i$ is in-preserving. Equivalently, if $p_j$ is true for $\nu_{kj}$, then $p_j$ is true for all possible values $\nu_{kj} \oplus_j \nu_{kj}$ since $\tau_j$ is in-preserving. Hence, triple $\tau$ is in-preserving when triples $\tau_i$ and $\tau_j$ are in-preserving.

**Lemma 6.** If triple $\tau_i = (p_i, \oplus_i, a_i)$ is in-preserving, then triple $\tau$ that results from the negation $\neg \tau_i$ is out-preserving.

**Proof.** By definition, to show that triple $\tau$ is out-preserving we need to show that, $p_i$ is true if and only if $\neg p_i$ is false. If $p_i$ is true for $\nu_{ki}$ then $p_i$ is true for all possible values $\nu_{ki} \oplus_i \nu_{ki}$.
since $\tau_i$ is in-preserving. Also, if $\neg p_i$ is false for $\nu_{ki}$, then $\neg p_i$ is false for all possible values $\nu_{ki} \oplus_i \nu_{li}$ since $\tau_i$ is in-preserving. Hence, triple $\tau$ is out-preserving when triple $\tau_i$ is in-preserving. $\square$

### 3.4.3 Creating and Labeling the Graph

To formally define vestigiality testing, we need to explain how QDA builds and labels the graph, see CREATE\-GRAPH(.) function in Figure 3.3. The main goal of this function is to avoid creating as many nodes and edges as possible in order to improve the efficiency. As common for ER techniques, the function starts by applying blocking (See Chapter 2) and will not create edges for pairs that cannot be duplicates according to blocking. More importantly, on top of blocking, the function will also remove from consideration nodes and edges that will not influence further processing of $Q$, thus improving the efficiency on top of blocking from the very beginning.

The algorithm starts by iterating over each tuple $r_k \in R$ to create the corresponding node $v_k$. It sets label $\ell[v_k]$ of $v_k$ (Lines 1–8) as:

1. $\ell[v_k] = \text{in}$ when triple $(p, \oplus, a_{\ell})$ is in-preserving and $v_k$ satisfies $Q$. Node $v_k$ is added to $A_{\text{cur}}$ as it is guaranteed to be in the final answer.

2. $\ell[v_k] = \text{out}$ when triple $(p, \oplus, a_{\ell})$ is out-preserving and $v_k$ does not satisfy $Q$. Node $v_k$ is added to $V_{\text{out}}$.

3. $\ell[v_k] = \text{maybe}$, otherwise. Node $v_k$ is added to $V_{\text{maybe}}$.

The following is an example that explains such labeling:

**Example 1.** Consider relation $R$ in Figure 3.1 and Query 1 ($p: \text{cited} \geq 45$). Node 1 is labeled in because it is guaranteed to appear in the result-set. Note that no other node,
CREATE-GRAPH($R, Q, A_{cur}, V_{out}, V_{maybe}, \oplus$)
1. for each $r_k \in R$ do
2. $v_k \leftarrow$ CREATE-NODE($r_k$)
3. if IS-IN-PRESERVING($p, \oplus, a_\ell$) and SATISFY-QUERY($v_k, Q$) then
4. $A_{cur} \leftarrow A_{cur} \cup \{v_k\}$
5. else if IS-OUT-PRESERVING($p, \oplus, a_\ell$) and not SATISFY-QUERY($v_k, Q$) then
6. $V_{out} \leftarrow V_{out} \cup \{v_k\}$
7. else
8. $V_{maybe} \leftarrow V_{maybe} \cup \{v_k\}$
9. $V \leftarrow \{A_{cur}, V_{out}, V_{maybe}\}$
10. $E$ $\leftarrow$ CREATE-EDGES-WITH-BLOCKING($V_{maybe}, V_{maybe}$)
11. $E$ $\leftarrow$ $E$ $\cup$ CREATE-EDGES-WITH-BLOCKING($V_{out}, V_{maybe}$)
12. return $G(V, E)$

Figure 3.3: CREATE-GRAPH(.) Function

if combined with node 1, can change its labeling because ($p$, ADD, cited) is in-preserving. Nodes 2, 3, ..., 6 are labeled maybe since their cited attribute value might increase to above or equal to 45 as a result of them being consolidated.

The algorithm then creates edges, but only if they can exist according to blocking and (i) only among nodes in $V_{maybe}$ and (ii) for each $v_i, v_j$ pair where $v_i \in V_{out}$ and $v_j \in V_{maybe}$ (Lines 10–11). This is because nodes in $V_{maybe}$ that merge with $V_{out}$ nodes cannot be in the answer. For each edge $e_{ij} \in E$, QDA sets labels as:

1. $\ell[e_{ij}] = \text{yes}$, when $\mathcal{R}(r_i, r_j)$ has already been called and returned MustMerge,
2. $\ell[e_{ij}] = \text{no}$, when $\mathcal{R}(r_i, r_j)$ has already been called and returned MustSeparate,
3. $\ell[e_{ij}] = \text{maybe}$, when $\mathcal{R}(r_i, r_j)$ has already been called and returned Uncertain,
4. $\ell[e_{ij}] = \text{vestigial}$, when, Definition 8 holds. Note that as QDA proceeds forward, some edges that were not vestigial previously may become vestigial. But once they become vestigial, they remain so,
5. $\ell[e_{ij}] = \text{unresolved}$, otherwise.

It should be noted that edge labeling is a convenient semantic notation useful for explaining
Figure 3.4: Example of Edge Labeling in Graph $G$. Unlabeled Edges Are **unresolved** Edges

various concepts. For efficiency, however, the algorithm does not utilize **yes** and **no** labels in its actual processing. For example, instead of labeling edge $e_{ij}$ as a **no** edge, it simply removes this edge since this simplifies the graph. Similarly, instead of labeling an edge $e_{ij} = (v_i, v_j)$ as a **yes** edge, the algorithm merges nodes $v_i$ and $v_j$ into a new node $v_m = v_i \oplus v_j$. Note that merging two nodes $v_m = v_i \oplus v_j$, is made to be semantically consistent with other **no** edges.

We will say that the current labeling of the graph determines the current state of the resolution process. Now we can define the concept of the current answer $A_{cur}$.

**Definition 7.** Based on the given edge labeling, the current answer $A_{cur}$ to $Q$ is the answer resulting from assuming that all **vestigial** and **unresolved** edges are **no** edges.

**Example 2.** Consider relation $R$ in Figure 3.1 and an in-preserving triple e.g., $(cited \geq 45, ADD, cited)$. Initially only $p_1$ is labeled in, since it is guaranteed to be in the result-set ($A_{cur} = \{p_1\}$). Thus, all edges incident to $p_1$, that is $e_{12}, e_{13}, e_{14}, e_{15},$ and $e_{16}$, are **vestigial**, see Figure 3.4a. If the algorithm then calls $\mathcal{R}(p_2, p_3)$ and $\mathcal{R}(p_4, p_5)$, the corresponding edges will be assigned labels $\ell[e_{23}] = yes$ and $\ell[e_{45}] = no$, as illustrated in Figure 3.4b.
3.4.4 Vestigiality vs. Minimality

In this section, we present the difference between the concept of vestigiality and the concept of minimality. Even though, intuitively, avoiding resolving vestigial edges should lead to improved efficiency, it is possible that the minimal edge resolve sequence includes a vestigial edge, as demonstrated by the following example.

**Example 3.** Consider the graph in Figure 3.5a that corresponds to three clusters \( C_1 = \{v_1, v_2\} \), \( C_2 = \{v_3\} \), and \( C_3 = \{v_4\} \) and an in-preserving triple e.g., \((val \geq 30, \text{ADD}, val)\). Edge \( e_{12} \) is vestigial, since even if \( v_1 \) and \( v_2 \) are the same, the combined citation count is \( 5 + 20 = 25 \geq 30 \). The other two edges are not vestigial, since the combined citation count is 30. However, querying for the vestigial edge \( e_{12} \) would result in the graph shown in Figure 3.5b, where edges \( e_{23} \) and \( e_{24} \) are no edges since we know that \( v_1 \) is not equal to \( v_3 \) and \( v_4 \). Hence, by issuing resolve once, the algorithm can compute the final answer, which is the empty set in this case. It is easy to see that if the algorithm starts by resolving any other edge instead of \( e_{12} \), then one more resolve will be required.
3.4.5 Vestigiality Testing Using Cliques

Before introducing the new notions of relevant and minimal cliques which are used to test for vestigiality of an edge, let us first define the concept of a vestigial edge. Intuitively, an edge is vestigial if its resolution outcome does not influence the query result. Formally:

**Definition 8.** Let \( \mathcal{A} \) be an original entity resolution algorithm such as TC. An edge \( e_{ij} \in E \) is vestigial when, regardless of what the ground truth for \( e_{ij} \) might be, QDA can guarantee that by treating \( e_{ij} \) as a no edge, it can still compute an equivalent answer to that of algorithm \( \mathcal{A} \).

Now, we introduce some of the necessary concepts utilized in our solution. We use the standard definition of a clique in an undirected graph \( G = (V, E) \), which is a subset of the node set \( S \subseteq V \), such that for every two nodes in \( S \), there exists an edge in \( E \) that connects them. A clique is an important concept for entity resolution since it identifies which groups of nodes (viz., records) might or might not co-refer:

**Lemma 7.** Nodes (i.e., records) co-refer only if they form a clique consisting of only yes edges in the ground truth.

Consequently, if a group of nodes is not a clique (e.g., some edges are marked no (i.e., removed)), and the algorithm did not make a mistake in removing those edges, then that group corresponds to at least two distinct entities. Note that Lemma 7 deals with the ground truth labels and not the decisions returned by the resolve function.

Let \( C_{\text{cur}} \) be the set of clusters in the current answer \( A_{\text{cur}} \). Now, we can define the notions of a relevant clique and a minimal clique.

**Definition 9.** A clique \( S \) is called relevant to \( Q \), if we can assign labels to its edges such that this labeling might change \( C_{\text{cur}} \), by either adding (at least one) new cluster to \( C_{\text{cur}} \), or removing (at least one) cluster from \( C_{\text{cur}} \).
The concept of relevant cliques provides a mechanism to test if an edge is vestigial as stated in the next theorem.

**Theorem 1.** Given the current labeled graph \( G \), a selection query \( Q \) with predicate \( p \) on attribute \( a_\ell \), if no relevant clique exists that includes \( e_{ij} \), then \( e_{ij} \) is *vestigial*. However, the reverse does not hold: a vestigial edge could be part of a relevant clique.

**Proof.** Let us prove that if an edge \( e_{ij} \) is not vestigial, then it belongs to a relevant clique. By contradiction, if an edge does not belong to any relevant clique, then it cannot participate in changing \( C_{\text{cur}} \) by adding or removing one (or more) cluster from it, thus such an edge is vestigial.

The next example helps in illustrating the importance of considering relevant cliques.

**Example 4.** Consider \( G \) shown in Figure 3.1 after resolving \( e_{23} \) only and Query 1, \( p = (\text{cited} \geq 45) \). \( A_{\text{cur}} = \{p_1, p_2 \oplus p_3\} \) because both \( p_1 \) and \( p_2 \oplus p_3 \) have citation counts \( \geq 45 \) (65 and 45, respectively). As a result, all edges incident to \( p_1 \) and \( p_2 \oplus p_3 \) are vestigial. Note that nodes 4, 5, and 6 form a clique \( S \). The sum up of the \textit{cited} attribute for these nodes is \( 15 + 10 + 5 = 30 \not\geq 45 \). Thus, merging nodes in \( S \) cannot change \( A_{\text{cur}} \) (no new clusters can be added to it). Hence, \( S \) is not a relevant clique with respect to \( p \). Thus, edges \( e_{45}, e_{46}, \) and \( e_{56} \) are not part of any relevant cliques and hence, *vestigial*.

In fact, when triple \( (p, \oplus, a_\ell) \) is in-preserving, we can show that the edge must not only be part of a relevant clique, but a minimal clique as defined below:

**Definition 10.** A relevant clique \( S \) is called a *minimal clique*, if no subset of nodes in \( S \) can form a relevant clique.

**Theorem 2.** Given a graph \( G \) and an in-preserving triple \( (p, \oplus, a_\ell) \), an unresolved edge \( e_{ij} \) is *vestigial* if and only if no minimal clique exists that includes \( e_{ij} \).
Proof. Let us first prove that if edge $e_{ij}$ belongs to a minimal clique $S_{\text{min}}$, then it is not vestigial. Note that it is possible that all of the unresolved edges, except for those in $S_{\text{min}}$, are no edges. If that is the case, and if at least one of the edges in $S_{\text{min}}$ is a no edge in the ground truth, then the nodes in $S_{\text{min}}$ do not co-refer and $\mathcal{C}_{\text{cur}}$ is the correct final answer. Otherwise, if all of the edges in $S_{\text{min}}$ are yes edges in the ground truth, then the nodes in $S_{\text{min}}$ do co-refer and the correct final answer will be different from current answer $A_{\text{cur}}$, in which a new cluster $C_i$ will be added to the set of clusters $\mathcal{C}_{\text{cur}}$ since triple $(p, \emptyset, a_t)$ is in-preserving. Therefore, the algorithm cannot permanently ignore edge $e_{ij}$ by treating it as a no edge, since it might influence the correctness of the answer which the algorithm produces. Hence, this edge is not vestigial.

Let us now prove that if an edge $e_{ij}$ is not vestigial, then it belongs to a minimal clique. By contradiction, if an edge does not belong to any relevant clique, then it cannot participate in forming a new cluster that would change $\mathcal{C}_{\text{cur}}$ and thus it is vestigial. If, however, an edge does belong to a relevant clique $S$, but not a minimal clique, then two cases are possible. If this is a no edge in the ground truth, then the algorithm will compute the correct answer by declaring this edge as vestigial, because all vestigial edges are treated as no edges in the final answer computations. If this is a yes edge, then two more cases are possible. Let $S_{\text{min}}$ be a minimal clique that is part of $S$. Then, if the nodes in $S_{\text{min}}$ do not co-refer, then it is safe to declare $e_{ij}$ to be vestigial with respect to $S_{\text{min}}$, since $S_{\text{min}}$ does not form a new cluster that must be accounted for in forming the final answer. If, however, the nodes in $S_{\text{min}}$ do co-refer, then let $C_i$ be the new cluster they represent. But the algorithm does not need to know the value of $\mathcal{R}(e_{ij})$ to determine how to represent $C_i$ in its final answer (it can and will determine that, instead, when resolving edges from $S_{\text{min}}$), and hence $e_{ij}$ can be declared vestigial.

The next example shows the concept of minimal cliques.

Example 5. Consider $G$ shown in Figure 3.1 after resolving $e_{45}$ only. Note that the triple
\textbf{Is-Vestigial}(e_{ij}, G, Q, \oplus)

1. \textbf{if} Is-In-Preserving(\(p, \emptyset, a_{\ell}\)) \textbf{then}
2. \textbf{return not} Is-In-a-Minimal-Clique(e_{ij}, G, Q)
3. \textbf{else}
4. \textbf{return not} Is-In-a-Relevant-Clique(e_{ij}, G, Q)

Figure 3.6: Is-Vestigial(.) Function

\((cited \geq 45, \text{ADD, cited})\) is in-preserving. Observe that \(S = \{p_2, p_3, p_4\}\) is a relevant clique \((25 + 20 + 15 = 60 \geq 45)\). However, there exists \(S_{\text{min}} = \{p_2, p_3\} \subset S\) which forms a \textit{minimal} clique \((25 + 20 = 45 \geq 45)\). Therefore, edges \(e_{24}\) and \(e_{34}\) are \textit{vestigial} since both \(e_{24}\) and \(e_{34}\) do not belong to any minimal clique.

The above two theorems suggest that testing for vestigiality can be implemented by checking for relevant and minimal cliques as shown in \textbf{Is-Vestigial(.)} function, see Figure 3.6. However, finding such cliques is \textit{NP-hard} as shown in the next theorem.

\textbf{Theorem 3.} Testing for vestigiality using \textbf{Is-Vestigial(.)} function is \textit{NP-hard}.

\textit{Proof.} This can be shown through a straightforward reduction from the well-known \(k\)-clique problem, and hence is computationally infeasible. \hfill \Box

Thus, implementing \textbf{Is-Vestigial(.)} is impractical as the naive algorithm that calls all the \(O(n^2)\) resolves is going to be faster. Consequently, the challenge is to design a \textit{QDA} strategy that still performs vestigiality testing, but does it fast enough to outperform the naive approach. Thus, in the next section, we will explain how to devise efficient approximation-based techniques for vestigiality testing.
3.5 Query-Driven Solution

In this section we describe our QDA approach. We begin by presenting an overview of the framework. Next, we explain the framework components in more detail.

3.5.1 Overview of the Approach

The main task of the QDA approach is to compute an answer to query $Q$ very efficiently. The answer should be equivalent to first applying a standard algorithm, such as transitive closure (TC) on the whole dataset and then querying the resulting cleaned data with query $Q$.

Recall that traditional TC operates by iteratively choosing a pair of nodes to resolve next, then applying the resolve function, merging nodes if the resolve returns a positive answer, and then repeating the process. The QDA approach is very similar to the original TC with two noticeable differences. First, QDA uses its own pair-picking strategy to select pairs of nodes to resolve next. The goal of this strategy is to minimize the number of calls to resolve to answer the given query. Second, instead of calling resolve on the chosen pair, QDA first tries to quickly determine if it can avoid making this call altogether by checking if the chosen pair is vestigial.

Conceptually, the QDA approach can be viewed as consisting of the following steps:

1. **Creating and Labeling the Graph.** The approach starts by creating and labeling graph $G$ (Section 3.4.3).

2. **Choosing an Edge to Resolve.** Based on its edge-picking policy, the approach selects edge $e_{ij}$ to resolve. Intuitively, such a policy should select $e_{ij}$ in a way that resolving it would allow QDA to either quickly add some cluster-representative to the result-set, or would break many relevant cliques. We have experimented with many different policies.
The one that has demonstrated the best results is based on picking edges according to their weight, where weight $w_{ij}$ for edge $e_{ij}$ is computed by combining the values of its incident nodes: $w_{ij} = \nu_{i\ell} \oplus \nu_{j\ell}$.

3. **Lazy Edge Removal.** We have implemented many optimizations in **QDA**, here we briefly describe one of them. In this step the algorithm checks if the chosen edge $e_{ij}$ still exists. If it does not, then the algorithm will go back to Step 2 to pick another edge. Note that $e_{ij}$ can disappear as the result of merging of two nodes $v_k$ and $v_\ell$. Observe that after merging $v_k$ and $v_\ell$, only edges that are common to both of them must remain in $G$. But checking for common edges and then aggressively removing them from auxiliary data structures at the time of the merge is an $O(|R|)$ operation in general for each merge operation. To reduce this cost, **QDA** does not remove the edges at the time of the merge, but removes them *lazily* in this step. It does so in $O(1)$ time by checking if $v_i$ (or $v_j$) of edge $e_{ij}$ has been merged with some other node $v_k$ by the algorithm on a prior iteration, and hence (i) $v_i$ (or $v_j$) was removed from $V_{\text{maybe}}$, or (ii) $v_i$ is not in $v_j$'s neighborhood or vice versa.

4. **Vestigiality Testing.** The algorithm, in this step, tries to avoid calling resolve on edge $e_{ij}$ by checking if it is *vestigial* (Section 3.5.2).

5. **Stopping Condition.** If there exists an edge $e_{ij} \in E$ that is neither resolved nor vestigial, then the algorithm iterates by going to Step 2.

6. **Computing the Answer.** Finally, the algorithm computes the query’s final answer using the required answer semantics, denoted by $\mathcal{S}$ (Section 3.5.3).

Thus, our goal translates into designing algorithms that implement the above steps. Such algorithms should minimize the number of invocations of the expensive resolve function, and be able to correctly and efficiently find an answer to a given query. In addition, the chosen algorithms must themselves be very efficient, otherwise their cost will dominate the cost of calling the resolve function, and a simple strategy such as resolving all $O(n^2)$ edges
Vestigiality-Testing($e_{ij}, G, Q, \oplus$)

1. if Is-In-Preserving($p, \oplus, a_{\ell}$) and Might-Change-Answer($\emptyset, v_i \oplus v_j, Q$) then
   2. $res \leftarrow R(v_i, v_j)$
   3. if $res = \text{MustMerge}$ then
      4. $A_{\text{cur}} \leftarrow A_{\text{cur}} \cup \{v_i \oplus v_j\}$
      5. $V_{\text{maybe}} \leftarrow V_{\text{maybe}} - \{v_i, v_j\}$
   6. else if $res = \text{MustSeparate}$ then
      7. $E \leftarrow E - \{e_{ij}\}$
   8. else
      9. $\ell[e_{ij}] = \text{maybe}$
   10. else if Check-Potential-Clique($e_{ij}, G, Q$) then
      11. $res \leftarrow R(v_i, v_j)$
      12. if $res = \text{MustMerge}$ then
          13. $v_i \leftarrow v_i \oplus v_j$
          14. $\mathcal{N}[v_i] = \mathcal{N}[v_i] \cap \mathcal{N}[v_j]$
          15. $V_{\text{maybe}} \leftarrow V_{\text{maybe}} - \{v_j\}$
      16. else if $res = \text{MustSeparate}$ then
          17. $E \leftarrow E - \{e_{ij}\}$
      18. else
          19. $\ell[e_{ij}] = \text{maybe}$
      20. else
          21. $E \leftarrow E - \{e_{ij}\}$  // this edge is vestigial

Figure 3.7: Vestigiality-Testing(.) Function

in random order might be more efficient.

The following sections elaborate all of these steps in detail.

### 3.5.2 Vestigiality Testing

Given an edge $e_{ij}$ selected by the edge-picking strategy, the main task of vestigiality testing is to determine if $e_{ij}$ is vestigial and thus calling resolve on it can be avoided. However, from Section 3.4, we know that testing for the exact vestigiality via clique-checking is an NP-hard problem. Hence, QDA employs a highly efficient approximate solution instead of the exact check. Namely, QDA tests for vestigiality by using an inexact-but-fast check to determine if $e_{ij}$ can potentially be part of any relevant clique at all.
The vestigiality testing function, provided in Figure 3.7, can conceptually be viewed as consisting of the following steps:

1. **Edge Min-clique Check Optimization.** This is another optimization which the algorithm employs (Lines 1–9). Here, a check for a special case allows the algorithm to remove two nodes from the graph instead of one in case of a merge, leading to extra savings. Namely, this special case exists when triple \((p, \oplus, a_\ell)\) is in-preserving and edge \(e_{ij}\) by itself can change the current answer to \(Q\). If so, then \(e_{ij}\) is not vestigial and the algorithm calls resolve on it. Now, if resolve returns \text{MustMerge}, then the algorithm adds the merged node \((v_i \oplus v_j)\) to the answer set and then removes \(v_i\) and \(v_j\) nodes from \(G\). The algorithm can perform this optimization because \(v_i\) and \(v_j\) are already represented by their merged representation in \(A_{cur}\).

2. **Check for Potential Clique.** If Step 1 does not apply, then the function calls the \text{Check-Potential-Clique}(.) procedure to test if \(e_{ij}\) can potentially be part of any relevant clique at all. If the call returns \text{true}, then the function calls resolve function on \(e_{ij}\) and labels \(G\) accordingly (Lines 11–19). If the call returns \text{false}, then the function marks \(e_{ij}\) as vestigial (Line 21).

The key intuition behind the \text{Check-Potential-Clique}(.) function is to quickly check if an edge \(e_{ij}\) can potentially be involved in a relevant or a minimal clique at all, see Figure 3.8. It is a safe approximate function: it returns \text{false only} when \(e_{ij}\) is guaranteed not to be a part of any relevant or minimal clique (i.e., vestigial). In contrast, it returns \text{true} when \(e_{ij}\) might be a part of some relevant clique and thus, the algorithm cannot guarantee it is vestigial. For efficiency, the algorithm treats it as non-vestigial without performing any further costly checks.

The idea of this function can be illustrated with the following example. Assume two nodes \(v_i\) and \(v_j\) with citation counts of 10 and 10, and further assume that they have only two
common neighbors whose citation counts are 15 and 20. Then edge $e_{ij}$ cannot be part of any clique with combined citation count above $10 + 10 + 15 + 20 = 55$. Also note that while the edge $e_{ij} = (v_i, v_j)$ might be a part of a clique with $cited = 55$, the algorithm cannot guarantee that without checking the existence of the edge between 15 and 20.

The function that utilizes this intuition is illustrated in Figure 3.8. It begins by merging nodes $v_i$ and $v_j$ and then checking if their merge might change $Q$’s answer. If it does not, then the function computes the intersection of $v_i$ and $v_j$ neighborhoods (Line 4) and then, tries to find the smallest potential clique from their common neighbors which might change $Q$’s answer. The function will keep expanding the size of such clique until no common neighbors are left. Once the function succeeds in finding a potential clique that might change $Q$’s answer then it will return true (Lines 5–9). Otherwise, it returns false (Line 10).

### 3.5.3 Computing Answer of Given Semantics

After the algorithm is done processing edges, it computes its final answer $A_{cur}$ to query $Q$ based on the answer semantics $S$ the user requested. For that, it uses the COMPUTE-ANSWER(.) function illustrated in Figure 3.9. The function starts by adding nodes from $V_{maybe}$ which satisfy $Q$ to $A_{cur}$ (Lines 1–3). At this stage $A_{cur}$ satisfies representative answer
Figure 3.9: Compute-Answer(.) Function

semantics. As such, \( A_{\text{cur}} \) might contain duplicates and/or it might not be equivalent to the canonical merged representation produced by the original TC algorithm.

If the user requests stricter \textit{distinct} or \textit{exact} answer semantics, then the algorithm continues building the corresponding answers based on the current \( A_{\text{cur}} \). The algorithm implements \textit{distinct} semantics by cleaning the (small) representative answers in \( A_{\text{cur}} \) by using the original TC algorithm. That is, duplicates are removed by resolving all pairs of nodes in set \( A_{\text{cur}} \) (Lines 4–9). Thus, the additional cost of cleaning this small result-set is \( O(\frac{1}{\text{divides.alt0}}A_{\text{cur}}/\text{divides.alt0}) \) resolves in the worst case, where \( |A_{\text{cur}}| \) is the size of the result-set.

To generate an answer that satisfies \textit{exact} semantics, the algorithm proceeds by comparing clusters in the result-set with clusters that are not in the result-set. That is, it compares all nodes in \( A_{\text{cur}} \) with all nodes in \( A_{\text{cur}} \cup V_{\text{maybe}} \) (Lines 10–16). Therefore, the extra cost of cleaning leads to \( O(|A_{\text{cur}}| \cdot |R|) \) additional resolves in the worst case.

Note that to produce \textit{distinct} or \textit{exact} answer, edges previously labeled \textit{vestigial} are con-
sidered unresolved edges.

3.5.4 Answer Correctness

From a theoretical perspective, it could be useful to analyze the properties of our QDA algorithm with respect to answer correctness. Note that if the resolve function is always accurate, then TC will compute clustering $C$ that is identical to the ground-truth clustering $C_{gt}$. Consequently, the following lemma holds trivially:

Lemma 8. If the resolve function is always accurate, then QDA will compute answers that are: representationally, distinctly, or exactly equivalent to those in $C_{gt}$.

Lemma 8 essentially states a theoretical result: the QDA algorithm is guaranteed to compute the correct answer, provided that the resolve function is accurate. Naturally, resolve functions are not always accurate, and hence no ER technique can guarantee the correctness of its answer. We also do not assume that resolve is always accurate.

3.5.5 Optimizations of Equality and Range Queries

In this chapter, we have covered very generic algorithms that can be applied to a broad class of cases that are based on categorizing triples into in- and out-preserving and neither. However, various optimizations of these algorithms are possible when considering each specific case separately. In this section, we study a different implementation for equality and range style queries.

Queries with Equality Predicates. In the EQ case, a relevant clique is either a clique that adds up to the value of the query threshold $t$ or a clique that contains a sub-clique which adds up to $t$ and at least one more node with a value greater than 0. However, this is the NP-hard
Sub Set Problem and hence, at first, we implemented a $\frac{3}{4}$ linear approximation algorithm [55] to find an approximate subset sum ($z \ t$). This approximation lead to some unnecessary calls to resolve causing QDA, in the EQ case, to issue more resolves than both GTE and LTE. However, an optimization which postpone the resolution of some edges (instead of greedily resolving them) allowed QDA to issue a slightly smaller number of resolves when compared to the GTE case and a much smaller number of resolves compared to the LTE case.

The intuition behind this optimization is three-fold. First, the algorithm does not create an edge between any two nodes labeled as out. This optimization is similar to an optimization implemented when the triple is out-preserving. However, the process of labeling nodes differs in the EQ case; here, only nodes with values greater than $t$ are labeled as out whereas when the triple is out-preserving, any node that does not satisfy the query’s condition is labeled as out.

Second, when trying to find the set of mutual neighbors, there might exist some nodes such that their values are greater than $t$. Such nodes can be discarded since they cannot lead to a sum value, denoted by $\nu_{\text{sum}}$, equal to $t$. This idea can be further optimized by discarding all nodes with values greater than the dynamic move past point (denoted by MPP) such that \[ MPP = t - \nu_i - \nu_j, \] where $\nu_i$ and $\nu_j$ are the values of edge $e_{ij}$ endpoints since they also cannot contribute to $\nu_{\text{sum}}$ equal to $t$.

Third, after determining the set of mutual neighbors, the algorithm starts to sum them up. If it reaches a point where $\nu_{\text{sum}}$ is greater than or equal to $t$, it invokes a resolve call on edge $e_{ij}$. Otherwise, if $\nu_{\text{sum}} < t$, then it removes $e_{ij}$ (lazily) from graph $G$ (causing other edges to be removed too) and adds it to the postponed edges set (the other edges removed from $G$ are not added to this set). At the end, the algorithm checks whether the edges in the postponed edges set can remove nodes from the answer set. This removal can happen if the value of one of the edge’s endpoints is equal to $t$. 

50
The following example helps in illustrating the previous optimization:

**Example 6.** Consider $G$ shown in Figure 3.1 and the following query:

**Query 3.** \texttt{SELECT \_\_\_ FROM $R$ WHERE cited = 55.}

The algorithm starts by labeling the graph nodes: node 1 is labeled \texttt{out} since 65 $>$ 50, nodes 2, 3, 4, 5 and 6 are labeled \texttt{maybe}. Then, it creates an edge between every two nodes with the exception of not creating an edge between any two \texttt{out} nodes (note that this optimization cannot be utilized in this example since only one \texttt{out} node exists).

At first, all edges are not resolved and the algorithm uses the edge picking strategy to pick an edge and tries to resolve it. Assume that edge $e_{34}$ is picked and nodes 3 and 4 mutual neighbors are \{1, 2, 5, 6\}. Herein, $MMP = 55 - 20 - 15 = 20$ and hence the algorithm discards both nodes 1 and 2 (since their values are greater than $MMP$) and computes the mutual neighbors as \{5, 6\}. Next, the algorithm proceeds and checks whether the combined value of $3 \oplus 4 \oplus 5 \oplus 6 = 20 + 15 + 10 + 5 = 50 \not\geq 55$, and hence the algorithm does not invoke a resolve on $e_{34}$. Instead the algorithm adds it to the postponed edges set and continues by selecting a different edge. Finally, the algorithm revisits edges in the postponed set and checks whether one of their endpoints values is equal to 55. If yes, then it resolves it; if not, then this edge is \texttt{vestigial}.

**Queries with Range Predicates.** The idea behind this optimization is to use two-stage processing. In the first stage an edge is resolved only if it is part of a relevant clique and thus it can result in a new node that should be in the answer. Thus, all nodes that must be in the answer will be in $A_{cur}$, but $A_{cur}$ is a superset of nodes, as it may contain \textit{erroneous} nodes that should not be there. To remove these erroneous nodes from $A_{cur}$, the second stage of the algorithm resolves edges only if an edge is a part of a clique that includes at least one node in $A_{cur}$ and that clique can change the answer for that node from being \texttt{in} $A_{cur}$ to
being \textbf{out} of $A_{\text{cur}}$. This two-stage strategy leads to a noticeable improvement in processing range queries.

\section*{3.5.6 QDA for Different Clusterings}

This section demonstrates that \texttt{QDA} is a generic technique and can be applied to different types of clustering algorithms. However, before illustrating how to apply \texttt{QDA} for different clustering algorithms, we first need to discuss how \texttt{QDA}'s efficiency is affected. In terms of the computational efficiency, it is instructive to look at the eagerness aspect of various existing ER solutions. Eager approaches are those that make their merging decisions early \cite{21, 31}. Variants of transitive closure (TC) that merge records as soon as the resolve function returns a positive decision are examples of eager approaches. Non-eager approaches are those that tend to delay their merge decisions toward the final clustering step. An example of a non-eager approach is correlation clustering (CC) \cite{19} that needs to know outcomes of all $O(n^2)$ resolves to make its final clustering decision. Note that answering the query might not need to know the outcomes of all the resolves, and hence the eager approaches are more conducive to be effective query-driven solutions.

To further cover techniques from both sides of the eagerness spectrum, we next present our \texttt{QDA}-based solution for non-eager approaches. Note that in our experiments we developed \texttt{QDA} for non-eager approaches using \textit{correlation clustering} (abbreviated \texttt{QDA-CC}) as an example. Also note that, \texttt{QDA-CC} cannot guarantee equivalence to CC; however, it finds answers that are very close to those of CC while being more efficient.

\textbf{Query-Driven Approach for Non-Eager Approaches.} This section develops a query-driven solution for a non-eager approach, using CC as an example. In general, developing \texttt{QDA} techniques for non-eager approaches, such as CC, is more challenging, since they need to know the results of all $O(n^2)$ resolves.
The main idea of the solution is that QDA-CC can be implemented by leveraging QDA for TC (see Sections 3.5.1 to 3.5.4). Specifically, observe that if resolve cannot return Uncertain and always returns accurate answers, then QDA is guaranteed to compute an answer equivalent to $C_{gt}$ (see Lemma 7). Let $C_{cc}$ be the clustering computed by CC. Let us modify the resolve function on edge $e_{ij}$ to return the same MustMerge/MustSeparate value as this edge $e_{ij}$ has in $C_{cc}$. Then, according to Lemma 7, QDA with this new resolve function will compute an answer that is equivalent to $C_{cc}$ instead of $C_{gt}$, which is exactly what we need.

The challenge is that we do not know what CC will return as final clusters without calling all the resolves, which defeats the purpose of QDA. To imitate the result of CC without calling all the resolves, QDA-CC first calls QDA but then checks whether the outcome of resolves called by QDA could be different according to CC. If they cannot be, then QDA is supposed to produce equivalent answer to that of CC.

To perform the above check, QDA-CC needs to know if there exists some assignment (of MustMerge/MustSeparate values) of yet-unresolved (by QDA) edges such that CC applied on graph $G$ with that assignment could change the values of the resolves consumed by QDA. To do so, QDA-CC applies a “stress test” – by using two CC extremes. The first extreme (denoted by CC+) clusters records by assuming all unresolved edges are yes edges and then apply CC clustering normally. In contrast, the second extreme (denoted by CC−) marks all unresolved edges as no edges and then executes CC. Intuitively, CC+ tries to group all records into as few clusters as possible with the maximum force – given the known constraints for already resolved edges. In contrast, CC− tries to do the opposite – to split all records into as many clusters as possible with the maximum force. If the resolves called by QDA cannot “flip” (e.g., from yes to no or from no to yes) according to these two stress tests, then the edges are considered “stable”. This test is a very good heuristic to check for stable edges.

More precisely, QDA-CC trusts clusterings obtained by both CC+ and CC− (denoted by $C^+$ and $C^−$ respectively) rather than QDA clustering. Thus, it compares the labeling of each
resolved edge in $C^+$ and $C^-$. If both clusterings agree on edge $e_{ij}$ labeling (e.g., $\ell[e_{ij}^+] = \text{yes}$ and $\ell[e_{ij}^-] = \text{yes}$), then $e_{ij}$ is stable and regardless of $R(v_i, v_j)$, $e_{ij}$ is labeled as yes. Yet, if clusterings disagree (e.g., $\ell[e_{ij}^+] = \text{yes}$ and $\ell[e_{ij}^-] = \text{no}$), then this $e_{ij}$ is unstable and hence labeled maybe. At the end, if all edges are stable then QDA-CC terminates, if not then it proceeds to the next iteration.

In the next iteration, QDA-CC will update $G$ based on the new modifications from $CC^+$ and $CC^-$. For example, it will merge two nodes if the edge between them was deemed to be yes by both $C^+$ and $C^-$. However, if an edge between two records flipped from yes (or no) to maybe, then QDA-CC will neither merge nor separate the two records. It will represent the edge as a maybe edge. To avoid an excessive number of iterations, QDA-CC will try to overcome the destabilization in the graph caused by maybe edges by resolving other edges close to them. Note that QDA-CC will not resolve an edge which had been already resolved. After that, QDA-CC proceeds iteratively from the beginning. It terminates when all edges are stable.

### 3.6 Experimental Evaluation

In this section, we empirically evaluate the efficiency of our QDA approach on real and synthetic data.

In Section 3.6.1, we study QDA for different query types (GTE, LTE, etc.) and compare it to TC in terms of, both, the end-to-end running time and the number of calls to resolve. The experiments will show how the chosen answer semantics affect the efficiency of QDA. We also study the impact of vestigiality testing as the cost of resolve functions is varied. We will demonstrate the result of using/not-using blocking on both QDA and TC. Next, we analyze the effectiveness of our greedy edge-picking strategy. In addition, we compare our QDA-CC
algorithm to a standard CC implementation in terms of the number of resolve invocations. We also study the quality reached by our approaches. We further test how close the result of QDA-CC get to those of CC.

In Section 3.6.2, we study QDA using more complex selection queries with multiple predicates connected via logical connectives. In particular, we study QDA using the three generic categories: in-preserving, out-preserving, and neither.

Finally, in Section 3.6.3, we summarize a few other interesting experiments.

3.6.1 Google Scholar Dataset Experiments

In this section, we evaluate the efficacy of our approach on a real bibliographic dataset collected from Google Scholar. The dataset represents publications of the top 50 computer science researchers each having h-index of 60 or higher [1]. The dataset schema is similar to that of Table 3.1. The dataset consists of 16,396 records where 14.3% are duplicates.

We use two blocking functions to cluster records that might be duplicates together. The first function partitions records (i.e., papers) into buckets based on the first two letters of their titles. Similarly, the second one partitions them based on the last two letters. That is, if two papers match in either their first two letters or their last two letters then they are put in the same block. Note that both TC and QDA use the same blocking procedure.

We have implemented a highly-accurate pairwise resolve function which operates on two records $r_i, r_j \in R$ to decide whether they refer to the same real-world entity. The resolve function utilizes Soft-TF-IDF [28] to compare titles and the Jaro-Winkler distance to compare author names. If the similarity is sufficient (above threshold) and there are $\min(|A_i|,|A_j|)$ authors in common ($|A_i|$ and $|A_j|$ are the number of authors), then resolve returns MustMerge and $r_i, r_j$ are considered to be duplicates.
Figure 3.10: QDA vs. TC [Time]  
Figure 3.11: QDA vs. TC [#R()]  
Figure 3.12: Answer Semantics

**Experiment 1 (QDA vs. TC).** Figures 3.10 to 3.12 use a set of GTE (≥) queries to show the effects of vestigially testing by comparing our QDA algorithm (using representative, distinct, and exact answer semantics) with TC. These algorithms are compared in terms of their end-to-end running time and the number of resolves called.

Figure 3.10 plots the actual end-to-end execution time of QDA (for representative answer semantics) and TC for different values of the GTE query threshold $t$. Figure 3.11 is similar but plots the number of calls to resolve instead of the execution time. These are log-lin scale plots, where $t$’s range is chosen to show the entire spectrum of QDA’s behavior. Note that the curves in Figures 3.10 and 3.11 are similar, thus demonstrating the fact that the calls to resolve are indeed the bottleneck of QDA and TC.
As expected, for all the threshold values and all the answer semantics, QDA is both faster (than TC) and issues fewer resolves (than TC). This is since the query-awareness gives QDA the ability to exploit the in-preserving predicate property to add some records to the result-set without the need to resolve their corresponding edges.

In Figure 3.10, QDA takes only 0.004 seconds when \( t = 1 \) and all records satisfy the query threshold, whereas TC takes 0.52 seconds. This large saving happens because for \( t = 1 \) QDA will label all nodes as \texttt{in} and will not issue any call to resolve. However, for difficult thresholds, e.g., \( t = 128 \), most nodes are labeled \texttt{maybe} and there are many potential cliques that can be added up to 128, which need to be resolved. Thus, QDA resolves 1,770 edges and spends 0.2 seconds to answer the query, while TC takes 0.52 seconds. Note that the number of resolves issued (and thus the time spent) is related to the the number of potential cliques that may satisfy the query. That is, whenever the number of potential cliques that may satisfy the query decreases, the number of calls to resolve (and obviously the time) will decrease and vice versa.

Figure 3.12 presents the end-to-end running time and the number of resolves called by QDA using (more strict) distinct and exact answer semantics. QDA computes \textit{distinct semantics} by first computing the initial result set (denoted by \( RS \)) using QDA with representative semantics, then it de-duplicates \( RS \). The larger the cardinality of \( RS \), the higher the extra cost is. For instance, when \( RS \) is large (e.g., \( t \geq 1 \)) the number of resolve calls is also large. QDA with the \textit{exact semantics} goes one step further and resolves all edges between the records in \( RS \) and the remaining records and thus it is more expensive than QDA for distinct semantics.

Finally, note that techniques like [22], discussed in Chapter 2, are not meant and not designed to optimize for queries with numeric attributes. Unlike QDA, they cannot avoid calling resolves and hence will not be able to outperform TC (and thus QDA).

**Experiment 2 (QDA Speed Up).** Figure 3.13 plots the speed up of QDA (using represen-
Figure 3.13: QDA Speed Up

QDA can be from 1.2 to 100 times faster than TC.

As discussed in Experiment 1, the cost of calling resolve is the dominant factor of the overall execution time. Thus, the end-to-end execution time of QDA depends on the number of potential cliques which may satisfy the query since these potential cliques must be resolved.

For instance, QDA for LTE (≤) takes slightly more time compared to QDA for GTE (yet, still 1.5 to 6 times faster than TC) because QDA using LTE cannot exploit the in-preserving property to add records that satisfy the query threshold to the answer set. However, it now exploits the out-preserving property to remove records from the result-set. In this case, some of the edges connected to these discarded nodes need to be resolved because they might remove records from the result-set if they are declared duplicates.

The EQ (=) predicate is very selective and the number of relevant cliques that may satisfy
the query is much smaller than that for GTE and LTE cases. In the EQ case, a relevant clique is either a clique that adds up to the value of the query threshold $t$ or a clique that contains a sub-clique which adds up to $t$ and at least one more node with a value greater than 0. However, this is the well-known NP-hard Subset Sum Problem and thus we implemented a $\frac{3}{4}$ linear approximation algorithm [55] to find an approximate subset sum ($\approx t$). This approximation lead to some unnecessary calls to resolve, and thus more time, causing $QDA$ for EQ to be slightly more expensive compared to $QDA$ for GTE. $QDA$ for EQ is 1.5 to 6 times faster than TC.

In Figure 3.13, range queries are tested using the predicate $p: t - 50 \leq \text{cited} \leq t$. Recall that range queries are neither in- nor out-preserving (see Table 3.5). $QDA$ for range queries is 1.2 to 6 times faster than TC. It takes a bit more time compared to $QDA$ using LTE since the number of potential cliques which may change the query answer in $QDA$ for range queries is slightly higher because one potential clique may put a record (say $r_i$) in the answer and then another potential clique may remove $r_i$ from it.

Finally, in Figure 3.13 the predicate utilized to test categorical queries is $p: \text{cited} \geq t \land \text{venue} = \text{VLDB'}$. On the one hand, the number of potential cliques which satisfy the query in this case is much smaller when compared to all previous cases (viz., GTE, LTE, EQ, and Range) because $p$ is very selective. On the other hand, $QDA$ spends more time checking for such cliques since they involve a categorical attribute. $QDA$ for categorical queries is 7 to 10 times faster than TC.

**Experiment 3 (Resolve Cost).** Figure 3.14 demonstrates the importance of minimizing the number of calls to resolve, especially when the resolve function is not cheap. This experiment uses a smaller dataset of 448 publications written by a prolific CS professor and tests three different resolve functions of various costs. Function one is the least expensive and uses a normalized edit-distance function to compare titles and authors. The second function is more expensive and calculates Soft-TF-IDF for the titles and Jaro-Winkler distance between
the authors. The third one is the most expensive: it computes TF-IDF for the abstracts of the papers. Note that in general, modern resolve functions can be even more expensive, e.g. involving ontology matching, web queries, etc. Figure 3.14 demonstrates that the gap between QDA and TC increases when the cost of the resolve function increases. Thus, minimizing the number of resolves is very important specifically for non-cheap resolve functions.

**Experiment 4 (Applying Blocking).** Figure 3.15 and 3.16 study the effects of using/not-using blocking on both QDA and TC. Figure 3.15 plots the speed up of QDA over TC and Figure 3.16 shows the percentage of resolves saved by using QDA instead of TC. Note that when no blocking is applied, all publications of an author are put in one block.

As expected, QDA outperforms TC according in both comparison criteria, for all threshold values – even when no blocking is applied. However, QDA’s performance with blocking is better than its performance without blocking. This is since blocking removes some edges from consideration, thus causing the number of potential cliques which satisfy \( t \) to decrease dramatically.

An interesting case is when \( t = 1 \) and thus all records satisfy the query threshold. In that case, QDA without blocking is 1,200 times faster than TC without blocking; whereas QDA
with blocking is 100 times faster than TC with blocking. This is because, for \( t = 1 \), QDA with blocking take comparable amount of time to QDA without blocking, whereas TC with blocking is much faster than TC without blocking.

**Experiment 5 (Edge Picking Strategy).** Figure 3.17 studies the effectiveness of our edge-picking strategy. It compares three different strategies in terms of their end-to-end execution time and the number of calls to resolve: (i) our greedy policy, which chooses edges with higher weights first, (ii) a random policy, which selects edges randomly, and (iii) an enumeration policy that enumerates all minimal cliques and chooses the edge involved in the maximum number of such cliques. Since the third approach is computationally very expensive, we had to conduct this test on a smaller dataset of 177 papers, written by the
same author, to make sure that the test terminates in a reasonable amount of time.

As expected and shown in Figure 3.17, the third strategy tends to be very competitive in terms of the number of resolves called, as it quickly reduces the edge-search space. However, it is by far the worst strategy in terms of the end-to-end execution time. This is because enumerating all minimal cliques is computationally very expensive. In other words, this policy finds good edges, but it spends way too much time to find them.

Thus, our proposed greedy policy surpasses all other techniques: it not only finds good edges that are able to quickly reduce the edge-search space, it also finds them very quickly.

**Experiment 6 (QDA-CC vs. CC).** This experiment compares our QDA-CC algorithm to a standard CC implementation in terms of the number of resolve invocations. It uses the representative semantics to compute the final answer. Note that, in this experiment, we are being conservative by not using blocking (viz., no blocking, see Experiment 4) to show that QDA-CC outperforms CC even in this difficult case.

From Figure 3.18 we observe that QDA-CC issues less number of calls to resolve compared to CC for all $t$ values. Since QDA-CC is cognizant of $t$, it marks some edges as vestigial without resolving them, whereas CC must resolve all $O(|E|^2)$ edges in the graph. Note that, for very large thresholds (e.g., $t = 16,384$), QDA-CC resolves only a small number of edges in the first
iteration. However, most of these few edges “flip” to maybe edges in the “CC+ vs. CC−” test. Hence, to avoid too many iterations, QDA-CC stabilizes the graph by resolving edges that are close to the flipped edges. QDA-CC is also faster than CC in terms of the execution time for all values of t (not shown in plots).

**Experiment 7 (Quality).** Figure 3.19 plots the quality reached by QDA, QDA-CC, TC, and CC when answering cited ≥ t query for various values of t. To compute quality, we use the standard quality metric called F-measure. Specifically, the precision of an entity resolution algorithm \( \mathcal{A} \) (e.g., TC, QDA, etc.) is computed as \( Pr = \frac{|C_{A,Q} \cap C_{GT,Q}|}{|C_{A,Q}|} \), where \( C_{A,Q} \) is the set of clusters returned by \( \mathcal{A} \) that satisfy query \( Q \), and \( C_{GT,Q} \) is the set of clusters in the ground truth which satisfy \( Q \). The recall is computed as \( Re = \frac{|C_{A,Q} \cap C_{GT,Q}|}{|C_{GT,Q}|} \). The F-measure is computed as the harmonic mean of the precision and recall \( F = \frac{2 \cdot Pr \cdot Re}{Pr + Re} \).

In order to measure the quality accurately, we use a synthetic dataset (where its ground truth is known) in this test. This dataset is generated using an enhanced version of the UIS data generator [47]. UIS has been applied on a clean dataset of 100 tuples to generate datasets of size 100 to 1000 tuples by injecting various types of errors in it. Note that we modified our resolve function \( \mathfrak{R}(\) to make errors occasionally. That is, after resolving a pair of records and returning the decision, it reverses its decision intentionally to make a mistake. The percentage of reversed decisions (that is, errors) is 5%. We had to modify our resolve
since otherwise all methods will be very accurate and all will reach nearly perfect quality. The majority of the introduced errors will merge distinct nodes.

From Figure 3.19 we observe that all four algorithms reach very high quality for small (1 to 16) and large ($\geq 512$) thresholds. We also observe that the quality of QDA and TC drops significantly when $t = 128$ because of their eagerness-to-merge behavior. Recall that, most introduced errors merge distinct nodes; and hence, construct an incorrect cluster which satisfies the query (i.e., causing a drop in the precision).

Note that in this plot the results of CC-based strategies turned out to be better than those of TC. However, this is not always the case in general and the reverse might be true for different datasets, e.g., see [43].

**Experiment 8 (QDA-CC Deviation).** In this experiment, we test how close the results of QDA-CC get to those of CC. We measure this closeness in terms of the F-measure where the results of CC are treated as the ground truth.

Figure 3.20 demonstrates that our tests of $CC^+$ and $CC^-$ work very well, as QDA-CC reaches very high F-measure values. Note that even when the value of F-measure drops to 0.95, QDA-CC reaches about the same quality as CC in Figure 3.19 – since they both make errors, some of which are different.
3.6.2 Hotels Dataset Experiments

In this section, we run several queries on a real hotels dataset, which is larger than the Google Scholar dataset used in the previous section. This dataset includes hotels information (e.g., hotel-id, hotel-name, hotel-address, hotel-city, hotel-country, hotel-stars, hotel-price, etc.). It contains 184,169 hotels where almost 40% are duplicates.

We use min-hashing to partition the 184,169 records into 1,000 big blocks. Next, we apply the same blocking technique used in the previous section to further partition these big blocks. That is, we partition the records in each big block into smaller blocks based on the first two letters and the last two letters of the hotel’s name. As a result, if the names of two hotels in one big block match in either the first or last two letters then they are put in the same small block.

We implemented a pairwise resolve function which operates on two records \( r_i, r_j \in R \) to decide whether they refer to the same real-world entity. It uses Soft-TF-IDF to compare the names of hotels.

We can classify the different queries used in these tests into three different classes.

1. **Class one – Inexpensive good hotels in the US.** Queries in this class consist of the three predicates \( p_1 : price \leq t_1 \), \( p_2 : stars \geq t_2 \), and \( p_3 : country = \text{‘US’} \). Thus, these queries consist of three triples: an in-preserving triple \( \tau_1 = (price \leq t_1, \text{MIN}, price) \), an in-preserving triple \( \tau_2 = (stars \geq t_2, \text{MAX}, stars) \), and an in-preserving triple \( \tau_3 = (country = \text{‘US’}, \text{EXEMPLAR}, country) \). From Table 3.5, we can see that the resulting combination \( \tau_1 \land \tau_2 \land \tau_3 \) is in-preserving.

2. **Class two – Overpriced hotels.** Queries in this class consist of the two predicates \( p_1 : price \geq t_1 \) and \( p_2 : stars \leq t_2 \). Hence, such queries consist of two triples: an out-preserving triple \( \tau_1 = (price \geq t_1, \text{MIN}, price) \) and an out-preserving triple \( \tau_2 = (stars \leq \text{MAX}, stars) \).
3. Class three – Poor quality hotels. Queries in this class consist of the two predicates $p_1: \text{stars} \leq t_1$ and $p_2: \text{country} = t_2$. Therefore, these queries consist of two triples: an out-preserving triple $\tau_1 = (\text{stars} \leq t_1, \text{MAX}, \text{stars})$ and an in-preserving triple $\tau_2 = (\text{country} = t_2, \text{EXEMPLAR}, \text{country})$. From Table 3.5, we can see that the resulting combination $\tau_1 \land \tau_2$ is neither in nor out-preserving.

Experiment 9 (In-preserving Triples). In this test, we compare QDA (using the repre-
Figure 3.25: Q5: \(\text{price} \geq t \land \text{stars} \leq 1\)  

Figure 3.26: Q6: \(\text{price} \geq t \land \text{stars} \leq 3\)

sentative answer semantics) versus TC in terms of the number of calls to resolve. This test shows the effects of testing for vestigiality by using four queries that belong to class one.

Figures 3.21 to 3.24 plot the number of resolves invoked by QDA and TC for all four queries. As expected, QDA outperforms TC since QDA is aware of the query thresholds \((t_1, t_2, \text{ and } t_3)\) while TC is not. This thresholds awareness gives QDA the ability to exploit the in-preserving predicate property to add some records to the result-set without the need to resolve their corresponding edges.

In Figures 3.21 and 3.22, where the range of the stars rating is large (viz., \(\geq 1\) and \(\geq 2\)), we can observe that when the price increases the number of calls to resolve decreases. This is because the number of records which satisfy the query increases and thus many records are labeled in causing the edges connected to them to be vestigial.

However, in Figures 3.23 and 3.24, where the range of the stars rating is not large (viz., \(\geq 3\) and \(\geq 4\)), we can see that when the price increases the number of calls to resolve neither increases nor decreases. This is because the majority of records are labeled maybe and there are many potential cliques that can satisfy the query.

**Experiment 10 (Out-preserving Triples).** In this experiment, we use two queries that belong to class two to compare QDA (using representative answer semantics) and TC in terms
of the number of resolves called. Figures 3.25 to 3.26 show the number of resolves invoked by QDA and TC for the two queries.

QDA is superior to TC for both queries. However, in this test, QDA issues a higher number of resolves compared to QDA in the previous test. This is because, here, QDA can only exploit the out-preserving property to remove records from the result-set. Recall that, some of the edges connected to these discarded nodes need to be resolved because they might remove records from the result-set if they are declared duplicates.

Note that Figures 3.25 and 3.26 look similar since the two predicates are opposites. That is, when \( p_1 \) is satisfied, \( p_2 \) is not, and vice versa. For example, in Figure 3.25, there are not many hotels with \( \text{price} \geq 200 \) and \( \text{stars} \leq 1 \); thus, many nodes are labeled \text{out} and the edges between these nodes are vestigial. In contrast, in Figure 3.26, there are many hotels with \( \text{price} \geq 200 \) and \( \text{stars} \leq 3 \); however, there are not many potential cliques that can remove them from the answer. As a result, many of the edges connected to these nodes are vestigial.

**Experiment 11 (Neither in- not out-preserving Triples).** This experiment uses two queries that belong to class three to compare QDA vs. TC in terms of the number of resolves. Figures 3.27 and 3.28 demonstrate the number of calls to resolve by both QDA and TC for these two queries.
As shown in Figures 3.27 and 3.28, QDA outperforms TC for both queries. Note that the number of resolves called in this test is relative to the number of hotels that are located in the country. That is, the more hotels in the country, the larger the number of potential cliques that may alter the query answer, and thus the more resolves that need to be called.

### 3.6.3 Discussion

Here we summarize a few other interesting experiments.

**Analysis of QDA+ (≥/ADD).** QDA+ is a QDA approach for TC+, instead of TC. TC+ is a version of TC which treats MustSeparate softly. That is, when RN(pi, pj) returns MustSeparate, pi and pj are not merged at the moment due to a lack of evidence supporting a merge, but clusters containing pi and pj may be merged later due to transitivity.

Let us consider the performance of QDA+, by looking at few cases. For the triple (cited ≥ 64, ADD, cited), the percentage of resolves saved by QDA+ over TC+ is 46.4% whereas QDA saves 69.46% of resolves over TC. For p : t ≥ 128, the savings percentage for QDA+ is 30.07% while it is 65.04% for QDA. This lower achieved gain is expected, since QDA can merge (or partition) the graph by exploiting the yes/no edges, whereas QDA+ can only merge nodes (i.e., exploite the yes edges only).

**Test of MAX semantics.** This experiment tests the MAX combine function for GTE queries. Note that this case is trivial, as for all threshold values no node is labeled maybe. That is, a node either satisfies t and hence is labeled in; or does not satisfy t and thus is labeled out. There are no two nodes which can be combined to change the result-set. Thus, QDA answers such queries very efficiently, without making a single call to resolve.
3.7 Conclusions

In this chapter, we have studied the query-driven entity resolution problem in which data is cleaned “on-the-fly” in the context of selection queries. We have developed a query-driven entity resolution framework which efficiently issues the minimal number of cleaning steps solely needed to accurately answer the given selection query. We formalized the problem of query-driven ER and showed empirically how certain cleaning steps can be avoided based on the nature of the query.
Chapter 4

QuERy: A Framework for Integrating ER with Query Processing

4.1 Introduction

While QDA, described in Chapter 3, provided a strong foundation for analysis-aware data cleaning, it is limited to only selection queries executed on top of a single entity-set containing duplicates. Data analysis, however, often requires a significantly more complex type of queries requiring SQL-style joins. For instance, a user interested in comparative shopping may wish to find cellphones that are listed on two distinct data sources: Best Buy and Walmart to compare their ratings and reviews. Clearly, the query that corresponds to the user’s interest will require joining Best Buy’s and Walmart’s cellphone-listings.

This chapter explores the problem of analysis-aware cleaning for the more general case where queries can be complex SQL-style selections and joins spanning single/multiple dirty entity-sets. We propose QuERy, a novel framework for integrating ER with SQL query processing. The objective of QuERy is to efficiently and accurately answer complex SQL-style selection and
join queries issued on top of dirty data. The predicates in those queries may be associated with any attribute in the entity-sets being queried.

In particular, we develop two different solutions that utilize the selectivities offered by the query predicates to reduce the amount of cleaning and thus minimize the total execution time of the query. Our solution relies on novel polymorphic operators, which are analogous to the common relational algebra operators (i.e., selections and joins) with one exception: they know how to test the query predicates on the dirty data prior to cleaning it.

Overall, the main contributions of this chapter are:

- We propose QuERy, a novel framework that integrates ER with query processing to answer complex SQL-style queries issued on top of dirty data (Sections 4.2 and 4.4).
- We introduce and formalize the notion of polymorphic operators – a key concept in QuERy (Section 4.5).
- We develop two different solutions: lazy-QuERy and adaptive-QuERy, which reap the benefits of evaluating the query predicates to minimize the query execution time (Sections 4.6 and 4.7).
- We conduct extensive experiments to evaluate the effectiveness of both lazy-QuERy and adaptive-QuERy solutions on real and synthetic datasets (Section 4.8).

This chapter is organized as follows. Section 4.2 sets up our problem while Section 4.3 covers the ER preliminaries. The problem of integrating ER with query processing is defined in Section 4.4. Section 4.5 explains the notion of polymorphic operators. Our lazy-QuERy and adaptive-QuERy solutions are described in Sections 4.6 and 4.7 respectively. The experimental results are presented in Section 4.8. Finally, we conclude this chapter in Section 4.9.
| BB, iPhone 6, 4.7” retina..., 415, 4.6, Apple, apple.com, USA |
| BB, iPhone 5, 4” retina..., 220, 3.9, Apple, apple.com, US |
| BB, Galaxy S5, 5.1” super..., 275, 4.3, Samsung, samsung.com, S. Korea |
| WM, Apple Inc., www.apple.com, USA, iPhone-VI, 550, 4.8 |
| WM, Samsung, www.samsung.com, South Korea, Galaxy S-V, 180, 4.5 |

Table 4.1: A Collection of Raw Records from Two Distinct Data Sources

### 4.2 Problem Setup

This chapter addresses the problem of jointly cleaning and querying (potentially dirty) entity-sets. To better motivate our work, before formalizing the problem and describing our solution, we first discuss a concrete example context wherein such cleaning challenges arise.

Consider an analyst wishing to explore popular electronic items produced in the “USA” that customers buy and write reviews about. The analyst identifies multiple relevant online data sources like Google Shopping, eBay, Walmart, Best Buy, Yelp, etc., which contain raw records describing (similar) entities. Suppose that the analyst chooses two such data sources: Best Buy (denoted by BB) and Walmart (denoted by WM). Table 4.1 shows a toy example of six raw records collected from these two sources. In general, such datasets can be large and contain many entities. Suppose that the analyst wishes to quickly identify cellphones that are listed on both data sources with at least 300 reviews, and that have been manufactured in the “USA”.

In order to execute her analysis task, the analyst needs to utilize semi-automated tools (e.g., Trifacta [9], Data Wrangler [54], C4 [5], OpenRefine [2], SnapLogic [8], etc.) to convert the raw data into a form that is more convenient for data consumption, and hence allows for analysis execution.
In Tables 4.2 and 4.3, we present a transformed dataset instance that results from the raw records in Table 4.1 by collating entities of the same type into a single entity-set (see Section 4.3.1). In this transformed representation, cellphone entities are stored in Cellphones entity-set (denoted by $C$), and manufacturer entities are stored in Manufacturers entity-set (denoted by $M$). Note that there exists only one manufacturer per cellphone and therefore, we can augment each cellphone entity with its corresponding $m_{id}$, $c_{reviews}$, and $c_{ratings}$ values. Alternatively, we could have transformed the raw data into three tables: Cellphones entity-set, Manufacturers entity-set, and a relationship-set that contains the relationships between the cellphones and the manufacturers.

In such a transformed representation, since entities of the same type are collated into a single entity-set, standard ER algorithms like Swoosh [21], Sorted Neighbor (SN) [46], etc. can be used to cluster entities that co-refer into a single canonical representation\(^7\). For instance, entities $\{c_1, c_4\}$ and $\{c_3, c_5\}$ which refer to the same real-world object will be resolved and replaced by a common representation. Likewise, entities $\{m_1, m_2, m_4\}$ and $\{m_3, m_5, m_6\}$ are duplicates and will be replaced by their canonical representations. After this step, the data is ready for consumption and the analyst can pose her analysis task in the form of SQL-like queries. Figure 4.1 presents a process diagram of such a standard approach.

\(^7\)Such ER algorithms typically assume that each entity-set contains a set of entities of the same type.

<table>
<thead>
<tr>
<th>block</th>
<th>c_source</th>
<th>c_id</th>
<th>c_name</th>
<th>m_id</th>
<th>c_reviews</th>
<th>c_ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>BB</td>
<td>$c_1$</td>
<td>iPhone 6</td>
<td>$m_1$</td>
<td>415</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>BB</td>
<td>$c_2$</td>
<td>iPhone 5</td>
<td>$m_2$</td>
<td>220</td>
<td>3.9</td>
</tr>
<tr>
<td></td>
<td>WM</td>
<td>$c_4$</td>
<td>iPhone-VI</td>
<td>$m_4$</td>
<td>550</td>
<td>4.8</td>
</tr>
<tr>
<td>C2</td>
<td>BB</td>
<td>$c_3$</td>
<td>Galaxy S5</td>
<td>$m_3$</td>
<td>275</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>WM</td>
<td>$c_5$</td>
<td>Galaxy S-V</td>
<td>$m_5$</td>
<td>180</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>WM</td>
<td>$c_6$</td>
<td>Galaxy S-III</td>
<td>$m_6$</td>
<td>95</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 4.2: Cellphones Entity-set ($C$)
Note that in the above approach data is cleaned fully prior to executing the query. However, the bulk of this cleaning, as we will see later, might be completely unnecessary since only a small portion of the data might influence the results of the user’s analysis.

Based on this intuition, we propose QuERy, a framework that integrates ER with query processing into a single joint execution. In QuERy, after the analyst has restructured her data (viz., converted the raw records into entity-sets), she can postulate her queries over these entity-sets prior to cleaning them. Instead of cleaning the data fully beforehand and then executing the query, QuERy exploits the query semantics to reduce the cleaning overhead by deduplicating only those parts of data that influence the query’s answer. QuERy is agnostic to the specific technique of ER and thus, any ER algorithm (e.g., [21, 46]) can be used to clean the necessary data parts. Figure 4.2 depicts a process diagram of our proposed QuERy framework.
4.3 ER Preliminaries

In this section, we first present common ER notation, and then discuss the standard phases of ER which are necessary to explain our approach.

4.3.1 Entity-sets

Let \( \mathcal{D} \) be a dataset instance that holds a number of entity-sets, \( \mathcal{D} = \{ R, S, T, \ldots \} \). Each entity-set \( R \) contains a set of entities of the same type, \( R = \{ r_1, \ldots, r_{|R|} \} \) where \( r_i \) represents the \( i^{th} \) entity in \( R \) and \( |R| \) is its cardinality (s.t. \( 1 \leq i \leq |R| \)). Entity-set \( R \) is considered *dirty* if at least two entities \( r_i, r_j \in R \) represent the same real-world object, and hence \( r_i \) and \( r_j \) are duplicates. The attributes in \( R \) are denoted as \( A^R = \{ R.a_1, \ldots, R.a_{|A^R|} \} \), where \( |A^R| \) is the arity of \( R \). Thus, the \( k^{th} \) entity in \( R \) is defined as \( r_k = \{ \nu_{k1}, \ldots, \nu_{k|A^R|} \} \), where \( \nu_{ki} \) is the value of the \( i^{th} \) attribute in entity \( r_k \) (s.t. \( 1 \leq k \leq |R| \) and \( 1 \leq i \leq |A^R| \)).

Entity-set \( C \), shown in Table 4.2, is dirty since it contains two duplicate pairs \( \{ c_1, c_4 \} \) and \( \{ c_3, c_5 \} \). Likewise, entity-set \( M \), presented in Table 4.3, is dirty since manufacturers \( \{ m_1, m_2, m_4 \} \) and \( \{ m_3, m_5, m_6 \} \) are duplicates and refer to the same real-world objects.
4.3.2 Standard ER Phases

As we have seen in Chapter 2, a typical ER cycle consists of two phases: a blocking phase and a deduplication phase.

4.3.2.1 Blocking Phase

Blocking divides entities of the same entity-set into (possibly overlapping) smaller blocks. In our approach, we divide each entity-set \( R \in \mathcal{D} \) into a set of blocks \( B^R = \{ R_1, \ldots, R_{|B^R|} \} \) using one or more blocking functions.

For example, in Table 4.2, we used a blocking function to partition the cellphones entity-set into blocks based on the first two letters of their names. The first column of the table in Table 4.2 represents the block in which the entities reside. Entity-set \( C \) has two blocks \( C_1 \) and \( C_2 \).

4.3.2.2 Deduplication Phase

The goal of the (potentially expensive) deduplication phase is to detect, cluster, and then merge duplicate entities. It consists of three sub-phases: similarity computation, clustering, and merging, which can be intermixed.

**Similarity Computation** phase determines for each pair of entities within the same block whether they co-refer or not. In general, various ER algorithms use different techniques to perform this step. In our approach, any ER algorithm (e.g., Swoosh [21], Sorted Neighbor
Clustering phase groups duplicate entities in entity-set $R$ into a set of non-overlapping clusters $C^R = \{C^R_1, \ldots, C^R_{|C^R|}\}$. That is, any two entities $r_i$ and $r_j$ from the same cluster should co-refer, whereas any two entities $r_k$ and $r_\ell$ from two distinct clusters $C^R_m$ and $C^R_n$ should not co-refer.

Merging phase combines entities of each cluster into a single object that will represent the cluster to the end-user or application in the final result. Let $O^R = \{o^R_1, \ldots, o^R_{|O^R|}\}$ be the object-set that results from merging clustering $C^R = \{C^R_1, \ldots, C^R_{|C^R|}\}$ such that each object $o^R_i \in O^R$ represents the merging of cluster $C^R_i \in C^R$.

A merge function $m^R(C^R_m)$ will consolidate the elements in cluster $C^R_m$ to produce a new object $o^R_m$. As in Chapter 3, to merge two duplicate entities $r_i, r_j \in C^R_m$, a combine function is used for each attribute in $A^R$. A combine function $\oplus_{R.a_\ell}$ takes two values of attribute $R.a_\ell \in A^R$ and outputs a single value. Such a combine function performs different operations based on the type of attribute $R.a_\ell$. Note that the selection of these combine functions is domain-dependent and hence, should be done wisely by the analyst/user.

If $R.a_\ell$ is a numeric attribute, then we consider:

- **ADD semantics**: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{i\ell} + \nu_{j\ell}$,
- **AVG semantics**: $\nu_{i\ell} \oplus \nu_{j\ell} = \text{avg}(\nu_{i\ell}, \nu_{j\ell})$,
- **MAX semantics**: $\nu_{i\ell} \oplus \nu_{j\ell} = \max(\nu_{i\ell}, \nu_{j\ell})$,
- **MIN semantics**: $\nu_{i\ell} \oplus \nu_{j\ell} = \min(\nu_{i\ell}, \nu_{j\ell})$.

In general, the ADD and AVG semantics are used when the duplicate entities are split in parts. For example, the ADD semantics are utilized when the values of attribute $R.a_\ell$ are additive, e.g., citations count in a publications’ table. In contrast, the AVG semantics are used when...
the values of attribute $R.a_\ell$ are not additive, e.g., $c$.ratings in entity-set $C$.

The $\text{MAX}$ and $\text{MIN}$ semantics are usually applied to attributes that monotonically increase or decrease over time, such as $c$.reviews in entity-set $C$.

If $R.a_\ell$ is a categorical attribute, then we consider:

- **UNION** semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{i\ell} \cup \nu_{j\ell}$,
- **EXEMPLAR** semantics: $\nu_{i\ell} \oplus \nu_{j\ell}$ chooses either $\nu_{i\ell}$ or $\nu_{j\ell}$ according to some policy.

The **UNION** semantics are utilized when the system needs to retain all possible values of some attribute, e.g., $c$.name in entity-set $C$. In contrast, the **EXEMPLAR** semantics are used when one value holds richer information than the other value, e.g., in the $m$.name attribute, the value “Apple Inc.” dominates “Apple”.

If $R.a_\ell$ is an identifier (e.g., $c$.id) or a reference attribute (e.g., $m$.id in entity-set $C$), then we consider:

- **UNION** semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{i\ell} \cup \nu_{j\ell}$.

Note that the aforementioned combine functions have the commutativity and associativity properties defined as:

1. Commutativity: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{j\ell} \oplus \nu_{i\ell}$
2. Associativity: $(\nu_{i\ell} \oplus \nu_{j\ell}) \oplus \nu_{k\ell} = \nu_{i\ell} \oplus (\nu_{j\ell} \oplus \nu_{k\ell})$

Since these properties hold, the representation of the merged object (e.g., object $0_m^R$) will always be the same regardless of the merge order within $C_m^R$.

Tables 4.4 and 4.5 show the object-sets $\mathcal{O}^C = \{o_1^C, o_2^C, o_3^C, o_4^C\}$ and $\mathcal{O}^M = \{o_1^M, o_2^M\}$, which
<table>
<thead>
<tr>
<th>object</th>
<th>c_source</th>
<th>c_id</th>
<th>c_name</th>
<th>m_id</th>
<th>c_reviews</th>
<th>c_ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>o^C_1</td>
<td>{BB,WM}</td>
<td>{c_1,c_4}</td>
<td>iPhone 6,iPhone-VI</td>
<td>{m_1,m_4}</td>
<td>415</td>
<td>4.7</td>
</tr>
<tr>
<td>o^C_2</td>
<td>BB</td>
<td>c_2</td>
<td>iPhone 5</td>
<td>m_2</td>
<td>220</td>
<td>3.9</td>
</tr>
<tr>
<td>o^C_3</td>
<td>{BB,WM}</td>
<td>{c_3,c_5}</td>
<td>Galaxy S5,Galaxy S-V</td>
<td>{m_3,m_5}</td>
<td>180</td>
<td>4.4</td>
</tr>
<tr>
<td>o^C_4</td>
<td>WM</td>
<td>c_6</td>
<td>Galaxy S-III</td>
<td>m_6</td>
<td>95</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 4.4: Cellphones Object-set (O^C)

<table>
<thead>
<tr>
<th>object</th>
<th>m_source</th>
<th>m_id</th>
<th>m_name</th>
<th>m_url</th>
<th>m_country</th>
</tr>
</thead>
<tbody>
<tr>
<td>o^M_1</td>
<td>{BB,WM}</td>
<td>{m_1,m_2,m_4}</td>
<td>Apple Inc.</td>
<td><a href="http://www.apple.com">www.apple.com</a></td>
<td>USA</td>
</tr>
<tr>
<td>o^M_2</td>
<td>{BB,WM}</td>
<td>{m_3,m_5,m_6}</td>
<td>Samsung</td>
<td><a href="http://www.samsung.com">www.samsung.com</a></td>
<td>South Korea</td>
</tr>
</tbody>
</table>

Table 4.5: Manufacturers Object-set (O^M)

resulted from deduplicating entity-sets C and M using a traditional cleaning algorithm, e.g., [21,46]. In Tables 4.4 and 4.5, we assume that the analyst picked:

1. The **MIN** semantics to combine attribute c_reviews,
2. The **AVG** semantics to combine attribute c_ratings,
3. The **EXEMPLAR** semantics to combine attributes: m_name, m_url, and m_country, and
4. The **UNION** semantics otherwise.

### 4.4 ER and Query Processing

In this section, we first introduce the concepts of queries and query trees in Section 4.4.1. Section 4.4.2 shows how to evaluate predicates applied to multi-valued attributes. In Section 4.4.3, we present a standard solution to answer queries on top of dirty data. Finally, we formally define our problem in Section 4.4.4.
4.4.1 Queries and Query Trees

We will consider flat SQL queries [48] with \textsc{and} as the only boolean connective in their qualification, similar to the following syntax:

\begin{verbatim}
SELECT target-list FROM \mathcal{D} WHERE \Phi
\end{verbatim}

where \(\Phi\) denotes (i) the equi-join\(^8\) predicate(s) connecting entity-sets in \(\mathcal{D}\) and (ii) the optional selection predicates\(^9\) applied to attributes in entity-sets in \(\mathcal{D}\).

A flat SQL query is often represented by a select-project-join (SPJ) query tree. The leaves of such a tree are relations while the non-leaf nodes are relational algebra operators, e.g., selections (\(\sigma\)), projections (\(\pi\)), joins (\(\Join\)), rename (\(\rho\)), etc. Each intermediate node encapsulates a single task that is required to execute the query. The edges of such a query tree represent data flow from the bottom to the top.

4.4.2 Evaluating Set Values

As presented in Section 4.3, if the \textsc{union} semantics are used to merge a non numeric attribute (viz., a categorical, an identifier, or a reference attribute), then the value of this attribute will be multi-valued in the form of a set, e.g., attributes source, \(c_id\), \(c_name\), and \(m_id\) in Table 4.4. To evaluate predicate \(\varphi_k \in \Phi\) defined on one of these attributes, we must overload its (\(=\)) operator as discussed next.

Let \(op_i\) and \(op_j\) be the two operands in predicate \(\varphi_k : op_i = op_j\). The following four different cases exist since \(\varphi_k\) has two operands:

1. If both operands \(op_i\) and \(op_j\) are single-valued, then \(\varphi_k\) is \textbf{true} if \(op_i = op_j\).

\(^8\)An equi-join is a special type of join that only uses the (\(=\)) operator in the join-predicate.
\(^9\)If the selection predicate is applied to a non numeric attribute, then we solely consider the (\(=\)) operator.
2. If operand \( op_i \) is a single value but operand \( op_j \) is a set, then \( \varphi_k \) is true if \( op_i \in op_j \).

3. If operand \( op_i \) is a set but operand \( op_j \) is a single value, then \( \varphi_k \) is true if \( op_j \in op_i \).

4. If both operands \( op_i \) and \( op_j \) are sets, then \( \varphi_k \) is true if \( op_i \cap op_j \neq \emptyset \).

### 4.4.3 Standard Solution

Let us use an illustrative example to present the standard solution of answering queries on top of dirty data. Suppose that a user interested in comparative shopping wishes to find popular cellphones, say with at least 300 reviews, that have been manufactured in the “USA” and are listed on two distinct data sources: Best Buy and Walmart. Query 1, shown in Figure 4.3, represents the user’s interest.

The execution plan for Query 1, which is selected by some query optimizer, is shown in Figure 4.4. This plan is assumed to contain the best operators placement and join ordering. The result of executing this plan on the dirty tables shown in Figures 4.2 and 4.3 prior to cleaning them is the empty set \( \emptyset \). However, this is incorrect since the first clean object \{iPhone 6, iPhone-VI\} is listed on both sources, it has more than 300 reviews, and it has been manufactured in the “USA”. Thus, it should be returned as the answer to Query 1.
The standard way to answer Query 1 is to first apply the blocking phase on the dirty entity-sets $C$ and $M$, then to fully deduplicate $C$ and $M$ to create object-sets $O^C$ and $O^M$ (i.e., to obtain the tables in Tables 4.4 and 4.5), and finally, to compute the query over these object-sets. This corresponds to inserting the cleaning operator (denoted by $\delta$) directly above the tree leaves (viz., entity-sets) as shown in Figure 4.5.

However, such an approach could be very expensive as it might clean unnecessary blocks. For instance, we note that no cellphones from $C_2$ could satisfy Query 1. This is because if we enumerate all potential objects inside $C_2$ (viz., $\{c_3\}, \{c_5\}, \{c_6\}, \{c_3, c_5\}, \{c_3, c_6\}, \{c_5, c_6\}, \text{and } \{c_3, c_5, c_6\}$) none of them will satisfy Query 1 as the maximum number of reviews of all potential objects inside $C_2$ is 275, which does not satisfy the reviews criteria, i.e., $\geq 300$. Thus, all cellphones in $C_2$ will not be present in the query answer. As a result, deduplicating $C_2$ can be eliminated. By using a similar kind of reasoning, we can note that deduplicating $M_2$ is also unnecessary. Based on this intuition, we will build a principled solution for integrating ER with query processing in Sections 4.6 and 4.7 which demonstrates outstanding results, as shown in Section 4.8.

### 4.4.4 Problem Definition

Given a query $Q$, let $O^f$ denote the set of objects that satisfy $Q$ when all entity-sets in $D$ are cleaned first. Also, let $O^q$ be the set of objects returned by QuERy as the answer to $Q$. Then, we can formally define our problem as an optimization problem as follows:

\[
\text{Minimize: } \quad \text{Execution time of } Q \\
\text{Subject to: } \\
1. \forall o \in O^q, o \text{ satisfies } Q; \quad // \text{Query satisfaction} \\
2. O^q = O^f; \quad \quad // \text{Answer correctness}
\]
4.5 Polymorphic Query Trees

Before developing our approach, in this section, we formalize the notion of polymorphic operators, which is a key concept in QuERy.

4.5.1 Polymorphic Operators

In order to correctly answer a query while performing only a minimal amount of cleaning, we introduce the concept of polymorphic operators. Such polymorphic operators are analogous to the common relational algebra operators (i.e., selections and joins) with one exception: they know how to test the query predicates on the dirty data prior to cleaning it. These operators are called polymorphic since they accept as input not only clean data (objects) as regular operators, but also dirty data (blocks).

Let us define these novel operators formally. Let $R$ and $S$ be two entity-sets in $\mathcal{D}$ that are split into two sets of blocks $B^R = \{R_1, \ldots, R_{|B^R|}\}$ and $B^S = \{S_1, \ldots, S_{|B^S|}\}$. Also, let $O^R = \{o^R_1, \ldots, o^R_{|O^R|}\}$ and $O^S = \{o^S_1, \ldots, o^S_{|O^S|}\}$ be the object-sets that would result if $R$ and $S$ are fully cleaned. Note that the objective of QuERy is to compute the query answer without fully computing $O^R$ and $O^S$, unless obviously if the query answer requires computing them.
fully.

To make our next definitions clear, let us denote the power set of block \( R_i \) as \( \mathcal{P}(R_i) \). Power set \( \mathcal{P}(R_i) \) is the set of all subsets of \( R_i \), including the empty set and \( R_i \) itself. The merging of all entities in subset \( R'_i \in \mathcal{P}(R_i) \) would form a potential object \( \theta^{R'_i} \). Let the set of all potential objects inside block \( R_i \) be \( \Theta^{R_i} = \{ \theta^{R_i}_{\phi_k}, \ldots, \theta^{R_i}_{\phi_k|\mathcal{P}(R_i)|} \} \). Note that not all potential objects in \( \Theta^{R_i} \) will become real-world objects as a single entity cannot belong to two real-world objects.

A polymorphic selection is a unary operation, denoted by \( \varsigma_{\phi_k} \), where \( \phi_k \) is the propositional predicate. If the input to this operator is clean data (viz., object-set, say \( O^R \)), then it acts as the normal selection operator \( \sigma_{\phi_k} \) but applied to objects instead of tuples. That is, it selects all those objects in \( O^R \) that satisfy \( \phi_k \). However, if its input is dirty data (viz., a set of blocks, say \( B^R \)), then it selects all those blocks in \( B^R \) for which \( \phi_k \) holds as defined next:

**Definition 1.** Block \( R_i \in B^R \) satisfies \( \varsigma_{\phi_k} \) if there exists a potential object \( \theta^{R'_i} \in \Theta^{R_i} \) such that \( \theta^{R'_i} \) satisfies \( \sigma_{\phi_k} \).

Intuitively, \( \varsigma_{\phi_k} \) would select those blocks \( R_i \in B^R \) that contain at least one potential object that satisfies \( \sigma_{\phi_k} \). The next example helps in illustrating the concept of polymorphic selections.

**Example 7.** Consider the set of blocks \( B^C = \{ C_1, C_2 \} \) in entity-set \( C \) from Table 4.2, predicate
φ_i : C_c.ratings = 4.7, and combine function Φ_c.ratings = AVG. Block C_1 satisfies η_{φ_i} since there exists a potential object θ^{C_1'} \in Θ^{C_1} that satisfies σ_{φ_i}. Note that θ^{C_1'} resulted from merging subset C_1' = \{c_1, c_3\}, where C_1' \in \mathcal{P}(C_1). The ratings value of θ^{C_1'} is 4.7 (i.e., avg(4.6, 4.8) = 4.7). In contrast, block C_2 does not satisfy η_{φ_i} since no potential object θ^{C_2'} \in Θ^{C_2} exists, which satisfies σ_{φ_i}. As a result, η_{φ_i}(B^C) = C_1.

A polymorphic join is a binary operation, denoted by ◙ ◦ φ_k, where φ_k is the propositional predicate. Since ◙ ◦ φ_k has two inputs, four cases exist. Case one is when both inputs to this operator are sets of clean objects, say O^R and O^S, then it acts as the regular join operator ▷◁ φ_k but applied to objects instead of tuples. That is, it joins all object pairs in O^R and O^S that satisfy φ_k. Case two occurs when the left input is dirty data (i.e., a set of blocks, say B^R). The left polymorphic join operator, denoted by ◙ ◦ φ_k, joins all those block-object pairs for which φ_k holds as defined next:

**Definition 2.** Block R_i \in B^R and object o^S_j \in O^S satisfy ◙ ◦ φ_k if there exists a potential object θ^{R_i'} \in Θ^{R_i} that joins with o^S_j to form an object θ^{R_i'}o^S_j that satisfies ▷◁ φ_k.

In other words, ◙ ◦ φ_k would associate those blocks R_i \in B^R that contain at least one potential object θ^{R_i'} with object o^S_j such that θ^{R_i'} joins o^S_j to satisfy ▷◁ φ_k.

Similarly, case three happens if the right input is a set of dirty blocks (say B^S). The right polymorphic join operator, denoted by ◙ ◦ φ_k, joins all those object-block pairs for which φ_k holds as defined next:

**Definition 3.** Object o^R_j \in O^R and block S_i \in B^S satisfy ◙ ◦ φ_k if there exists a potential object θ^{S_i'} \in Θ^{S_i} that joins with o^R_j to form an object o^R_jθ^{S_i'} that satisfies ▷◁ φ_k.

Case four takes place when both inputs are sets of dirty blocks, then the polymorphic join operator ◙ ◦ φ_k joins all those block-block pairs for which φ_k holds as defined next:
Definition 4. Blocks \( R_i \in B^R \) and \( S_j \in B^S \) satisfy ▶◁\( \varphi_k \) if there exists a potential object \( \theta^{R_i} \in \Theta^{R_i} \), a potential object \( \theta^{S_j} \in \Theta^{S_j} \), and the joined object \( \theta^{R_i} \theta^{S_j} \) satisfies ▷◁\( \varphi_k \).

Intuitively, ▶◁\( \varphi_k \) would associate those blocks \( R_i \in B^R \) that contain at least one potential object \( \theta^{R_i} \) with those blocks \( S_j \in B^S \) that contain at least one potential object \( \theta^{S_j} \) such that \( \theta^{R_i} \) joins \( \theta^{S_j} \) to satisfy ▷◁\( \varphi_k \). The next example shows the concept of polymorphic joins.

Example 8. Consider the set of blocks \( B^C = \{C_1, C_2\} \) in entity-set \( C \) and the set of blocks \( B^M = \{M_1, M_2\} \) in entity-set \( M \) from Tables 4.2 and 4.3, predicate \( \varphi_j : C.m_id = M.m_id \), and combine function \( \Phi_{m_id} = \text{UNION} \). Blocks \( C_1 \) and \( M_1 \) satisfy ▶◁\( \varphi_j \) since potential object \( \theta^{C_1} \) that results from merging subset \( C_1' = \{c_1, c_2, c_4\} \) joins with potential object \( \theta^{M_1} \) that results from merging subset \( M_1' = \{m_1, m_2, m_4\} \). Note that \( C_1' \in \mathcal{P}(C_1) \) and \( M_1' \in \mathcal{P}(M_1) \). Also, note that the value of the \( m_id \) attribute in \( \theta^{C_1} \) is \( \{m_1, m_2, m_4\} \) and the value of the \( m_id \) attribute in \( \theta^{M_1} \) is \( \{m_1, m_2, m_4\} \).

Finally, a cleaning operator \( \delta \) deduplicates the entities in the blocks using an ER algorithm, e.g., [21, 31]. This operator along with the relational and polymorphic operators are summarized in Table 4.6.

4.5.2 Equivalent Polymorphic Query Trees

As discussed in Section 4.4.3, executing the plan shown in Figure 4.4 will return incorrect results because of the dirtiness in the data. To overcome this problem, a traditional solution will clean data first, which is semantically equivalent to inserting the deduplication operator \( \delta \) directly above the tree leaves, see Figure 4.5. However, such an approach could be expensive as it might perform unnecessary cleaning and hence increase query execution time.

To return a correct result while trying to avoid unnecessary cleaning, we will augment the plan presented in Figure 4.4 with appropriate polymorphic operators. Clearly, many ways
<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Operator Notation</th>
<th>Operator First Input</th>
<th>Operator Second Input</th>
<th>Operator Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>( \sigma )</td>
<td>Set of tuples</td>
<td></td>
<td>Set of tuples</td>
</tr>
<tr>
<td>Projection</td>
<td>( \pi )</td>
<td>Set of tuples</td>
<td></td>
<td>Set of tuples</td>
</tr>
<tr>
<td>Join</td>
<td>( \Join )</td>
<td>Set of tuples</td>
<td>Set of tuples</td>
<td>Set of tuple-tuple pairs</td>
</tr>
<tr>
<td>Deduplication</td>
<td>( \delta )</td>
<td>Set of blocks</td>
<td></td>
<td>Set of objects</td>
</tr>
<tr>
<td>Polymorphic Selection</td>
<td>( \varsigma )</td>
<td>Set of blocks or set of objects</td>
<td></td>
<td>Set of blocks or set of objects</td>
</tr>
<tr>
<td>Left Polymorphic Join</td>
<td>( \LeftJoin )</td>
<td>Set of blocks or set of objects</td>
<td>Set of objects</td>
<td>Set of block-object pairs or set of object-object pairs</td>
</tr>
<tr>
<td>Right Polymorphic Join</td>
<td>( \RightJoin )</td>
<td>Set of objects</td>
<td>Set of blocks or set of objects</td>
<td>Set of object-block pairs or set of object-object pairs</td>
</tr>
<tr>
<td>Polymorphic Join</td>
<td>( \Join )</td>
<td>Set of blocks or set of objects</td>
<td>Set of blocks or set of objects</td>
<td>Set of block-block pairs or set of block-object pairs or set of object-block pairs or set of object-object pairs</td>
</tr>
</tbody>
</table>

Table 4.6: Relational and Polymorphic Relational Algebra Operators

may exist in which we can add the polymorphic operators to such a plan. For instance, one plan may decide to perform polymorphic selections only and then clean all dirty blocks that pass such polymorphic selections, see Figure 4.6. The intuition for such a plan is that the polymorphic selections will be able to filter away some blocks without cleaning them. However, this strategy misses the opportunity to further prune unnecessary blocks which would have been pruned had we also considered the join predicate(s).

Another plan may choose to clean some entity-sets immediately while deferring the cleaning of other entity-sets to a later time. Figure 4.7 shows an example of such a plan where blocks from entity-set \( M \) are cleaned eagerly while the cleaning of blocks from entity-set \( C \) is deferred to a later time.
A different lazy plan may delay the cleansing of all dirty entity-sets to the end by inserting the cleaning operator $\delta$ above all selections and joins. Note that, in such a scenario, all selection and join operators that are originally applied on the dirty entity-set must be replaced with the corresponding polymorphic selections and joins, see Figure 4.8.

In general, we formally define the equivalence of polymorphic query trees as:

**Definition 5.** Two polymorphic query plans are said to be *equivalent* if the two plans return the same set of objects as their answer on every dataset $\mathcal{D}$ instance.

In the subsequent sections, we will explain how to use the polymorphic operators to intermix query evaluation with ER to improve cleaning efficiency. In particular, we develop two different solutions called lazy-QuERy (Section 4.6) and adaptive-QuERy (Section 4.7) which utilize the query semantics to reap the benefits of early predicate evaluation while still minimizing unnecessary computation in the form of data cleaning.

### 4.6 Lazy-QuERy Solution

In this solution, we develop a lazy architecture that attempts to delay cleansing of dirty entities as much as possible. The main idea in this approach is to try to avoid cleaning until it is necessary for the system to proceed. This architecture relies on the concepts of polymorphic operators and sketches to avoid unnecessary cleaning.

Conceptually, the lazy-QuERy solution can be viewed as consisting of the following steps:

1. *Create Blocks.* The approach starts, as most traditional ER approaches would; by partitioning entities of the same entity-set into (possibly overlapping) smaller blocks. It applies blocking procedures on one (or more) attribute(s), called blocking key(s), for
this purpose. For example, in Table 4.2, we chose the attribute $c\_name$ as our blocking key and we employed one blocking procedure on it to divide the cellphones into blocks based on the first two letters of their names.

2. *Create Sketches.* In this step, the approach creates a sketch for each block to summarize its content. Sketches allow the polymorphic operators to perform an inexact test to decide whether a block satisfies a predicate or not without cleaning the block. Note that the sketches of each entity-set will be maintained in a separate LIFO stack. We denote the LIFO stack for entity-set $E_i$ as $L^{E_i}$ (Section 4.6.1).

3. *Query Plan Execution.* The approach evaluates the query tree using polymorphic operators’ implementations based on the sketches. A block whose sketch does not satisfy the predicate will be discarded. A block whose sketch reaches the topmost operator (viz., passes all the predicates) will need to be cleaned using a cleaning algorithm. The output of cleaning (i.e., clean objects) will be pushed back into the query tree to be evaluated. The algorithm terminates when there are no more sketches/objects to be tested (Section 4.6.2).

### 4.6.1 Creating Sketches

Recall that a block $R_i$ satisfies a polymorphic operator if there exists a potential object $\theta^{R_i'}$, which resulted from merging entities in subset $R_i' \in \mathcal{P}(R_i)$, that satisfies the common relational algebra operator. However, constructing the *power set* of block $R_i$ in order to enumerate all potential objects inside $R_i$ is exponential\(^{10}\) and hence impractical.

 Consequently, the challenge translates into developing a good technique that allows polymorphic operators to perform their tests faster than the naive approach (viz., power set construction). The idea hinges on representing the values of the potential objects inside a

\(^{10}\)The complexity of constructing the power set of block $R_i$ is $O(2^{|R_i|})$. 
block efficiently without constructing them.

A *sketch* is a concise representation of block contents. The key intuition behind sketches is to provide the polymorphic operators with the ability to quickly check if a block satisfies a predicate or not without cleaning the block. This test is considered a *safe* approximation: while it may return false positives, it never returns false negatives. That is, the polymorphic operator returns *false* only when all potential objects inside block $R_i$ are guaranteed not to be a part of the query answer. In contrast, it returns *true* when at least one potential object inside $R_i$ *might* be part of the answer. In our solution, we create a sketch $K_i^R$ for each block $R_i$.

Before we formally define sketch $K_i^R$, we need to introduce some auxiliary notation. Given the set of predicates $\Phi$ in query $Q$, let $P^{R_i} = \{R_i.a_1, \ldots, R_i.a_m\}$ be the set of attributes in $R_i$ that are used in $\Phi$, where $P^{R_i} \subseteq A^{R_i}$. Also, let $V_j^{R_i} = \{\nu_{1j}, \ldots, \nu_{mj}\}$ represent the set of all values in block $R_i$ for attribute $R.a_j$.

Sketch $K_i^R$ is defined as $K_i^R = \{K_1^R, \ldots, K_m^R\}$, where $K_j^R$ is a signature for attribute $R_i.a_j$. The value of signature $K_j^R$ represents the compact representation of the values of the $R_i.a_j$ attribute. The type of $K_j^R$ depends on the type of attribute $R.a_j$.

Our approach computes signatures as follows. If $R.a_j$ is a numeric attribute, then $K_j^R$ is a *range* $[x, y]$. The values of $x$ and $y$ are computed based on the combine function $\oplus_{R.a_j}$ used to merge the values in $R.a_j$. That is, if $\oplus_{R.a_j}$ is:

1. **ADD**, then $x = \min(V_j^{R_i})$ and $y = \sum_{\ell=1}^m \nu_{\ell j}$.
2. **MAX**, **MIN**, or **AVG**, then $x = \min(V_j^{R_i})$ and $y = \max(V_j^{R_i})$.

If $R.a_j$ is an identifier, a reference, or a categorical attribute, then the type of $K_j^R$ is a *set* $U$. Set $U$ is computed as follows: $U = \bigcup_{\ell=1}^m \text{hash}(\nu_{\ell j})$, where $\text{hash()}$ is a function that computes the hash value of $\nu_{\ell j}$.
The following example describes the concept of sketches.

**Example 9.** Consider entity-set $C$, Query 1, and combine functions: $\oplus_{source} = \text{UNION}$, $\oplus_{c\_name} = \text{UNION}$, $\oplus_{m\_id} = \text{UNION}$, and $\oplus_{c\_reviews} = \text{MIN}$. We create sketch $\mathcal{K}_1^C$ for block $C_1$ in Table 4.2 as follows:

<table>
<thead>
<tr>
<th>source</th>
<th>c_name</th>
<th>m_id</th>
<th>c_reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(BB),</td>
<td>hash(iPhone 6),</td>
<td>hash($m_1$),</td>
<td>[220,550]</td>
</tr>
<tr>
<td>hash(WM)</td>
<td>hash(iPhone 5),</td>
<td>hash($m_2$),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hash(iPhone-VI)</td>
<td>hash($m_4$)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Sketch $\mathcal{K}_1^C$ for Block $C_1$ in Table 4.2

### 4.6.2 Query Plan Execution

The lazy-**QuERy** solution exploits the concept of “Equivalent Query Trees” to evaluate the query plan using polymorphic operators’ implementations based on the blocks’ sketches. It assumes the query tree that contains the best operator placement and join ordering is given. For example, the input to the lazy solution will be a plan similar to the “Query 1 Plan” in Figure 4.4. Given such a plan, the lazy solution substitutes each algebraic operator in the query tree, which is defined on a dirty entity-set, with its corresponding polymorphic operator. It then places the cleaning operator above all polymorphic selections and joins. As a result, a plan similar to the “Lazy Plan” in Figure 4.8 will be the plan that the lazy solution will create to execute the “Query 1 Plan” in Figure 4.4 on dirty data. The “Lazy Plan” is called lazy as it tries to delay the cleansing of dirty entities as much as possible. Its key insight is to try to avoid cleaning until it is necessary for the system to proceed.

**QuERy** employs the well-known database pipelining architecture\textsuperscript{11}. In pipelining, the output

\textsuperscript{11}Most current DBMSs (e.g., MySQL, PostgreSQL, etc.) use, when possible, pipelining to execute the query plan.
of one operator is passed to its parent without materializing the intermediate results. To support pipelining, the operators should implement the * Iterator Interface*, i.e., they implement three functions:

1. The **open()** function, which initializes the iteration.
2. The **getNext()** function, which calls operator-specific code to perform the operator’s task. It also calls the **getNext()** function on each child operator to pull up the next item.
3. The **close()** function, which ends the iteration when all output items have been produced through repeated calls to **getNext()**.

In **QuERy**, all polymorphic operators are implemented as *iterators*. An item (denoted as $I$) in **QuERy** could be a block sketch, a clean object, or a composite item that result from joining two (or more) items. We refer to items that belong to a composite item as subitems. Note that, in **QuERy**, all query plans are either left (or right) deep plans since **QuERy** does not materialize the intermediate results.

The **Execute-Plan(.)** function, shown in Figure 4.10, begins when the topmost operator
(viz., tree root\textsuperscript{12}) calls \texttt{open()} to initialize the state of the iterator (Line 2). Then, it issues repeated calls to \texttt{getNext()} to pull up the next items (Lines 3–4). Each \texttt{getNext()} call performs two tasks: (i) it calls \texttt{getNext()} on each child operator and (ii) performs a specific task based on the operator type. If the operator is \textit{select}, then it filters away items that do not satisfy the selection predicate. If the operator is \textit{join}, then it combines items (from an entity-set) with other items (from another entity-set) that do satisfy the join predicate to form a composite item. Note that a block whose sketch does not satisfy all predicates in the tree will be discarded.

Recall that, in this lazy solution, we place the cleaning operator above all polymorphic selections and joins and thus, an item will not reach the deduplicate operator unless it passes all predicates in the query tree. Such an item is, in fact, a composite item as it resulted from joining two (or more) items. If all subitems in this composite item are objects, then this composite item satisfies the query and hence, is added to the answer (Lines 5–6). However, if at least one subitem in this item is a sketch, then a \textit{block-picking policy}\textsuperscript{13} will choose a sketch from this item to clean its corresponding block (Lines 7–8). Note that this approach treats the cleaning strategy as a “black-box” and hence, any cleaning strategy (e.g. [21,46], etc.) will suffice.

The output of cleaning (viz., clean objects) are pushed back into their appropriate stack (Line 9). It is important to note that the choice of a LIFO stack is critical since it provides higher priority to the clean objects compared to dirty blocks. Interestingly, this choice should intuitively reduce cleaning since the probability that an object will satisfy a predicate is usually much smaller when compared to the probability that a sketch will satisfy that exact predicate.

The algorithm terminates the iteration by calling \texttt{close()} when all items in the stacks are

\textsuperscript{12}In our discussion, we ignore projections and hence, we assume that the deduplicate operator is the tree root.

\textsuperscript{13}Recall that, each sketch corresponds to a block and hence picking a sketch is, in fact, picking a block.

94
execute-plan(root, L^E_1, \ldots, L^E_n)
1   Ans←{}
2   root.open()
3   I←root.getNext()
4   while I≠null
5     if \forall i∈I,isObject(i) then //all subitems in I are objects
6       Ans←Ans∪I
7     else //at least one item is a sketch
8       O^E_j←Pick-And-Clean-Block(I)
9       L^E_j.push(O^E_j)
10      I←root.getNext()
11     root.close()
12 return Ans

Figure 4.10: Execute-Plan(.) Function

consumed (Line 11).

4.6.3 Discussion

In this section, we briefly explain a few interesting points that are related to blocking.

Multiple blocking functions. ER techniques typically use several blocking functions to ensure that all the likely matching entities are compared, improving the quality of the result.

Our solution deals with such a case by creating a separate set of LIFO stacks for each blocking function. Next, it considers each set of stacks independently.

Blocking key sketch optimization. An important (and frequent) special case takes place when there is a predicate defined on an attribute that was selected as a blocking key to partition the entity-set. In this case, the value of the signature for this attribute is chosen to be equal to the value returned by the blocking procedure (such a value is denoted by BKV).

For instance, the second join predicates in Query 1 is ϕ_1 : C_x.c_name = C_y.c_name and entity-set C in Table 4.2 is divided based the c_name attribute. Thus, the value of signature K^{C_1}_{c_name} is “iP”. Note that this special case allows for even more efficient block-to-block join.
processing compared to using a union sketch, as we present in Section 4.8.

**Block-picking policy.** A block-picking policy selects one block from a composite item (that reached the deduplicate operator) to clean it. Intuitively, such a policy should choose a block that may reduce the query execution time. Many different policies can be used. We classify such policies into three main classes. The first class picks a block based on its location in the composite item. For example, a policy from this class may pick and clean the leftmost (or rightmost) available block first. The second class chooses a block to clean based on its size. For instance, it selects the smallest block available in the composite item. The third class picks a block to clean based on the selectivities of the predicates in the query (see Section 4.7). We have experimented with different policies from each class. The one that has demonstrated the best results is based on picking the rightmost available block from the composite item first.

### 4.7 Adaptive-QuERy Solution

The previous solution is considered lazy since it tries to delay the cleaning of dirty entities as much as possible. While such an approach will reduce the cost of cleaning, it might increase the cost of processing the query. For example, assume that all sketches end up reaching the deduplicate operator (viz., the topmost operator in Figure 4.8), meaning that their corresponding blocks need to be cleaned.

In this case, the time spent in trying to filter away these blocks is wasted. In fact, cleaning these blocks *eagerly* (without passing their sketches up the tree) might be more efficient.

To address this issue, we implement a different solution which is an *adaptive* cost-based approach that, given a query tree (with polymorphic operators) and dirty entity-sets, can devise a good plan to simultaneously clean and process the query. The key intuition hinges on
placing decision nodes as the bottommost nodes in the query tree, as presented in Figure 4.9. The task of such decision nodes is to decide if eagerly cleaning some dirty blocks is more efficient (in terms of the overall query execution time) than delaying their cleansing until the last stage as in the lazy solution.

Our adaptive cost-based solution consists of two steps. In the first step, we use a sampling technique to collect different statistics (e.g., selectivities of predicates, cost of join, etc.). In the second step, the decision nodes utilize these statistics to make smart decisions, i.e., clean a block eagerly versus passing it up the tree.

### 4.7.1 Sampling Phase

The first step in our adaptive solution is to employ a sampling phase to collect various statistics such as predicates’ selectivities, cost of cleaning, cost of block-to-block join test, etc. The basic idea is to use two fractions $f_1$ and $f_2$, where $0 < f_1, f_2 < 1$, to control how many input blocks will be consumed (either cleaned or passed up the tree for evaluation) during this phase. In particular, the sampling phase begins by cleaning $f_1 \times |B_{\text{total}}|$ blocks, where $|B_{\text{total}}|$ is the total number of blocks in all entity-sets in dataset $D$. This early cleaning allows our adaptive-QuERy solution to estimate the following values:

1. The average cost of cleaning a block, given its size, denoted by $\text{clean}(R_i)^{14}$.
2. The average cost of joining a clean object with a clean object, denoted by $\text{cost}(OOjoin)$.
3. The clean object selectivity of each predicate $\varphi_i$ in the query tree, denoted by $\text{SO}(\varphi_i)$.

Subsequently, fraction $f_2$ is utilized (by our solution) to ensure that at least $f_2 \times |B_{\text{total}}|$ blocks will be passed up the tree for evaluation. This block-evaluation authorizes our solution to estimate

---

14The cost of cleaning a block is algorithm-dependent. That is, it varies based on what cleaning algorithm (e.g., Swoosh [21], SN [46], etc.) was used to clean it.
estimate the next set of values:

1. The average cost of joining a block sketch with a block sketch, denoted by $cost(BBjoin)$. 
2. The average cost of joining a block sketch with an object, denoted by $cost(BOjoin)$. 
3. The dirty block selectivity of each predicate $\varphi_i$ in the query tree, denoted by $S_B(\varphi_i)$. 

If predicate $\varphi_i$ is associated with a select operator, then we calculate $S_B(\varphi_i)$ as follows:

$$S_B(\varphi_i) = \frac{|B_s(\varphi_i)|}{|B_t(\varphi_i)|}$$

where $|B_s(\varphi_i)|$ is the number of blocks that satisfy predicate $\varphi_i$ and $|B_t(\varphi_i)|$ is the block count that was tested by predicate $\varphi_i$. In addition, we compute $S_O(\varphi_i)$ as follows:

$$S_O(\varphi_i) = \frac{|O_s(\varphi_i)|}{|O_t(\varphi_i)|}$$

where $|O_s(\varphi_i)|$ is the number of clean objects that satisfy predicate $\varphi_i$ and $|O_t(\varphi_i)|$ is the clean object count that was tested by predicate $\varphi_i$.

Yet, if predicate $\varphi_i$ is associated with a join operator, then we estimate $S_B(\varphi_i)$ as follows:

$$S_B(\varphi_i) = \frac{|B_s(\varphi_i)|}{|B_{t_1}(\varphi_i)| \times |B_{t_2}(\varphi_i)|}$$

where $|B_{t_1}(\varphi_i)|$ is the cardinality of the first input set of blocks and $|B_{t_2}(\varphi_i)|$ is the cardinality of the second input set of blocks of the join operator. Moreover, we calculate $S_O(\varphi_i)$ as follows:

$$S_O(\varphi_i) = \frac{|O_s(\varphi_i)|}{|O_{t_1}(\varphi_i)| \times |O_{t_2}(\varphi_i)|}$$

where $|O_{t_1}(\varphi_i)|$ is the cardinality of the first input set of clean objects and $|O_{t_2}(\varphi_i)|$ is the cardinality of the second input set of clean objects of the join operator.
During this sampling phase, we also calculate an important statistic: the probability of a given block $R_i$ passing all predicates in the query tree and hence, reaching the deduplicate operator\(^{15}\). This probability, denoted by $\mathbb{P}(R_i)$, is computed as follows:

$$
\mathbb{P}(R_i) = \frac{|B_{top}^R|}{|B_{read}^R|}
$$

where $|B_{top}^R|$ is the number of blocks from entity-set $R$ that reached the top operator and $|B_{read}^R|$ is the number of blocks from $R$ that were read during the sampling phase.

Intuitively, to better estimate the previous values, we would like to read a comparable number of blocks from each entity-set in $\mathcal{D}$. For instance, a join algorithm such as *tuple-based nested-loop*, where all tuples of the inner entity-set are read before reading the second tuple from the outer entity-set, will not be appropriate. Therefore, we implement the *ripple join* algorithm [41] as our join algorithm. This join algorithm aims to draw tuples (i.e., items) from entity-sets at the same rate and hence, should allow for better estimations.

### 4.7.2 Adaptive Cost-based Cleaning

Based on the statistics computed in the sampling phase, the decision nodes in our adaptive-QuERy solution use the decision plane, presented in Table 4.8, to compare the cost of cleaning block $R_i$ eagerly ($\text{CleanNow}(R_i)$) versus the cost of evaluating $R_i$’s sketch in the query tree ($\text{Evaluate}(\mathcal{K}_i^R)$).

The decision whether to $\text{CleanNow}(R_i)$ or $\text{Evaluate}(\mathcal{K}_i^R)$ relies on the following four values:

1. The probability of block $R_i$ reaching the deduplicate operator, denoted by $\mathbb{P}(R_i)$.
2. The cost of cleaning block $R_i$, denoted by $\text{clean}(R_i)$.

\(^{15}\)Recall that, similar to the lazy solution, this solution places the cleaning operator above all polymorphic selections and joins.
3. The cost of evaluating block $R_i$’s sketch ($\mathcal{K}_i^R$) in the query tree, denoted by $\text{tree}(\mathcal{K}_i^R)$. We compute $\text{tree}(\mathcal{K}_i^R)$ as:

$$\text{tree}(\mathcal{K}_i^R) = \mathcal{T}^B \times \text{cost}(BB\text{join}) + \mathcal{T}^O \times \text{cost}(BO\text{join})$$

where $\mathcal{T}^B$ is the estimated number of block-to-block (block-to-object) join tests that will be performed to check if sketch $\mathcal{K}_i^R$ will reach the top operator. Similarly, $\mathcal{T}^O$ is the estimated number of block-to-object join tests that will be performed to test if sketch $\mathcal{K}_i^R$ will reach the deduplicate operator. Function \textit{Estimate-Number-Of-Join-Tests}(.) presented in Figure 4.11 show how to estimate $\mathcal{T}^B$ and $\mathcal{T}^O$. Note that, in this function, we assume that the query plan is similar to the plan presented in Figure 4.9. That is, the ID of the predicate associated with the root operator is 0 and there is a selection predicate above all leaf nodes\(^{16}\).

4. The cost of evaluating $R_i$’s cleaned objects ($\mathcal{O}_i^R$) in the tree, denoted by $\text{tree}(\mathcal{O}_i^R)$. We compute $\text{tree}(\mathcal{O}_i^R)$ as:

$$\text{tree}(\mathcal{O}_i^R) = |R_i|(\mathcal{T}^B \times \text{cost}(BO\text{join}) + \mathcal{T}^O \times \text{cost}(OO\text{join}))$$

where $|R_i|$ is the size of block $R_i$.

For instance, if $\mathbb{P}(R_i)$ is high and both $\mathbb{clean}(R_i)$ and $\text{tree}(\mathcal{O}_i^R)$ are low, then it might be

\(^{16}\)The existence of the selection predicates is only necessary for the numbering of the predicates. If the query does not contain such a selection predicate then we add a dummy one that passes all items up the tree.
Estimate-Number-Of-Join-Tests(Φ, pid)
1 \( T^B \leftarrow 1 \)  //number of block-to-block join tests
2 \( T^O \leftarrow 1 \)  //number of block-to-object join tests
3 firstLoop \leftarrow true
4 while true
5 \hspace{1em} \text{if } pid \% 2 \neq 0 \hspace{1em} //left bottommost selection only
6 \hspace{1em} \phi_{\ell} \leftarrow \Phi.get(pid + 1)
7 \hspace{1em} pid \leftarrow pid + 1
8 \hspace{1em} \text{else} \hspace{1em} //all other selections
9 \hspace{1em} \phi_{\ell} \leftarrow \Phi.get(pid - 1)
10 \hspace{1em} \text{if } firstLoop = true \hspace{1em} then
11 \hspace{2em} T^B \leftarrow |B_t(\phi_{\ell})| \times S_B(\phi_{\ell})
12 \hspace{2em} T^O \leftarrow |O_t(\phi_{\ell})| \times S_O(\phi_{\ell})
13 \hspace{1em} firstLoop \leftarrow false
14 \hspace{1em} \text{else}
15 \hspace{2em} \phi_r \leftarrow \Phi.get(pid)
16 \hspace{2em} T^B \leftarrow T^B \times S_B(\phi_{\ell})
17 \hspace{2em} T^B \leftarrow T^B \times |B_t(\phi_r)| \times S_B(\phi_r)
18 \hspace{2em} T^O \leftarrow T^O \times S_O(\phi_{\ell})
19 \hspace{2em} T^O \leftarrow T^O \times |O_t(\phi_r)| \times S_O(\phi_r)
20 \hspace{1em} pid \leftarrow \text{pid} \div 2
21 \hspace{1em} \text{if } pid < 2 \hspace{1em} then
22 \hspace{2em} \text{break}
23 \hspace{1em} \text{return } T^B, T^O

Figure 4.11: Estimate-Number-Of-Join-Tests(.) Function

more tempting to clean \( R_i \) immediately instead of deferring its cleaning with little hope of it being discarded.

The Make-A-Decision(.) function, shown in Figure 4.12, presents the method a decision node employs to make its choice using the decision plane. It starts by computing the cost of CleanNow(\( R_i \)), which corresponds to the cost of cleaning block \( R_i \) plus the cost of evaluating the clean objects which resulted from cleaning \( R_i \) in the query tree (Line 1). It also computes the cost of Evaluate(\( \mathcal{K}_R \)), which corresponds to computing the cost of two cases (Line 2). On the one hand, if the sketch passes all predicates and thus reaches the deduplicate operator, then the cost of evaluating it in the query tree is equal to the summation of the following three costs (i) the cost of evaluating this sketch in the query tree, (ii) the cost of cleaning
MAKE-A-DECISION($R_i$)
1 \text{CleanNow} (R_i) \leftarrow \text{clean} (R_i) + \text{tree} (O^{R_i})
2 \text{Evaluate} (K_i^R) \leftarrow P(R_i) \times (\text{clean} (R_i) + \text{tree} (O^{R_i}) + \text{tree} (K_i^R)) + (1 - P(R_i)) \times \text{tree} (K_i^R)
3 \text{if CleanNow} (R_i) < \text{Evaluate} (K_i^R) \text{ then}
4 \ O^{R_i} \leftarrow \text{CLEAN-BLOCK} (R_i)
5 \text{else}
6 \ Evaluate (K_i^R)

Figure 4.12: MAKE-A-DECISION(.) Function

the block that corresponds to this block, and (iii) the cost of testing the clean objects that resulted from cleaning its block. On the other hand, if the sketch is eliminated by one predicate in the query tree, then its evaluation cost is only the cost of testing it in the query tree. Finally, the algorithm picks the decision with the least amount of cost (Lines 3–6).

4.8 Experimental Evaluation

In this section, we empirically evaluate the efficiency of our proposed approaches on real and synthetic datasets.

4.8.1 Experimental Setup

Solutions. In our experiments, we compare the following three solutions:

1. The standard solution (abbreviated SS), presented in Section 4.4.3.
2. The lazy-QuERy solution (abbreviated LQS), explained in Section 4.6.
3. The adaptive-QuERy solution (abbreviated AQS), described in Section 4.7.

Cleaning Algorithm. To deduplicate a dirty block $R_k \in B^R$, we resolve any two entities $r_i, r_j \in R_k$ to try to decide whether they are duplicates or not as in [56]. We next cluster the
duplicate pairs using the well-known transitive closure algorithm [43].

As mentioned in Section 4.7, the cost of cleaning block \( R_k \) is algorithm-dependent. Thus, for this algorithm, we compute this cost as:

\[
\text{clean}(R_k) = \text{cost(resolve)} \times \frac{|R_k| \times (|R_k| - 1)}{2}
\]

where \( \text{cost(resolve)} \) is the average cost of resolving two entities in block \( R_k \) and \( |R_k| \) is its size.

### 4.8.2 Performance Factors

We can split the end-to-end execution time of SS, LQS, and AQS into two main parts: (i) the time spent at cleaning the dirty blocks and (ii) the time spent at processing the input items (viz., blocks and object). As we discuss next, several factors are expected to affect the performance of the three solutions. In this section, we summarize the expected effects of each factor prior to testing and validating their effects experimentally in the following sections.

**Block selectivity** \((S_B)\). SS is not affected by the block selectivity because it does not deal with blocks. In contrast, LQS is affected in the following fashion. When \( S_B \) is high, a small number of blocks will satisfy the query and thus, the time spent at evaluating the query and at cleaning the dirty blocks will decrease. However, when it is low, a larger number of blocks will satisfy the query. In general, the processing of the dirty blocks will be mostly overhead as most blocks will not be discarded. Therefore, the time spent at both evaluating the query and at cleaning the dirty blocks will increase. In addition, it affects AQS in the following way. When \( S_B \) is high, AQS’s performance will be similar to LQS. However, when it is low,
the performance of AQS will usually be similar to SS.

Object selectivity ($S_O$). Object selectivity affects all three solutions similarly. When it is high, a small number of objects will satisfy the query and thus, the time to process the query will decrease. However, when $S_O$ is low, a larger number of objects will satisfy the query, and thus the time to evaluate the query will increase.

Cleaning Cost ($cost(resolve)$). All three solutions are affected by this factor. Clearly, when the cost of cleaning is high, the time spent at cleaning the dirty blocks will be more than the time spent at cleaning them when it is low. Note that SS will suffer the most (compared to LQS and AQS) because it will always clean all blocks.

BBJoin Cost ($cost(BBJoin)$) / BOJoin Cost ($cost(BOJoin)$). SS is not affected by these two costs since it does not deal with blocks. In contrast, LQS is affected in the following way. When the costs of BBJoin and BOJoin are low, the overhead of evaluating the blocks will be low, and vice versa. Moreover, they affect AQS in the following fashion. When $cost(BBJoin)$ and $cost(BOJoin)$ are low, AQS’s performance will often mimic LQS. However, when they are low, its performance will usually imitate SS.

OOJoin Cost ($cost(OOJoin)$). This cost affects all three solutions similarly. When $cost(OOJoin)$ is high, the time spent in processing the query will increase. However, when it is low, the time spent in evaluating the query will decrease.

Overall, LQS is expected to perform excellently when $S_S$ is high and/or $cost(BBJoin)$ and $cost(BOJoin)$ are low. We expect LQS to perform reasonably (still better than SS) when the costs of BBJoin and BOJoin tests are lower than the cleaning cost (of blocks that do not satisfy the query) plus the cost of evaluating the objects inside them.

Moreover, AQS is expected to perform competently when LQS performs well. In addition, it
is expected to overcome the difficulties that LQS may face when the costs of BBJoin and BOJoin tests are higher than the cleaning cost (of blocks that do not satisfy the query) plus \(\text{cost}(OOJoin)\) for the the objects inside these blocks.

### 4.8.3 Products Dataset Experiments

In this section, we evaluate the efficacy of our approaches on a real electronic products dataset collected from two different data sources: Best Buy [3] and Walmart [4]. We obtained a subset of 89,070 raw records from these sources. To enrich our data further, we collected 1,237 raw records from Wikipedia [6] regarding the product manufacturers.

Using basic data wrangling techniques, we restructured these raw records into dataset \(\mathcal{D} = \{C, M\}\). Entity-set \(C\) contains entities that describe electronic products and entity-set \(M\) holds entities that describe their manufacturers. The dataset schema is similar to that of Tables 4.2 and 4.3. The cardinalities of entity-sets are \(|C| = 89,070\) and \(|M| = 2,144\).

Each entity-set is partitioned into a set of blocks. We use a blocking function to split entity-set \(C\) into a set of blocks (viz., \(\mathcal{B}^C\)) based on the first six letters of the products’ names. In addition, we use a blocking function that partitions entity-set \(M\) into a set of blocks (i.e., \(\mathcal{B}^M\)) based on the first two characters of the manufacturers’ names. The sizes of the resulted block-sets are \(|\mathcal{B}^C| = 9,742\) and \(|\mathcal{B}^M| = 292\).

To resolve a pair of electronic products (i.e., entity-set \(C\)), we use the Edit-Distance algorithm to compute the similarity between the products’ names. If this similarity is sufficient, then the resolve function declares the two products to be duplicates. In addition, to resolve entities from entity-set \(M\), we utilize Jaro-Winkler distance to compare the names of the manufacturers.

To merge duplicate pairs from entity-sets \(C\) and \(M\), we select the \(\text{MAX}\) semantics to combine
all numeric attributes and the UNION semantics to combine all non numeric attributes (viz.,
categorical, identifier, or reference attributes).

Finally, we compute sketches for entity-set $C$ as:

$$
\begin{array}{|c|c|c|}
\hline
\text{source} & \text{c._name} & \text{m._id} \\
\hline
\bigcup_{\ell=1}^{m} \text{hash}(\nu_{ij}) & \text{BKV} & \bigcup_{\ell=1}^{m} \text{hash}(\nu_{ij}) \\
\hline
\end{array}
$$

Table 4.9: Sketches for Entity-set $C$

We also compute sketches for entity-set $M$ as:

$$
\begin{array}{|c|c|}
\hline
\text{m._id} & \text{m._country} \\
\hline
\bigcup_{\ell=1}^{m} \text{hash}(\nu_{ij}) & \bigcup_{\ell=1}^{m} \text{hash}(\nu_{ij}) \\
\hline
\end{array}
$$

Table 4.10: Sketches for Entity-set $M$

**Experiment 1 (Lazy-QuERy Solution vs. Standard Solution).** In this experiment, we use a set of queries similar to Query 1 to compare our lazy-QuERy solution ($LQS$) with the standard solution ($SS$) in terms of their end-to-end running time and the number of resolves called. Figure 4.13 plots the actual end-to-end execution time of both solutions for five different countries (viz., “Finland”, “Japan”, “Korea”, “UK”, and “USA”) using two different $c._reviews$ values (viz., $t = 16$ and $t = 128$). Figure 4.14 is similar to Figure 4.13 but plots the number of resolve calls instead of the execution time. Note that the histograms in the two figures are identical, thus demonstrating that calling resolve (and not query evaluation) is the bottleneck of both solutions in this test.

As expected, $LQS$ is both faster and issues fewer resolves than $SS$. This is due to its awareness of the query which gives it the ability to discard blocks (whose sketches do not satisfy one of the query predicates), resulting in savings in resolves as such blocks do not require cleaning.

In Figure 4.13, when $m._country = “Finland”$ and $c._reviews \geq 16$ (or $c._reviews \geq 128$), $LQS$ takes only 4 seconds to evaluate the query and return the answer while $SS$ takes more than
140 seconds to do so. This huge savings is due to the fact that, in our dataset, there is only one manufacturer that is located in “Finland” and hence a small number of blocks will require cleaning. However, there are more manufacturers that are located in “Japan”, for instance, and thus LQS takes more than 20 seconds to answer such a query.

**Experiment 2 (Selectivity Effects).** Figure 4.15 utilizes a set of queries, similar to Query 1, to study the effects of the query selectivity on LQS as well as on SS. To control the query selectivity, we fix the manufacturer’s country (i.e., \( m_{\text{country}} = \text{“USA”} \)) and vary the \( c_{\text{reviews}} \) values (viz., 1, 4, \ldots, 4,096). Note that Figure 4.15 uses a log-lin scale plot and the selection of different \( c_{\text{reviews}} \) values is done carefully to show the entire spectrum of LQS’s behavior.

As in Experiment 1, herein, the cost of calling resolve is also the dominant factor of the overall execution time. Hence, the end-to-end execution time of both solutions depends heavily on the number of blocks that will be cleaned.

The number of resolves invoked by LQS and obviously its query execution time (since resolves are the dominant factor) are relative to the number of blocks that satisfy the query. For instance, the number of blocks that satisfy predicate \( \varphi_i : c_{\text{reviews}} \geq 1 \) is almost equal to the number of blocks that satisfy predicate \( \varphi_j : c_{\text{reviews}} \geq 4 \); and hence, the LQS takes
almost equal time to answer these two queries. In contrast, the block count that satisfies $\varphi_1: c_{\text{reviews}} \geq 1$ is considerably more than the block count that satisfies $\varphi_k: c_{\text{reviews}} \geq 128$ and hence, LQS solution takes considerably less time to answer the second query. In short, in case of LQS, whenever the number of blocks that satisfy the query (and hence require cleaning) increases, the total execution time to answer the query will increase, and vice versa.

Note that the query selectivity effects are not noticed on SS (in this experiment) since the cleaning cost overshadows the cost of query processing.

**Experiment 3 (Execution Time Breakdown).** LQS almost always outperforms SS as shown in Experiments 1 and 2. However, there are some cases in which SS performs better than LQS. Such cases usually occur when (i) the query selectivity is low (viz., most blocks will require cleaning) and/or (ii) cost(BBJoin) tests is high.

To present such a case, we use the following query:

```sql
SELECT * FROM C AS C_x, C AS C_y WHERE C_x.c_name = C_y.c_name AND C_x.c_source = "BB" AND C_x.c_reviews $\geq$ t AND C_y.c_source = "WM" AND C_y.c_reviews $\geq$ t
```

where we choose the value of $t$ to be equal to 0. Note that the selectivity in this query is only offered by the join predicate and hence its selectivity is low. In addition, we experiment with
Table 4.11: Execution Time Breakdown

<table>
<thead>
<tr>
<th></th>
<th>SS</th>
<th>LQS (BKV)</th>
<th>LQS (U₁)</th>
<th>LQS (U₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Time (s)</strong></td>
<td>328.586</td>
<td>221.719</td>
<td>258.023</td>
<td>664.690</td>
</tr>
<tr>
<td><strong>Initialization Time</strong></td>
<td>1.148%</td>
<td>1.715%</td>
<td>1.495%</td>
<td>0.578%</td>
</tr>
<tr>
<td><strong>Resolution Time</strong></td>
<td>40.579%</td>
<td>49.988%</td>
<td>42.519%</td>
<td>16.440%</td>
</tr>
<tr>
<td><strong>BBJoin Time</strong></td>
<td>0%</td>
<td>2.374%</td>
<td>6.00%</td>
<td>63.043%</td>
</tr>
<tr>
<td><strong>BOJoin Time</strong></td>
<td>0%</td>
<td>16.128%</td>
<td>23.331%</td>
<td>9.482%</td>
</tr>
<tr>
<td><strong>OOJoin Time</strong></td>
<td>44.947%</td>
<td>19.222%</td>
<td>17.397%</td>
<td>6.791%</td>
</tr>
<tr>
<td><strong>Iteration Time</strong></td>
<td>13.299%</td>
<td>10.573%</td>
<td>9.258%</td>
<td>3.666%</td>
</tr>
</tbody>
</table>

three different BBJoin tests of various costs. The first test (denoted by BKV) is the least expensive one. It uses the blocking key value (BKV) as the signature value for attribute c.name. It decides if two blocks join or not by comparing their BKVs. Test two (denoted by U₁) is more expensive than test one. The signature of attribute c.name, in this case, is equal to a set of hashed strings. It checks if two blocks join or not by efficiently computing the intersection between two sets of integers. The third test (denoted by U₂) is the most expensive one. In this case, the signature of attribute c.name is a set of strings. It checks if two blocks join or not by exhaustively comparing the string values in the two sets. In short, the cost of testing $U₂ \gg U₁ > BKV$.

To conduct this experiment, we ran each of the four approaches shown in Table 4.11 multiple times and recorded the average total execution time (the first row in Table 4.11). We also computed the breakdown of that execution time which consists of: the times spent at (i) blocking and creating sketches if necessary, (ii) resolving dirty blocks, (iii) joining dirty blocks, (iv) joining blocks with objects, (v) joining clean objects, (vi) and running the iterator interface. Results showed that running LQS using the second block-to-block join test (viz., $U₁$) takes 258.023 seconds. In particular, we show that resolving the dirty blocks takes 42.519% of the total execution time while joining them takes only 6.00%, demonstrating that
resolving blocks is more expensive than joining them in this case.

As shown in Table 4.11, SS outperforms LQS when we use the third BBJoin test (viz., $U_2$). This is because $cost(BBJoin)$ dominates the cost of cleaning and hence, it might be better to clean the blocks eagerly as we show next.

**Experiment 4 (Adaptive-QuERy Solution).** Figure 4.16 studies the performance of our adaptive-QuERy solution (AQS). We, herein, use a set of queries similar to the query presented in Experiment 3. In this experiment, we choose three different values (i.e, 0, 16, and 128) for variable $t$ and we continue to experiment with the three different block-to-block join tests (viz., $BKV$, $U_1$, and $U_2$). In addition, we experimentally set the fraction values to: $f_1 = 5\%$ and $f_2 = 20\%$.

In this experiment we study three interesting cases. The first case occurs when the resolution cost dominates the block-to-block join test cost (i.e., $BKV$ and $U_1$). In such a case, AQS acts similarly to LQS and hence, it always outperforms SS and has almost identical performance to LQS, see Figure 4.16. Case two takes place when the the block-to-block join test cost dominates the resolution cost as in $U_2$ and the query selectivity is not very low (viz., $t = 16$ or $t = 128$). Herein, AQS surpasses both LQS and SS. This is due to the fact that, AQS often makes the correct decision of whether to clean the block eagerly or pass it up for evaluation.
based on the statistics collected in the sampling phase. The third case happens when the block-to-block join test cost dominates the resolution cost (i.e., \( U_2 \)) and the query selectivity is low (viz., \( t = 0 \)). In this case, AQS outperforms LQS since it has the ability to utilize the knowledge that cleaning the block eagerly is more efficient than passing it up the tree. However, it requires some time (i.e., the sampling phase time) to discover this fact, and hence SS outperforms AQS in this extreme case.

Note that, unlike Experiment 2, herein the impact of the query selectivity is noticed on SS since the cleaning cost does not overshadow the cost of query processing query.

### 4.8.4 Synthetic Dataset Experiments

To evaluate our approach in a wider range of various scenarios, we built a synthetic dataset generator that allows us to generate datasets with different characteristics. In each synthetic dataset, we control the following input parameters: (i) \( n \): the number of entity-sets, (ii) \( r \): the number of rows in each entity-set, (iii) \( \ell \): blocking level which leads to the creation of \( b \) non-overlapping blocks with size equal to \( s \), (iv) \( S_B(\varphi_i) \): the block selectivity of predicate \( \varphi_i \), and (v) \( S_O(\varphi_i) \): the object selectivity of predicate \( \varphi_i \).

In addition, we control the costs of \( \text{resolve} \), \( BBJoin \), \( BOJoin \), and \( OOJoin \) to study their impacts more precisely.

**Experiment 5 (Resolves vs. BBJoin Tests).** To study the trade-off between the resolves and block-to-block join tests on LQS, when varying the blocking level value (\( \ell \)) while fixing the other two parameters to: \( n = 2 \) and \( r = 1,000 \); this test employs the following query:

```sql
SELECT * FROM R,S WHERE R.a_i = S.a_i
```

Note that the block/object selectivities do not effect the outcome of this test as we only
have one join predicate and every block/object from entity-set \( R \) joins with exactly one block/object from entity-set \( S \).

When \( \ell = 0 \), which indicates the loosest blocking level, we obtain one block \((b = 1)\) with 1,000 rows in it \((s = 1,000)\). Clearly, in this case, calling resolve is the bottleneck of \textsc{LQS} since it invokes 999,000 resolves versus performing one \textit{BBJoin} test, see Figure 4.17. This explains why \textsc{LQS} takes almost 100 seconds when resolve is expensive \((\text{cost(resolve)} = 0.1 \text{ ms})\) and almost one second when it is cheap \((\text{cost(resolve)} \approx 0 \text{ ms})\). As depicted in Figure 4.17, when \( \ell = 1 \) and \( \ell = 2 \), the number of resolves is comparable to the number of \textit{BBJoin} tests; and hence, the performance of \textsc{LQS} is convergent when resolves are expensive and \textit{BBJoin} tests are cheap and vice versa. Finally, when \( \ell = 3 \) (viz., the tightest blocking level) we obtain \( b = 1,000 \) and \( s = 1 \). \textsc{LQS} invokes 0 calls to resolve versus it performs 500,500 block-to-block join tests. Herein, the cost of \textit{BBJoin} tests is the dominant factor of the overall execution time. This demonstrates why our solution takes almost 1 second when a \textit{BBJoin} test is cheap \((\text{cost(BBjoin)} \approx 0 \text{ ms})\) and almost 52 seconds when it is expensive \((\text{cost(BBjoin)} = 0.1 \text{ ms})\).

\textbf{Experiment 6 (Effects of Object Selectivity).} To evaluate the effects of the object
selectivity on LQS, Figure 4.18 utilizes the following query:

\[
\text{SELECT } * \text{ FROM } R, S \text{ WHERE } R.a_i = S.a_i \text{ AND } R.a_j \geq t \text{ AND } S.a_k \geq t
\]

Note that in the above query, there are three predicates: \( \varphi_1 : R.a_i = S.a_i \), \( \varphi_2 : R.a_j \geq t \), and \( \varphi_3 : S.a_k \geq t \).

In this experiment, we create a dataset \( D = \{R, S\} \) by fixing the input parameters to: \( n = 2 \) and \( r = 100,000 \), and \( \ell = 3 \) (which leads to \( b = 1,000 \) and \( s = 100 \)). Note that, in dataset \( D \), every block/object from \( R \) joins with exactly one block/object from \( S \). We vary the block selectivities \( S_{BR}(\varphi_2) = S_{BS}(\varphi_3) \) from 10\% to 100\% (x-axis). We further set three different values for their object selectivities: \( S_{OR}(\varphi_2) = S_{OS}(\varphi_3) = 10\%, 30\%, \) or \( 50\% \).

In addition, we set the costs of resolve, BBJoin, BOJoin, and OOJoin to be the same. Due to this assignment, the cost of OOJoin overshadows all other costs since the number of OOJoin operations (performed by the algorithm) surpasses all other operations (i.e., resolve, BBJoin, and BOJoin).

As expected, the higher the selectivity of the query (e.g., \( S_{BR}(\varphi_2) = S_{BS}(\varphi_3) = S_{OR}(\varphi_2) = S_{OS}(\varphi_3) = 10\% \)) the faster LQS is able to evaluate the query and return an answer (viz., 1.35 seconds).
4.9 Conclusions

In this chapter, we have studied the problem of analysis-aware data cleaning. We have developed QuERy, a novel architecture for integrating ER with query processing to answer complex SQL-like queries issued on top of dirty data. In particular, we developed two distinct solutions which reap the benefits of evaluating the query predicates to minimize the query execution time. We empirically showed how our approach is significantly better compared to cleaning the entire dataset, especially when the query is very selective.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we have responded to some of the challenges that are facing standard entity resolution techniques, due to the new requirements set by today's modern applications, by developing an analysis-aware approach to entity resolution.

In Chapter 3, we explored analysis-aware data cleaning in the context of selection queries. In particular, we introduced QDA that performs a minimal number of cleaning steps that are necessary to answer a given selection query correctly. We then introduced the concept of vestigiality of certain computations in the context of SQL selection queries. After that, we developed query-driven techniques that leverage the concept of vestigiality to reduce computation. We also demonstrated that QDA is a generic approach that could be applied to different types of clustering techniques. Finally, we provided a comprehensive empirical evaluation of the proposed approach which demonstrates its significant advantage in terms of efficiency over traditional entity resolution techniques.
In Chapter 4, we explored the problem of analysis-aware data cleaning for a large class of flat SQL queries. Specifically, we proposed QuERy. The aim of QuERy is to correctly and efficiently answer complex queries issued on top of dirty data. We then introduced and formalized the notion of polymorphic operators which is a key concept in QuERy. After that, we developed two different solutions: lazy-QuERy and adaptive-QuERy, which reap the benefits of evaluating the query predicates to minimize the query execution time. Finally, we conducted extensive experiments to evaluate the effectiveness of both lazy-QuERy and adaptive-QuERy solutions on real and synthetic datasets.

5.2 Future Work

In this thesis, we have laid out the foundations of two generic analysis-aware entity resolution frameworks, QDA and QuERy, respectively. However, our problems are by no means solved, and there are several interesting directions for future investigation.

Extend QDA to deal with query workloads. Our goal in Chapter 3 was to set up the theoretical underpinning (viz., the concept of vestigiality and optimizing cleaning using different answer semantics) for a query-driven approach for selection queries.

As we have seen, QDA optimizes for a single query at a time. However, it does not optimize for subsequent querying. An interesting future step is to extend our approach so that the cleaning results can be efficiently maintained, and hence the cost of cleaning across query workloads can be amortized.

Embedding QuERy into a DBMS/BDMS. Our goal in Chapter 4 was to set up the academic foundation (i.e., the notion of polymorphic operators, a lazy solution, and an adaptive cost-based solution) for an analysis-aware approach for complex SQL-like queries.
An interesting future direction is to put QuERy into context. In particular, we envision an engineering technique to embed QuERy into a Database Management System (DBMS) and/or a Big Data Management System (BDMS) such as PostgreSQL [7], Spark [8], AsterixDB [12, 13], etc.

To add ER features to such a DBMS/BDMS to provide its end-users with the ability to clean data “on-the-fly” while querying, we need to consider:

1. Extending the query language of the DBMS/BDMS to be able to query dirty data. For instance, we need to add capabilities which allow the end-users to: (i) pinpoint the dirty attributes, (ii) determine the method of cleaning (i.e., which ER algorithm to use), and (iii) select the required answer semantics, etc.

2. We visualize that sketches (see Section 4.6.1) can be stored as tuples/records in the DBMS/BDMS and hence, an extension to the data model of the DBMS/BDMS might be required. Also, an interesting future step is to implement tighter types of sketches for numerical and categorical attributes. For example, instead of compressing the values of a numerical attribute into one interval \([x, y]\), we may compress them into \(k\) values \(\{\nu_1, \ldots, \nu_k\}\). Evidently, such a decision will expose a possible tradeoff analysis between how much time to spend in creating a sketch versus how much pruning we will get from it when it is tight.

3. Updating the query optimizer to be capable of devising good plans to execute a given query over dirty data. Clearly, this modification includes the implementation of the polymorphic operators (see Section 4.5.1) inside such a DBMS/BDMS. Note that such an update must account for issues such as “when to clean?” – eagerly versus lazy, “How much to clean?” that is, full or partial cleaning, etc.

4. Indexing dirty data is an interesting consideration. For instance, we envision that multi-dimensional indexes can be used to index the range values in the sketches.
In addition to the previous considerations, we need to explore issues related to ER such as:

1. The materialization of the ER results to deal with query workloads.
2. The implementation of domain independent cleaning features, that is, different types of resolve and merge functions should be provided to the end-users.

**Analysis-aware ER in the context of streaming data.** Another interesting future direction is that of streaming data in the context of continuous analysis tasks of social media data. Adapting the analysis-aware cleaning techniques to stream data opens a set of new challenges. The semantics of cleaning when performing analysis-aware ER in case of static data (e.g., relations) are well-defined. In contrast, in the context of streaming data, such semantics are not clearly defined which makes the analysis-aware tasks for streaming data very complex.

Intuitively, we would like to specify a cleaning mechanism and define answer equivalence for streaming data, similar to what we proposed in Chapters 3 and 4. However, streaming data is dynamic and cleaning is a data-dependent process where the decision of whether or not two entities co-refer depends not just on the two entities being considered but the entire dataset being cleaned. Thus, the concept of cleaning a data stream, denoted by $S$, is not well-defined (i.e., what should we consider as a dataset for cleaning?). Nevertheless, most stream languages relies on “black box” mappings among streams to relations to store the stream data in the form of sliding windows. For instance, CQL [18] uses “$S[\text{Rows 5}]$” to specify a 5-element sliding window on stream $S$. Likewise, it uses “$S[\text{Range 30 Seconds}]$” to denote a time-based sliding window of 30 seconds over input stream $S$.

One possibility is to define the cleaning semantics on sliding windows instead of defining them on a data stream directly. However, since the process of cleaning is data dependent, this would be challenging and can lead to counter-intuitive answer semantics.
In this research, we envision that the first challenge we must address is to correctly define (i) the semantics of cleaning and (ii) the correctness of continuous queries answers in the context of streaming data. Once both are defined we can turn our attention to developing efficient solutions that exploit the semantics of the analysis tasks to reduce cleaning when data is streaming.
Bibliography


[27] P. Christen et al. *Towards parameter-free blocking for scalable record linkage*. Department of Computer Science, Faculty of Engineering and Information Technology, Australian National University, 2007. 8


[71] S. M. Van Dongen. Graph clustering by flow simulation. 2001. 10


