Equivalence of Queries with Nested Aggregation

by

David E. DeHaan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

© David E. DeHaan 2009
I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.
Abstract

Query equivalence is a fundamental problem within database theory. The correctness of all forms of logical query rewriting—join minimization, view flattening, rewriting over materialized views, various semantic optimizations that exploit schema dependencies, federated query processing and other forms of data integration—requires proving that the final executed query is equivalent to the original user query. Hence, advances in the theory of query equivalence enable advances in query processing and optimization.

In this thesis we address the problem of deciding query equivalence between conjunctive SQL queries containing aggregation operators that may be nested. Our focus is on understanding the interaction between nested aggregation operators and the other parts of the query body, and so we model aggregation functions simply as abstract collection constructors. Hence, the precise language that we study is a conjunctive algebraic language that constructs complex objects from databases of flat relations. Using an encoding of complex objects as flat relations, we reduce the query equivalence problem for this algebraic language to deciding equivalence between relational encodings output by traditional conjunctive queries (not containing aggregation). This encoding-equivalence cleanly unifies and generalizes previous results for deciding equivalence of conjunctive queries evaluated under various processing semantics. As part of our study of aggregation operators that can construct empty sub-collections—so-called “scalar” aggregation—we consider query equivalence for conjunctive queries extended with a left outer join operator, a very practical class of queries for which the general equivalence problem has never before been analyzed. Although we do not completely solve the equivalence problem for queries with outer joins or with scalar aggregation, we do propose useful sufficient conditions that generalize previously known results for restricted classes of queries. Overall, this thesis offers new insight into the fundamental principles governing the behaviour of nested aggregation.
Acknowledgements

I would like to thank my supervisor, Frank Wm. Tompa, for his continued encouragement and support throughout my tenure as a graduate student at Waterloo. I greatly appreciate the freedom that Frank provided me to pursue research problems of my own interest, especially when charting the course of this dissertation. In his patient approach to graduate supervision, Frank prioritized giving me the opportunity to grow as an academic. I am extremely grateful for that. I could not ask for a better mentor.

Thank you to my external examiner, Dirk Van Gucht from Indiana University, and to the other members of my examining committee, Ashraf Aboulnaga, David Toman, and Ross Willard, for their valuable comments and questions. My oral defence of this dissertation was a highlight of my time as a graduate student; I am grateful for their part in making that experience an enjoyable yet meaningful culmination of my work as a graduate student.

I did not initially set out to write a theoretical study of query equivalence. In late 2003, Glenn Paulley from Sybase iAnywhere suggested that there were many unsolved problems regarding the use of materialized views. I am very grateful to Glenn for that suggestion. My preliminary investigations led me to an ongoing research project by Paul Larson at Microsoft Research which modelled the use of distributed caches as a materialized view rewriting problem. Paul invited me to work with him during the summer of 2004, and I am very thankful to him and to MSR for furnishing that opportunity. Working within the Microsoft SQL Server code base was a wonderful experience, affording many insights into the design of commercial-grade query optimizers. Reflecting on my work at the completion of my internship, I had some lingering discontent over my inability to determine when there existed a query rewriting over a materialized view that our algorithm was failing to find. The seeds of the thesis topic had been sown.

Returning to graduate study with an interesting problem in hand, I found myself woefully unprepared to tackle it. I am indebted to both David Toman and Grant Weddell for their patient handling of the countless questions of a database-theory neophyte. Their combined help in pointing me toward useful concepts and/or literature was instrumental in the formation and writing of this dissertation.

Many friends and colleagues within the UW database group have encouraged and supported me. I owe a special debt of gratitude to Khuzaima Daudjee, a close friend and former office-mate. Khuzaima’s patient listening and sage counsel over many years and during uncountable coffee runs got me past many unforeseen obstacles.

Finally, and most of all, I thank my wife Cara for her confidence in my abilities and firm support of my goal of completing this dissertation. As draft proofs coalesced only to disintegrate like sand castles on a beach, she journeyed beside me and helped me to keep perspective and focus.

*Soli Deo gloria.*
# Contents

List of Figures .......................... xiii

List of Symbols .......................... xvii

1 Introduction .......................... 1
   1.1 Motivating Examples .................. 2
   1.2 Related Problems and Technologies ....... 7
   1.3 Road Map ........................... 10

2 Equivalence of Conjunctive Queries that Construct Objects 11
   2.1 Complex Objects ..................... 11
      2.1.1 Objects and Sorts ................. 11
      2.1.2 Object Linearization ............... 17
      2.1.3 Object-Constructing Queries ......... 24
   2.2 Encoding Objects as Relations ............ 34
      2.2.1 Encoding and Decoding Chain Objects ....... 34
      2.2.2 Encoding and Decoding Linearized Objects ...... 38
      2.2.3 Encoding-Equality ................... 41
   2.3 Encoding Queries .................... 44
      2.3.1 Hierarchical Conjunctive Encoding Queries ... 46
      2.3.2 Reducing Equivalence to Encoding-Equivalence .... 52
   2.4 Relevant Literature ................... 65
      2.4.1 Query Languages for Complex Objects ...... 65
      2.4.2 Relational Encodings of Complex Objects ....... 67
      2.4.3 Algebraic Transformations ............... 68
      2.4.4 Query Equivalence ................... 69
3 Encoding-Equivalence of Conjunctive Encoding Queries

3.1 A Normal Form for CEQs
3.1.1 Dependencies Over Query Results
3.1.2 Converting CEQs to Normal Form
3.2 CEQ Encoding-Equivalence
3.3 Relevant Literature
3.3.1 Conjunctive Queries under Set Semantics
3.3.2 Other Processing Semantics
3.3.3 Aggregation Queries

4 Equivalence of Conjunctive Queries with Outer Joins

4.1 Conjunctive Queries with Outer Joins
4.1.1 The SPCL Algebra
4.1.2 Directed Acyclic Conjunctive Queries
4.2 A Canonical Form for DACQs
4.2.1 Functionally-Determined Nullability
4.2.2 Converting DACQs to Canonical Form
4.3 Equivalence of Constant-Headed Queries
4.3.1 Head Graphs and Canonical Tuples
4.3.2 Tuple-Specific Equivalence
4.4 Queries with Variable Heads
4.4.1 Canonical Freezings and Canonical Tuples
4.4.2 Avoiding Tuple-Specific Equivalence Tests
4.5 Soundness and Completeness
4.5.1 Early Termination
4.5.2 Constraint Relaxation
4.6 Relevant Literature
4.6.1 Algebraic Optimization of Outer Joins
4.6.2 Queries with Negation
4.6.3 Query Containment with Null Values
4.6.4 Query Containment under Dependencies
4.6.5 Dependencies over Relations with Nulls
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A simple relational database schema for tracking company sales</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Quarterly average order value, per agent</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Maximal residential customer order counts, per quarter</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Graphical representation of sorts</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Completion of objects $o_7$, $o_8$, and $o_9$</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Linearization of sorts $\tau_7$, $\tau_8$, and $\tau_9$</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Linearization of objects $o_7$, $o_8$, and $o_9$</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>COCQL algebra operators</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Encoding of complete chain object $o_4$</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>Encoding relation $R_3$ and select sub-relations</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>Encoding relations $R_4$, $R_5$, and $R_6$</td>
<td>39</td>
</tr>
<tr>
<td>2.9</td>
<td>Encoding relation $R_7$</td>
<td>41</td>
</tr>
<tr>
<td>2.10</td>
<td>ns-certificate proving $R_1 \ns \neq R_3$</td>
<td>45</td>
</tr>
<tr>
<td>2.11</td>
<td>Hierarchical conjunctive query $Q_{17}$</td>
<td>48</td>
</tr>
<tr>
<td>2.12</td>
<td>The six trees in $\text{trees}(Q_{17})$</td>
<td>50</td>
</tr>
<tr>
<td>2.13</td>
<td>Embeddings of $\text{trees}(Q_{17})$ into database $D_2$</td>
<td>51</td>
</tr>
<tr>
<td>2.14</td>
<td>Terminal embeddings and their partial applications to $\overline{W}$</td>
<td>51</td>
</tr>
<tr>
<td>2.15</td>
<td>COCQL{$f_b,f_s,f_d$} queries $Q_{18}$ and $Q_{19}$</td>
<td>53</td>
</tr>
<tr>
<td>2.16</td>
<td>Output sort $\tau$ and its linearization $\text{Chain}(\tau) = (\text{snsb}, 5)$</td>
<td>54</td>
</tr>
<tr>
<td>2.17</td>
<td>Query result $o_{12} = (Q_{18})^{D_3} = (Q_{19})^{D_3}$ and its linearization w.r.t. $\tau$</td>
<td>55</td>
</tr>
<tr>
<td>2.18</td>
<td>Encoding query $Q_{20} = \text{ENCODE}(Q_{18})$</td>
<td>57</td>
</tr>
<tr>
<td>2.19</td>
<td>Encoding query $Q_{21} = \text{ENCODE}(Q_{19})$</td>
<td>59</td>
</tr>
<tr>
<td>2.20</td>
<td>Encoding query result $R_8 = (Q_{20})^{D_3}$ satisfying $\text{DECODE}(R_8, \text{snsb}) = o_{13}$</td>
<td>61</td>
</tr>
</tbody>
</table>
2.21 Encoding query result \( R_9 = (Q_{21})^{D_3} \) satisfying DECODE\( (R_9, snsb) = o_{13} \) .......................... 62
2.22 Query result \( o_{13} = \text{DECODE}( (Q_{20})^{D_3}, snsb) = \text{DECODE}( (Q_{21})^{D_3}, snsb) \) 63
2.23 COQL queries \( Q_{22}, Q_{23}, \) and \( Q_{24} \) .......................... 70
2.24 Database instance \( D_4 \) .......................... 71
2.25 Evaluating \( Q'_{22}, Q'_{23}, \) and \( Q'_{24} \) over \( D_4 \) .......................... 71

3.1 Database \( D_5 = \{\text{Prof, Student}\} \) .......................... 76
3.2 Query result \( R_{14} = (Q_{25})^{D_5} = (Q_{26})^{D_5} \) .......................... 77
3.3 Two hypergraphs .......................... 79
3.4 Query results \( R_{11} = (Q_{29})^{D_5}, R_{12} = (Q_{31})^{D_5}, \) and \( R_{13} = (Q_{30})^{D_5} = (Q_{32})^{D_5} \) .......................... 82

4.1 SPCL query \( Q_{33} \) equivalent to HCQ \( Q_{17} \) .......................... 93
4.2 SPCL query results \( R_{14} = (Q_{33})^{D_2} \) and \( R_{15} = (Q_{34})^{D_2} \) .......................... 94
4.3 Directed acyclic conjunctive query \( Q_{35} \) .......................... 95
4.4 Terminal embeddings of \( Q_{35} \) into \( D_2 \) .......................... 95
4.5 DACQ \( Q_{36} \) equivalent to SPCL query \( Q_{34} \) .......................... 96
4.6 Query \( Q_{37} \) and its terminal embeddings into \( D_6 \) .......................... 98
4.7 HCQ \( Q_{40} \) prior to conversion to canonical form .......................... 106
4.8 Query \( Q_{41} = \text{CANONICAL}(Q_{40}) \) .......................... 106
4.9 A constant-headed DACQ with its head graph and canonical tuples 108
4.10 Four CH-DACQs with dual-node head graphs .......................... 114
4.11 Three DACQs with their canonical tuples .......................... 121

5.1 Encoding queries \( Q_{55} = \text{CANONICAL}(Q_{20}) \) and \( Q_{56} = \text{CANONICAL}(Q_{21}) \) 137
5.2 The \( snsb \)-normal forms of \( Q_{55} \) and \( Q_{56} \) and their results over \( D_3 \) .......................... 143
5.3 Encoding queries \( Q_{59}, Q_{60}, \) and \( Q_{61} \) evaluated over database \( D_9 \) .......................... 148
5.4 HCQs \( Q_{62} \) and \( Q_{63} \) .......................... 148

6.1 \( Q_{68} \), the \( \text{COCQL}\{f_{5}, f_{8}, f_{s}\} \) translation of SQL query \( Q_1 \) .......................... 154
6.2 \( Q_{69} \), the \( \text{COCQL}\{f_{5}, f_{8}, f_{s}\} \) translation of SQL query \( Q_3 \) .......................... 155
6.3 Output sort \( \tau_{14} \) of \( Q_{68} \) and \( Q_{69} \), and its transformation to a chain .......................... 155
6.4 Queries \( Q_{70} := \text{ENCODE}(Q_1), Q_{71} := \text{ENCODE}(Q_3) \), and \( Q_{72} := \text{chase}_{\Sigma}(Q_{70}) \) 156

xiv
6.5  \( Q_73 \) and \( Q_{74} \), the \( \text{COCQL}\{f_s,f_b,f_n\} \) translations of SQL queries \( Q_4 \) and \( Q_6 \) 158

6.6  Encoding queries derived from \( Q_73 \) and \( Q_{74} \) ........................................ 158

A.1  Databases \( D_{10} \) and \( D_{11} \) for Example 65 ................................. 167
A.2  Query results \( R_1 = (Q_{78})^{D_{10}}, R_2 = (Q_{79})^{D_{10}}, \) and \( R_3 = (Q_{80})^{D_{10}} \) . . . 167
A.3  Query results \( R_4 = (Q_{78})^{D_{11}}, R_5 = (Q_{79})^{D_{11}}, \) and \( R_6 = (Q_{80})^{D_{11}} \) . . . 167
A.4  Database \( \Delta^{[2,1,3,2]}(D_{10}) \) ............................................................ 169
A.5  Database \( D_{12} = \{\text{Wrote}\} \), query result \( R_7 = (Q_{81})^{D_{12}}, \) and object \( o_{16} = \text{DECODE}(R_7, ss) \). ................................................... 175
A.6  Database \( D_{14} = \Delta^{[2,3,2]}(D_{13}) \), query results \( R_{8} = (Q_{86})^{D_{14}} \) and \( R_{9} = (Q_{87})^{D_{14}}, \) and object \( o_{17} = \text{DECODE}(R_8, n) = \text{DECODE}(R_9, n) \) . . . 184
A.7  Database \( D_{16} = \Delta^{[2,3,2,1,1]}(D_{15}) \), query results \( R_{30} = (Q_{86})^{D_{16}} \) and \( R_{31} = (Q_{87})^{D_{16}}, \) and objects \( o_{18} = \text{DECODE}(R_{30}, n) \) and \( o_{19} = \text{DECODE}(R_{31}, n) \). 184
## List of Symbols

*Page numbers indicate the location of definitions.*

### A

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{A}$</td>
<td>(set of aggregation functions)</td>
<td>26</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>(set of aggregation functions)</td>
<td>28</td>
</tr>
<tr>
<td>$f_b$</td>
<td>(bag constructor)</td>
<td>28</td>
</tr>
<tr>
<td>$f_n$</td>
<td>(normalized bag constructor)</td>
<td>28</td>
</tr>
<tr>
<td>$f_s$</td>
<td>(set constructor)</td>
<td>28</td>
</tr>
</tbody>
</table>

### B

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{body}_v$</td>
<td>(body of CQ formed by $v$ and its ancestors)</td>
<td>47</td>
</tr>
<tr>
<td>$\text{body}_T$</td>
<td>(body of CQ formed from all vertices in $T$)</td>
<td>49</td>
</tr>
<tr>
<td>$\text{body}_G$</td>
<td>(body of CQ formed from all vertices of $G$)</td>
<td>95</td>
</tr>
<tr>
<td>$\text{body}_{\text{ancestors}}$</td>
<td>(body of CQ formed from ancestors of $v$)</td>
<td>47</td>
</tr>
<tr>
<td>$\text{body}_v$</td>
<td>(body of CQ within vertex $v$)</td>
<td>47</td>
</tr>
<tr>
<td>$\text{body}_Q$</td>
<td>(body of query $Q$)</td>
<td>46</td>
</tr>
</tbody>
</table>

### C

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_Q$</td>
<td>(canonical database for query $Q$)</td>
<td>173, 179, 186</td>
</tr>
</tbody>
</table>

### D

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dags}(G)$</td>
<td>(DAGs form-able by deletion from $G$, set of)</td>
<td>95</td>
</tr>
<tr>
<td>$\text{dags}(\text{body}_Q, \mathcal{S})$</td>
<td>(DAGs in $\text{dags}(\text{body}_Q)$ that output tuples of shape $\mathcal{S}$)</td>
<td>109</td>
</tr>
<tr>
<td>$D$</td>
<td>(database)</td>
<td>25</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Minimal DAG in ( \text{dags(\text{body}_{Q,T})} )</td>
<td>109</td>
</tr>
<tr>
<td>N</td>
<td>(null constant)</td>
<td>38</td>
</tr>
<tr>
<td>( \cong )</td>
<td>(nullability dependency)</td>
<td>98</td>
</tr>
<tr>
<td>( S_Q )</td>
<td>(nullability set of DACQ ( Q ))</td>
<td>102</td>
</tr>
<tr>
<td>#(t, c_i)</td>
<td>(number of occurrences of ( c_i ) within ( t ))</td>
<td>169</td>
</tr>
<tr>
<td>O</td>
<td>(output attributes, set of labelled)</td>
<td>50</td>
</tr>
<tr>
<td>( \mathcal{V} )</td>
<td>(output attributes, set of unlabelled)</td>
<td>35, 50</td>
</tr>
<tr>
<td>( \mathcal{V}^\prime )</td>
<td>(output attributes, tuple of)</td>
<td>35, 50</td>
</tr>
<tr>
<td>P</td>
<td>( \text{delta}_i )</td>
<td>168</td>
</tr>
<tr>
<td>( \text{delta}^{-1} )</td>
<td>(painting function inverse (whitewash))</td>
<td>168</td>
</tr>
<tr>
<td>( \mathcal{P}^\text{anc}_v )</td>
<td>(parameter variables (inter-query) from root to vertex ( v ), exclusive)</td>
<td>48</td>
</tr>
<tr>
<td>( \mathcal{P}_v )</td>
<td>(parameter variables (inter-query) from root to vertex ( v ), inclusive)</td>
<td>48</td>
</tr>
<tr>
<td>( \mathcal{P}_Q )</td>
<td>(parameter variables (inter-query) within body ( Q ))</td>
<td>48</td>
</tr>
<tr>
<td>( \mathcal{P}_v )</td>
<td>(parameter variables (inter-query) within vertex ( v ))</td>
<td>47</td>
</tr>
<tr>
<td>( \mathcal{P}_a )</td>
<td>(parameter variables for shape ( s ))</td>
<td>110</td>
</tr>
<tr>
<td>( \mathcal{P} )</td>
<td>(parameter variables of a query)</td>
<td>46, 47</td>
</tr>
<tr>
<td>( \Pi^2_v )</td>
<td>(polynomial hierarchy, second tier)</td>
<td>87</td>
</tr>
<tr>
<td>( \Sigma^2_v )</td>
<td>(polynomial hierarchy, second tier)</td>
<td>87</td>
</tr>
<tr>
<td>( \text{pred}_v )</td>
<td>(predicates within ( v ) or its ancestors)</td>
<td>47</td>
</tr>
<tr>
<td>( \text{pred}_\text{anc}^v )</td>
<td>(predicates within ancestors of ( v ))</td>
<td>47</td>
</tr>
<tr>
<td>( \text{pred}_Q )</td>
<td>(predicates within query ( Q ))</td>
<td>46</td>
</tr>
<tr>
<td>( \text{pred}_T )</td>
<td>(predicates within tree ( T ))</td>
<td>49</td>
</tr>
<tr>
<td>( \text{pred}_v )</td>
<td>(predicates within vertex ( v ))</td>
<td>47</td>
</tr>
<tr>
<td>Q</td>
<td>( H^Q )</td>
<td>(query hypergraph)</td>
</tr>
<tr>
<td>( Q^\text{db} )</td>
<td>(query result over database ( D ))</td>
<td>47</td>
</tr>
<tr>
<td>S</td>
<td>( \text{delta}^{-1} )</td>
<td>(whitewash function)</td>
</tr>
<tr>
<td>( Q_{ls} )</td>
<td>(shape-( s ) query for DACQ ( Q ))</td>
<td>109</td>
</tr>
<tr>
<td>( \mathcal{Q} )</td>
<td>(shape query for DACQ ( Q ))</td>
<td>109</td>
</tr>
<tr>
<td>( \mathcal{Q}^\mathcal{S} )</td>
<td>(signature)</td>
<td>15</td>
</tr>
<tr>
<td>( \delta_d )</td>
<td>(simulation to depth ( d ))</td>
<td>69</td>
</tr>
<tr>
<td>( \tau )</td>
<td>(sort)</td>
<td>13</td>
</tr>
<tr>
<td>( \text{sort}(A_i) )</td>
<td>(sort of attribute ( A_i ))</td>
<td>25</td>
</tr>
<tr>
<td>( T^\mathcal{S} )</td>
<td>(sorts, set of all)</td>
<td>13</td>
</tr>
<tr>
<td>( \text{trees}(T) )</td>
<td>(trees form-able by deletion from ( T ), set of)</td>
<td>48</td>
</tr>
<tr>
<td>( { \cdot } )</td>
<td>(tuple)</td>
<td>13</td>
</tr>
<tr>
<td>( t^Q )</td>
<td>(tuple shape in ( TS_Q ) corresponding to tuple ( t \in CT_Q ))</td>
<td>111, 120</td>
</tr>
<tr>
<td>( TS_Q )</td>
<td>(tuple shapes of DACQ ( Q ), set of)</td>
<td>108</td>
</tr>
<tr>
<td>( \mathcal{B} )</td>
<td>(variables occurring in a query)</td>
<td>46, 47</td>
</tr>
<tr>
<td>( \mathcal{B}^\text{anc}_v )</td>
<td>(variables within ( \text{body}^\text{anc}_v ))</td>
<td>47</td>
</tr>
<tr>
<td>( \mathcal{B}_v )</td>
<td>(variables within ( \text{body}_v ))</td>
<td>47</td>
</tr>
<tr>
<td>( \mathcal{B}_T )</td>
<td>(variables within tree ( T ))</td>
<td>49</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In this thesis we consider the problem of deciding when a pair of queries posed to a database system are equivalent—that is, guaranteed to return identical results. The query equivalence problem (and the closely related query containment problem) is a fundamental problem that appears within a variety of contexts in the design of database management systems.

The most obvious application of query equivalence is to the “global” query optimization problem: given a query and a cost model, find within the space of equivalent query formulations the query with minimal cost [6, Ch. 6]. For the class of simple conjunctive queries, query equivalence is well understood and global optimization techniques are widely implemented [16, 19]; however, query equivalence (and hence global optimization) is in general undecidable for arbitrary relational algebra expressions [6, Ch. 6]. For this reason, most database systems rely upon “local” optimization strategies for complex queries—i.e., the generation of alternative query formulations via localized (often algebraic) transformation rules, often called “query rewriting” or “logical rewriting” [17]. Within this thesis we study equivalence for queries defined by conjunctive relational algebra expressions extended with an aggregation operator that may be arbitrarily nested within the algebra expression. Our work thus contrasts with much of the literature on aggregation queries which assumes a single top-level aggregation operation. Our results are applicable to both forms of query optimization: by characterizing equivalence over more powerful classes of queries, we both enable global optimization techniques for those classes and enable development of more powerful rewriting rules for complex queries.

Beyond its application to query optimization, the query equivalence problem is directly applicable to various problems within information integration. The view rewriting problem—rewriting a query to use a set of materialized views—may arise either in the context of cost-based query optimization or in the context of information integration where the views model the underlying data sources [57]. The data exchange problem—transforming a database instance conforming to a source schema into an instance conforming to a target schema—is also based upon
query equivalence [33]. Finally, the recent advent of object-relational mappings promises convenient integration of relational data into object-oriented programming languages, but translating operations over objects into operations over relations requires reasoning over declarative mappings that makes heavy use of query equivalence tests (in the guise of materialized-view answering and maintenance problems) [83].

1.1 Motivating Examples

Our interest in query equivalence for conjunctive queries with nested aggregation stems from the problem of optimizing complex SQL queries. In particular, we are interested in how existing query optimizers might be extended with robust algorithms for cost-based view rewriting in a context where complex materialized view definitions are formed by “stacking” (i.e., joining and re-aggregating) simpler views. This class of so-called “stacked” materialized views is interesting in practice both because it allows efficient incremental maintenance algorithms, and because stacked views are useful for answering the complex aggregation queries that appear within decision-support workloads [29]. Unfortunately, as illustrated in the following examples, view rewriting algorithms implemented within contemporary relational database management systems (RDBMSs) are not robust when rewriting queries over complex view definitions.

Example 1 Consider the database schema shown in Figure 1.1, which stores information about customer orders solicited by a company’s agents. The database schema contains six base tables with primary and foreign key constraints (primary keys denoted by underlined attributes), along with six views defined by SQL queries, four of which are materialized.

The attribute Customer.ctype is code that classifies customers as either Residential or Corporate, and sales from the two sectors are often reported separately. Suppose that an end user desires a report that lists for each agent the quarterly average order value, with the Residential and Corporate metrics shown in separate columns. Equipped with a reporting tool capable of generating single-block conjunctive SQL queries (with top-level aggregation) only, the user could accomplish the desired report by using the logical view AgentSales, generating the query $Q_1$ shown in Figure 1.2.

\[ Q: \text{SELECT } * \]
\[ \text{FROM Customer, Order } o1, \text{ Order } o2 \]
\[ \text{WHERE Customer.cid } = \text{ o1.cid} \]
\[ \text{and Customer.cid } = \text{ o2.cid} \]

\[ Q': \text{SELECT } * \]
\[ \text{FROM C } \bowtie_{cid} \text{ o1 } \bowtie_{cid} \text{ o2} \]

For conciseness, we have abbreviated the SQL syntax by shortening relation names to their capital letters (using subscripts to distinguish repeated relations), and by using an equi-join operator to denote the join predicates. For example, the following query $Q$ (in proper SQL syntax) is abbreviated as query $Q'$.  

\[ Q: \text{SELECT } * \]
\[ \text{FROM Customer, Order } o1, \text{ Order } o2 \]
\[ \text{WHERE Customer.cid } = \text{ o1.cid} \]
\[ \text{and Customer.cid } = \text{ o2.cid} \]
<table>
<thead>
<tr>
<th>Abbr</th>
<th>Table Schema</th>
<th>Foreign Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Customer(cid, cname, ctype)</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>Order(oid, cid, date)</td>
<td>date:Date(date)</td>
</tr>
<tr>
<td>LI</td>
<td>LineItem(oid, lineno, price, qty)</td>
<td>oid:Order(oid)</td>
</tr>
<tr>
<td>A</td>
<td>Agent(aid, aname)</td>
<td></td>
</tr>
<tr>
<td>OA</td>
<td>OrderAgent(oid, aid)</td>
<td>oid:Order(oid), aid:Agent(aid)</td>
</tr>
<tr>
<td>D</td>
<td>Date(date, qtr)</td>
<td></td>
</tr>
</tbody>
</table>

Logical (non-Materialized) Views:

**AS** AgentSales(aid, aname, date, ctype, oid, oval)

```sql
SELECT aid, aname, date, ctype, oid, SUM(price * qty)
FROM C ⊙cid O ⊙oid LI ⊙oid OA ⊙aid A
GROUP BY aid, aname, date, ctype, oid
```

**QC** QuarterlyCustomer(qtr, cid, cname, ctype)

```sql
SELECT qtr, cid, cname, ctype
FROM C ⊙cid QCO
```

Materialized Views:

**OV** OrderValues(oid, oval)

```sql
SELECT oid, SUM(price * qty)
FROM LI
GROUP BY oid
```

**QAS** QuarterlyAgentSales(aid, qtr, ctype, avgOval)

```sql
SELECT aid, qtr, ctype, AVG(oval)
FROM C ⊙cid O ⊙oid OV ⊙oid OA ⊙aid A ⊙date D
GROUP BY aid, qtr, ctype
```

**QCO** QuarterlyCustomerOrders(qtr, cid, ordCount)

```sql
SELECT qtr, cid, COUNT(*)
FROM O ⊙date D
GROUP BY qtr, cid
```

**QMCO** QuarterlyMaxCustomerOrders(qtr, ctype, maxOrdCount)

```sql
SELECT qtr, ctype, MAX(ordCount)
FROM C ⊙cid QCO
GROUP BY qtr, ctype
```

Figure 1.1: A simple relational database schema for tracking company sales
$Q_1$: \[
\text{SELECT } \text{AS}_1\text{.aname, qtr,} \\
\quad \text{avg}(\text{AS}_1\text{.oval}) \text{ AS avgRsale,} \\
\quad \text{avg}(\text{AS}_2\text{.oval}) \text{ AS avgCsale} \\
\text{FROM (AS}_1 \bowtie_{\text{date } D_1} \bowtie_{\{\text{aid, qtr}\}} (\text{AS}_2 \bowtie_{\text{date } D_2}) \\
\quad \text{WHERE AS}_1\text{.ctype = ‘R’} \\
\quad \text{AND AS}_2\text{.ctype = ‘C’} \\
\text{GROUP BY aid, AS}_1\text{.aname, qtr}
\]

$Q_2$: \[
\text{SELECT } A_1\text{.aname, qtr,} \\
\quad \text{avg}(\text{OV}_1\text{.oval}) \text{ AS avgRsale,} \\
\quad \text{avg}(\text{OV}_2\text{.oval}) \text{ AS avgCsale} \\
\text{FROM (C}_1 \bowtie_{\text{cid } O_1} \bowtie_{\text{oid } OV_1} \bowtie_{\text{oid } OA_1} \bowtie_{\text{aid } A_1} \bowtie_{\text{date } D_1}) \bowtie_{\{\text{aid, qtr}\}} \\
\quad (C_2 \bowtie_{\text{cid } O_2} \bowtie_{\text{oid } OV_2} \bowtie_{\text{oid } OA_2} \bowtie_{\text{aid } A_2} \bowtie_{\text{date } D_2}) \\
\quad \text{WHERE C}_1\text{.ctype = ‘R’} \\
\quad \text{AND C}_2\text{.ctype = ‘C’} \\
\quad \text{GROUP BY aid, A}_1\text{.aname, qtr}
\]

$Q_3$: \[
\text{SELECT } \text{aname, qtr,} \\
\quad QAS_1\text{.avgOval AS avgRsale,} \\
\quad QAS_2\text{.avgOval AS avgCsale} \\
\text{FROM A } \bowtie_{\text{aid }} QAS_1 \bowtie_{\{\text{aid, qtr}\}} QAS_2 \\
\quad \text{WHERE QAS}_1\text{.ctype = ‘R’} \\
\quad \text{AND QAS}_2\text{.ctype = ‘C’}
\]

Figure 1.2: Quarterly average order value, per agent
We implemented query $Q_1$ on the latest versions of several commercial RDBMSs. The best rewriting of $Q_1$ found by any system that we tested uses schema information to push down the $\text{SUM}$ aggregate within $\text{AgentSales}$ in order to rewrite over two occurrences of the materialized view $\text{OrderValues}$—resulting in query $Q_2$ (cf. Figure 1.2). Observe that both $Q_1$ and $Q_2$ perform a Cartesian product between each agent’s quarterly Residential and Corporate orders prior to the aggregation.\textsuperscript{2}

In contrast, within query $Q_3$ (cf. Figure 1.2) the Residential and Corporate orders are aggregated independently. This allows the query to be rewritten over two instances of the materialized view $\text{QuarterlyAgentSales}$, yielding a drastically more efficient execution plan. (Even if view $\text{QuarterlyAgentSales}$ were not materialized, $Q_3$ would still allow much more efficient execution than either $Q_1$ or $Q_2$ because it avoids computing the Cartesian product as an intermediate result.) In Examples 62 and 63 (Chapter 6) we will show that $Q_3$ is equivalent to $Q_1$ over all database instances that conform to the primary and foreign key constraints in the database schema, but not equivalent over arbitrary instances.

Complex SQL queries containing aggregation are not restricted to decision-support workloads. The popularity of object-relational mapping tools means that the SQL queries within contemporary database workloads are increasingly program-generated, rather than hand-crafted. The trade-off in providing this layer of abstraction to the application programmers is that queries generated via compositions of mappings or logical views can be highly nested, with significant redundancy that is challenging for the optimizer to simplify. We illustrate this with an example.

Example 2 Consider again the database schema in Figure 1.1. The sales division of the company commonly refers to its “Quarterly Customers,” meaning the set of customers who have placed an order during a given quarter. To support this concept, the database administrator created the materialized view $\text{QuarterlyCustomerOrders}$ which totals the number of orders by quarter and customer id. She then created the logical view $\text{QuarterlyCustomer}$ for the convenience of users who only need to know the identity of the quarterly customers, not the count of their orders.

Application programmers at the company use an object-oriented query framework (on top of an object-relational mapping) to interface with the database. Defined within the entity schema of this framework are entities corresponding to each table and view within the database schema, along with entities $\text{QuarterlyResidentialCustomer}$

\textsuperscript{2}Some of the RDBMSs were restricted to query optimization over “single-block” materialized-view definitions. In this case, we tested their hypothetical rewriting ability over the “stacked” materialized views in Figure 1.1 by first testing their ability to rewrite over the leaf level views, and then—if successful—we converted the leaf level views to base tables and manually rewrote the queries over them in order to test rewriting at the next level. Some of the RDBMSs did not support the $\text{MAX}$ or $\text{AVG}$ aggregation functions within materialized-view definitions. Because an average can be computed as a sum divided by a count, for systems that did not support $\text{AVG}$ in materialized views we added the appropriate $\text{SUM}$ and $\text{COUNT}$ expressions to the view and then tested the query both with the original $\text{AVG}$ expressions and with manual decompositions into $\text{SUM}$ and $\text{COUNT}$. 

5
\(Q_4: \) \[\text{SELECT qtr, max(countOrders)}\]
\[\text{FROM (SELECT qtr, cid, cname,}\]
\[\left(\text{SELECT count(*)}\right)\]
\[\text{FROM D \(\bowtie_{date} \) O}\]
\[\text{WHERE D.qtr = QC.qtr}\]
\[\text{AND O.cid = QC.cid}\]
\[\) \text{AS countOrders}\]
\[\text{FROM (SELECT qtr, cid, cname}\]
\[\text{FROM QC}\]
\[\text{WHERE ctype = 'R'}\]
\[\) \text{AS QuarterlyResidentialCustomer}\]
\[\) \text{AS QuarterlyResCustWithOrderCounts}\]
\[\text{GROUP BY qtr}\]

\(Q_5: \) \[\text{SELECT qtr, max(countOrders)}\]
\[\text{FROM (SELECT qtr, cid, cname,}\]
\[\left(\text{SELECT count(*)}\right)\]
\[\text{FROM QCQ}\]
\[\text{WHERE QCQ.qtr = QCQ1.qtr}\]
\[\text{AND QCQ.cid = QCQ1.cid}\]
\[\) \text{AS countOrders}\]
\[\text{FROM C \(\bowtie_{cid} \) QCQ}\]
\[\text{WHERE ctype = 'R'}\]
\[\) \text{AS QuarterlyResCustWithOrderCounts}\]
\[\text{GROUP BY qtr}\]

\(Q_6: \) \[\text{SELECT qtr, maxOrdCount}\]
\[\text{FROM QMCO}\]
\[\text{WHERE ctype = 'R'}\]

Figure 1.3: Maximal residential customer order counts, per quarter
and QuarterlyCorporateCustomer that inherit from entity QuarterlyCustomer. Assume that an application programmer (who does not know SQL) requires a list of the maximum number of orders placed by quarterly residential customers, grouped by quarter. The programmer is not familiar with either the QuarterlyCustomerOrders or QuarterlyMaxCustomerOrders entities. Instead, within his object-oriented query framework he first creates an entity QuarterlyResCustWithOrderCounts, formed by supplementing QuarterlyResidentialCustomer with a computed attribute countOrders containing the count of the corresponding order values. Next, he selects the maximum value of countOrders across all instances of QuarterlyResCustWithOrderCounts, grouped by quarter. Query Q_4 in Figure 1.3 shows the SQL query generated by the programming framework. Observe that derived attribute countOrders is rendered as a SELECT-clause sub-query. From the perspective of entity-to-SQL translation this is the correct translation of a computed attribute, because QuarterlyResidentialCustomer entities without corresponding orders should yield instances of QuarterlyResCustWithOrderCounts entities with countOrders=0, and these zeros could potentially affect the calculation of the maximum. The SQL generator within the entity framework does not know that each instance of QuarterlyResidentialCustomer is guaranteed to have at least one corresponding order, because that information is encapsulated in the logical view QuarterlyCustomer inside the RDBMS.

Consider queries Q_5 and Q_6 in Figure 1.3. Query Q_5 is the best rewriting of Q_4 found by any RDBMS that we tested. None of the RDBMSs were able to rewrite Q_4 over the materialized view QuarterlyMaxCustomerOrders; however, in Example 64 (Chapter 6) we will show that queries Q_4 and Q_6 are equivalent.

1.2 Related Problems and Technologies

Materialized Views and Data Warehousing Halevy has written a thorough survey of research on the use of materialized views to answer queries [57]. He draws an important distinction separating works whose purpose is query optimization (including maintenance of physical data independence) from works whose purpose is data integration. Data integration typically utilizes rewritings that are contained within the original query, whereas query optimization requires rewritings to be equivalent to the original query. Because our motivation stems from query optimization, we restrict our attention to query equivalence and do not consider query containment in this thesis. Within the context of query optimization, Halevy distinguishes works treating view rewriting as a logical optimization (i.e., as a query language-level transformation) from works treating view rewriting as a physical optimization (i.e., during generation of query execution plans). Although this distinction is pertinent to how our results might be integrated into existing RDBMSs (see Section 6.3), it is orthogonal to the issue of guaranteeing equivalence between the original query and the rewritten query. Hence, for the purposes of motivating our study of query equivalence it suffices to treat view rewriting as a purely logical optimization (which is why in Examples 1 and 2 we show the rewritings performed
by RDBMS optimizers as SQL queries, even though the output we obtained from the optimizers were actually physical execution plans).

For the restricted case of conjunctive rewritings of conjunctive queries over conjunctive views, the query equivalence problem is solved [16, 19], as is the view rewriting problem both as a logical optimization [74] and as a physical optimization integrated into contemporary optimizer architectures [18, 44, 45]. From the perspective of query optimization, pre-materialization of views offers the greatest potential savings when the size of the query output is vastly smaller than the size of the raw data that must be examined in order to evaluate the query over a database instance. For this reason, materialized views are especially important within data warehouse environments, due to the heavy use of aggregation within decision-support queries (e.g., see the queries within the TPC-H [105] or TPC-DS [104] benchmark workloads). One restricted form of data warehousing depends upon a star schema (i.e., a (very large) central fact table connected by foreign key joins to (small) dimension tables), and both queries and materialized views are limited to defining portions of a multidimensional data cube [48]. Within this context, testing query equivalence is trivial, and view rewriting can be efficiently effected by analyzing the data cube lattice [59, 90]. A less restricted form of data warehousing assumes that queries and materialized views are limited to expressions with a single top-level aggregation operator. Within this context, the study of query equivalence has focused on numeric or algebraic properties of specific aggregation functions [25, 51], while the study of view rewriting ranges from formal aspects of modelling properties of aggregation functions [24] and determining the complexity of the rewriting language needed to fully utilize the views [53] to more pragmatic approaches that utilize various algebraic transformations for re-shaping the query to obtain sound (but incomplete) rewriting algorithms [54, 101].

There is only limited previous research that considers view rewriting in contexts where the query and view definitions contain nested aggregation. In one work, a bottom-up matching algorithm is proposed that applies templates based upon (previously known) algebraic transformations [115]. Another work focuses on increasing the robustness and efficiency with which transformational optimizers apply sets of (previously known) transformations to enact view rewriting [29]. All of this work presumes a set of algebraic transformations for re-shaping queries, and there is no consideration of whether the set of transformations is complete for traversing the space of equivalent queries.

Query Simplification and Logical Rewriting Work on simplifying complex SQL queries composed of arbitrarily nested queries was pioneered by Kim, who characterizes different forms of nested sub-queries and focuses on heuristics for flattening or de-correlating them to increase the search space of the query optimizer [69]. (Traditional query optimizer architectures only optimize join orders within the individual blocks of a complex query; hence, flattening nested blocks increases the space of possible join orders, while de-correlating nested blocks allows
the use of physical join operators other than nested loops.) Many of Kim’s heuristics are implemented as logical rewriting rules within modern query optimizers, along with various other algebraic transformations that re-shape the nesting blocks of a query by commuting join and aggregation operators \[54, 113, 114\]. Query simplification via logical rewriting can vastly reduce the execution cost of certain queries; however, the pattern-based rule languages of contemporary optimizers are quite restrictive \[46, 56\]. Performing the degree of query simplification we illustrate in Example 2 requires a formal analysis of the input expression that cannot be conveniently specified as an algebraic template.

**Complex Objects and Object-Relational Mappings** Extensions to the relational model for handling complex objects (a.k.a. complex values or nested relations) were studied extensively during the 1980s; the reader is referred to Abiteboul et al.’s textbook for an introduction to the topic \[6, Ch. 20–21\], and to Section 2.4 of this thesis where we include a brief survey of some of the pertinent literature in this area. Database systems based upon native nested-relational and/or object-oriented data models remain niche products, however. Instead, the current mainstream approach to integrating object-oriented programming languages with persistent database storage is to utilize object-relational mapping (ORM) tools that present the programmer with an abstraction of the relational schema in terms of abstract entities. A fixed language of query operations over these entities is then translated by the ORM framework into SQL queries over the underlying relations. Popular examples include Hibernate, which provides the object-oriented query language HQL \[1\]; ActiveRecord, the ORM component of the Ruby on Rails programming framework \[3\]; and the ADO.NET Entity Framework, the ORM component of Microsoft’s .NET framework which supports its Language Integrated Query (LINQ) facilities \[2, 82\].

The results in this thesis pertain to ORM technologies in two different ways. First, as mentioned in the previous section, ORM tools are quickly emerging as a source of very complex SQL queries, which motivates the study of query equivalence as a means towards deriving more powerful query simplification techniques for relational query optimizers. Second, the data models that ORM tools expose to the programmer are based upon nested collections, and many of the SQL queries issued by the ORM tools are used to construct in-memory complex objects out of data stored in flat relations. Within our study of nested aggregation we abstract aggregation functions as collection constructors, and so the query language we study constructs complex objects out of databases of flat relations. Hence, it is foreseeable that our results could be applied not only to optimization of SQL queries at the RDBMS level, but also to improve the quality of SQL translations generated by ORM packages.
1.3 Road Map

The query language for which we propose to study equivalence is a conjunctive algebra extended with aggregation functions that construct abstract collections which—when nested—form complex objects. We consider objects constructed by arbitrary nesting of three different types of unordered collections—sets, bags, and a collection type we call normalized bags—which abstract common aggregation functions by modelling their sensitivity to the tuple cardinalities within their inputs. We have chosen this abstraction to highlight the impact of nested aggregation on the query equivalence problem, without getting bogged down in complexities specific to particular aggregation functions.

In Chapter 2 we lay the necessary groundwork for considering the query equivalence problem for queries with nested aggregation. We start by formally defining the data model, algebraic query language, and a procedure for encoding arbitrary complex objects within a single flat relation (i.e., containing no aggregated values). The main contribution of this chapter is a reduction of the query equivalence problem to deciding a relationship we call encoding equivalence between conjunctive queries that do not contain any aggregation operators, but may contain an operator that introduces null values (a restricted form of the left outer-join operator).

Chapters 3–5 address the problem raised in Chapter 2 of deciding encoding equivalence between conjunctive queries containing null-introducing operators. In Chapter 3 we focus on understanding encoding equivalence (which is a generalization of standard query equivalence) in the restricted case where the null-introducing operator is not present. In Chapter 4 we focus on standard query equivalence in the case where null-introducing operators are present. We combine these orthogonal results in Chapter 5, where we consider encoding equivalence in the presence of null-introducing operators.

In Chapter 6 we summarize our results. We also revisit the motivating examples from Section 1.1 and use them to illustrate both how our theoretical results can be applied in practice and where shortcomings in our theory exist that require further attention.
Chapter 2

Equivalence of Conjunctive Queries that Construct Objects

Our primary purpose in this chapter is to lay a theoretic foundation for discussing queries—over relational databases—which return complex objects as their result. In future chapters we will build upon this foundation to reason about the equivalence of such queries. In Section 2.1 we define complex objects and related notation, including an algebraic language for constructing complex objects. In Section 2.2 we define a representation of complex objects as relations of flat tuples, and show how our complex object algebra queries can be mapped to queries in a (slightly extended) relational query language. Finally, relevant literature is discussed in Section 2.4.

2.1 Complex Objects

Informally, a complex object is a recursive structure built up from atomic values, tuples of complex objects, and collections of complex objects. In Section 2.1.1 we give a more formal definition of complex objects. In Section 2.1.2 we describe a transformation that we call linearization which converts objects to a canonical form. Finally, we introduce a query language for generating complex objects from a database of flat relations in Section 2.1.3.

2.1.1 Objects and Sorts

Our investigation of complex objects is motivated by asking about equivalence of SQL queries containing aggregation functions, and so our choice of which collection types to consider is determined by the semantics of commonly-supported aggregation functions. All SQL aggregation functions ignore the order of elements within their input, and so we restrict our attention to unordered collections. Aggregation functions such as MIN, MAX, and any function using the DISTINCT keyword (e.g.
COUNT(DISTINCT column-name) also ignore the element cardinalities within their input; thus, we model their input as a set. Aggregate functions such as COUNT and SUM are sensitive to the element cardinalities and so we model their input as a bag (a.k.a. multi-set). Finally, many statistical functions such as AVERAGE, MEDIAN, or STDEV are sensitive only to relative element cardinalities (e.g., doubling the cardinality of each value does not change the aggregated result), and so we model their input using a collection type we call a normalized bag.

Sets and bags are well-known collection types and we denote them using the delimiters \{ \cdot \} and \{ \cdot \}, respectively. Given an arbitrary finite collection \( c \), the bag projection of \( c \)—denoted \( \text{Bag}(c) \)—is the bag of elements that occur in \( c \), while the set projection of \( c \)—\( \text{Set}(c) \)—is the set of elements in \( c \). A normalized bag is a special case of a bag in which the greatest common divisor of the element frequencies is one; we denote a normalized bag with the delimiters \{ \cdot \} . Given an arbitrary collection \( c \), the normalized bag projection of \( c \)—\( \text{NormBag}(c) \)—is the normalized bag formed by dividing each element frequency by the GCD of all element frequencies.

Example 3 The following four bags are all distinct.

\[
\begin{align*}
B_1 & := \{ x, x, y, y, y \} \\
B_2 & := \{ x, x, x, y, y, y, y, y, y \} \\
B_3 & := \{ x, x, y \} \\
B_4 & := \{ x, x, x, x, y, y \}
\end{align*}
\]

The first and second have the same normalized bag projection

\[
N_1 := \{ \cdot x, x, y, y, y \} = \text{NormBag}(B_1) = \text{NormBag}(B_2)
\]

as do the third and fourth.

\[
N_2 := \{ \cdot x, x, y \} = \text{NormBag}(B_3) = \text{NormBag}(B_4)
\]

Finally, the four bags and two normalized bags all have the same set projection.

\[
\text{Set}(B_1) = \text{Set}(B_2) = \text{Set}(B_3) = \text{Set}(B_4) = \text{Set}(N_1) = \text{Set}(N_1) = \{ x, y \}
\]

We now formalize our definition of complex objects. Definitions 2.1.1 and 2.1.4 below borrow heavily from Abiteboul et al.’s definition of complex value databases [6, Ch. 20], the primary difference being our extension to multiple collection types. Another minor difference is that we do not assign attribute names to elements of tuples; the reason for this will be explained when we present our query algebra.

**Sorts**

We start by fixing a countably infinite set of atomic values \( \text{dom} \), called the underlying domain. We desire complex objects to be typed, and so we next define a set of complex sorts, which are meta-entities that each define a class of objects of a restricted structure.
Definition 2.1.1 (Sort) A sort (or type) $\tau$ is syntactically defined by the following recursive grammar:

$$
\tau := \text{dom} \mid \{ \tau \} \mid \{ \{ \tau \} \} \mid \{ \tau, \ldots, \tau \}
$$

We call sort $\text{dom}$ an atomic sort; sorts $\{ \tau \}$, $\{ \{ \tau \} \}$, and $\{ \tau, \ldots, \tau \}$ collection sorts; and sort $\{ \tau, \ldots, \tau \}$ a tuple sort (which may have any finite arity $k \geq 0$). We call a tuple sort flat if it is composed of only atomic sorts. We use $\mathcal{T}$ to denote the set of all sorts.

Definition 2.1.2 (Sort Depth) The depth of a sort $\tau$ is defined recursively as follows.

$$
\begin{align*}
\text{depth}(\text{dom}) &= 0 \\
\text{depth}(\{ \tau \}) &= 1 + \text{depth}(\tau) \\
\text{depth}(\{ \{ \tau \} \}) &= 1 + \text{depth}(\tau) \\
\text{depth}(\{ \tau, \ldots, \tau \}) &= 1 + \text{depth}(\tau) \\
\text{depth}(\{ \tau_1, \ldots, \tau_k \}) &= \max(0, \text{depth}(\tau_1), \ldots, \text{depth}(\tau_k))
\end{align*}
$$

Intuitively, the depth of $\tau$ is the maximum number of collection sorts occurring along any root-to-leaf path in the hierarchical definition of $\tau$.

Although not enforced by the grammar in Definition 2.1.1, without loss of generality we adopt a convention that tuple sorts are always structured with the child sorts of zero depth occurring to the left of child sorts with non-zero depth.

Example 4 Consider the following sorts.

$$
\begin{align*}
\tau_1 &:= \{ \text{dom}, \text{dom}, \text{dom} \} \\
\tau_2 &:= \{ \{ \text{dom} \} \} \\
\tau_3 &:= \{ \{ \text{dom} \}, \{ \text{dom}, \text{dom} \} \} \\
\tau_4 &:= \{ \{ \{ \text{dom} \} \} \} \\
\tau_5 &:= \{ \{ \{ \text{dom} \} \} \} \\
\tau_6 &:= \{ \{ \text{dom} \}, \{ \{ \text{dom} \} \} \} \\
\tau_7 &:= \{ \{ \{ \text{dom} \} \} \} \\
\tau_8 &:= \{ \{ \{ \{ \{ \text{dom} \} \} \} \} \} \\
\tau_9 &:= \{ \{ \{ \{ \text{dom} \} \} \} \} \\
\end{align*}
$$

An equivalent tree representation is shown in Figure 2.1. Sort $\tau_1$ has depth zero, $\tau_2$ and $\tau_3$ have depth one, $\tau_4$–$\tau_7$ have depth two, and $\tau_8$ and $\tau_9$ have depth three.

For each integer $i \geq 0$, we use $\mathcal{T}^i \subset \mathcal{T}$ to denote the (infinite) set of all sorts with depth at most $i$.

$$
\mathcal{T}^i := \{ \tau \mid \text{depth}(\tau) \leq i \} \quad (2.1)
$$

Branching, Non-Branching and Chain Sorts We classify sorts as either branching or non-branching as follows.

- Atomic sort $\text{dom}$ is non-branching.
Figure 2.1: Graphical representation of sorts
• Collection sorts \( \{ \tau \} \), \( \{ \tau \} \), and \( \{ \tau \} \) are branching if \( \tau \) is a branching sort; otherwise they are non-branching.
• Tuple sort \( \{ \tau_1, \ldots, \tau_k \} \) is branching if either
  1. one of its child sorts \( \tau_i \) is branching; or
  2. at least two of its child sorts \( \tau_i \) and \( \tau_j \) have depth greater than zero.

In other words, branching sorts include at least two collection types where neither type is nested within the other. For example, \( \tau_8 \) and \( \tau_9 \) in Figure 2.1 are branching sorts, while \( \tau_1 \)–\( \tau_7 \) are non-branching sorts. By our convention on the structuring of tuple sorts, within a non-branching tuple sort \( \{ \tau_1, \ldots, \tau_k \} \) only \( \tau_k \) can have depth greater than zero. Hence, non-branching sorts contain only “right deep” nesting of collections, as illustrated by sorts \( \tau_6 \) and \( \tau_7 \) in Figure 2.1.

We say that a sort is a *chain sort* if it contains precisely one descendant tuple sort, and that tuple sort is *flat*. Hence, a chain sort is a special case of a non-branching sort. For example, sorts \( \tau_1, \tau_2, \tau_4, \) and \( \tau_5 \) in Figure 2.1 are chain sorts. The object encoding method that we propose later in this chapter is based upon chain sorts. Corresponding to each chain sort we define a sequence \( \bar{\bar{\mathcal{S}}} \) called its *signature*.

**Definition 2.1.3 (Signature)** A semantic indicator \( \bar{\mathcal{S}} \) is one of the characters \( s, b, \) or \( n \) and denotes that a collection is of type set, bag, or normalized bag, respectively. A signature \( \bar{\bar{\mathcal{S}}} \) of length \( d \) is a finite sequence of semantic indicators \( \bar{\mathcal{S}}_1\bar{\mathcal{S}}_2\cdots\bar{\mathcal{S}}_d \).

Any chain sort of depth \( d \) can be represented by a pair \( (\bar{\mathcal{S}}, k) \), where \( \bar{\mathcal{S}} \) is a signature of length \( d \) that indicates from left-to-right the semantics of the successive descendant collection sorts, and \( k \) is the arity of the tuple at the leaf of the sort. For example, chain sorts \( \tau_1, \tau_2, \tau_4, \) and \( \tau_5 \) from Figure 2.1 can be represented as \( (\emptyset, 3), (s, 2), (ns, 1), \) and \( (sb, 1) \), respectively.

**Complex Objects**

Corresponding to each sort \( \tau \in \mathcal{T} \) is a space of possible values that conform to it, called its *interpretation*.

**Definition 2.1.4 (Sort Interpretation)** The interpretation of sort \( \tau \), denoted \( \llbracket \tau \rrbracket \), is defined recursively as follows.

\[
\begin{align*}
\llbracket \text{dom} \rrbracket &= \text{dom} \\
\llbracket \{ \tau \} \rrbracket &= \{ v_1, \ldots, v_j \} \mid j \geq 0, v_i \in \llbracket \tau \rrbracket, i \in [1,j] \\
\llbracket \{ v_1, \ldots, v_j \} \rrbracket &= \{ v_1, \ldots, v_j \} \mid j \geq 0, v_i \in \llbracket \tau \rrbracket, i \in [1,j] \\
\llbracket \{ v_1, \ldots, v_j \} \rrbracket &= \{ v_1, \ldots, v_j \} \mid j \geq 0, v_i \in \llbracket \tau \rrbracket, i \in [1,j] \\
\llbracket \{ \tau_1, \ldots, \tau_k \} \rrbracket &= \{ v_1, \ldots, v_k \} \mid v_j \in \llbracket \tau_j \rrbracket, j \in [1,k] 
\end{align*}
\]
For each integer \( i \geq 0 \), we use \( \text{dom}^i \) to denote the (infinite) set of all finite elements belonging to interpretations of sorts with depth at most \( i \).

\[
\text{dom}^i := \bigcup_{\tau \in \mathcal{T}} [[\tau]]
\]  

We use \( \text{dom}^\infty \) to denote the (infinite) set of all finite elements that conform to any sort.

\[
\text{dom}^\infty := \bigcup_{\tau \in \mathcal{T}} [[\tau]]
\]  

**Definition 2.1.5 (Complex Object)** A complex object is a member of \( \text{dom}^\infty \).

Any complex object \( o \) necessarily belongs to some sort interpretation \( [[\tau]] \), in which case we say that \( o \) conforms to \( \tau \). (If \( o \) contains one or more empty collections, it is possible that \( o \) conforms to infinitely many sorts.) Object \( o \) is a tuple if \( \tau \) is a tuple sort, and a collection if \( \tau \) is a collection sort.

**Example 5** Consider the following objects. Our convention throughout this thesis will be to use both integers and lowercase letters near the beginning of the alphabet to denote atomic elements (i.e., constants).

\( o_1 := \{ a, b, 0 \} \)
\( o_2 := \{ \{ a, 1 \}, \{ a, 2 \}, \{ b, 2 \}, \{ b, 3 \} \} \)
\( o_3 := \{ \{ a \}, \{ 2, 5 \}, \{ a \}, \{ 3, 4 \}, \{ \{ b \}, \{ 3, 4 \} \} \} \)
\( o_4 := \{ \{ 1 \}, \{ 2 \}, \{ 1 \}, \{ 3 \} \}; \{ \} \}
\( o_5 := \{ \{ 1 \}, \{ 1 \}, \{ 2 \}; \{ 1 \}, \{ 2 \}; \{ 1 \}, \{ 3 \}, \{ 3 \} \} \}
\( o_6 := \{ 1, \{ a, c \}; \{ 2, \{ a, c \} \} \}
\( o_7 := \{ \{ a \}, \{ 1 \}, \{ 2 \} \}; \{ b, \{ 1 \}, \{ 3 \} \}; \{ c, \{ \} \} \}
\( o_8 := \{ \{ \} \}
\( o_9 := \{ \{ \{ \{ a \}, \{ c \}, \{ c \} \}; \}
\qquad \{ \{ a \}, \{ c \}, \{ c \} \}; \}
\qquad \{ \{ a, c \} \}
\}
\( o_{10} := \{ \}
\( o_{11} := \{ a, \{ a, a, c \}; c, \{ a, c \} \}
\)

Objects \( o_1 \)–\( o_9 \) are complex objects because they conform to sorts \( \tau_1 \)–\( \tau_9 \), respectively, from Example 4. Object \( o_{10} \) conforms to any sort rooted by a set, including \( \tau_2, \tau_3, \tau_5, \tau_6, \tau_7, \) and \( \tau_9 \); therefore it is also a complex object. Finally, although \( o_{11} \) satisfies our informal characterization from the start of Section 2.1, it is not formally a complex object (as per Definition 2.1.5) because the elements of the set are not uniformly typed and so \( o_{11} \) does not conform to a sort.
We will not consider further untyped objects such as \( o_{11} \) in the above example. Henceforth, when we refer to an “object” the reader should interpret this to mean a “complex object” as defined above. We also represent objects as trees, analogous to the representation of sorts in Figure 2.1. At times we may refer to the “nodes” or “leaves” of an object or a sort, by which we mean the nodes or leaves of the tree representation.

Branching, Non-Branching and Chain Objects  We classify complex objects as either branching or non-branching. An object is non-branching if it belongs to the interpretation of a non-branching sort; otherwise it is branching. For example, consider the objects \( o_1 - o_{10} \) from Example 5 and the sorts \( \tau_1 - \tau_9 \) from Example 4. Objects \( o_1 - o_7 \) are all non-branching because they conform to \( \tau_1 - \tau_7 \), respectively, which are all non-branching sorts. Object \( o_8 \) is a branching object because any sort to which it conforms is necessarily branching, including \( \tau_8 \). Similarly, \( o_9 \) is a branching object because any sort to which it conforms is necessarily branching, including \( \tau_9 \). Object \( o_{10} \) is a non-branching object even though it conforms to the branching sort \( \tau_9 \)—it also conforms to the non-branching sort \( \tau_2 \) (among others).

We say that a complex object is a chain if there exists a chain sort to which it conforms. Objects \( o_1, o_2, o_4, o_5 \), and \( o_{10} \) are chain objects because they conform to the chain sorts \( \tau_1, \tau_2, \tau_4, \tau_5 \), and \( \tau_2 \), respectively.

Complete, Incomplete, and Trivial Objects  We say that a complex object \( o \) is complete if it does not contain an empty collection. We say that a complex object \( o \) is trivial if it is either an empty collection or a tuple of trivial objects. For example, consider complex objects \( o_1 - o_{10} \) from Example 5 (recall that \( o_{11} \) is not a complex object). Objects \( o_1 - o_6 \) are complete because they do not contain empty collections, while objects \( o_7 - o_{10} \) are incomplete. Object \( o_{10} \) is trivial.

2.1.2 Object Linearization

In this section we describe a transformation from an object of arbitrary sort to a chain object that is either trivial or complete.\(^1\) This transformation—which we call linearization—takes place in two steps:

1. If the object is non-trivial, complete it to remove empty collections.
2. Remove branching via a Cartesian product of siblings.

In the case where the original object was a collection object that does not contain any 0-ary tuples, this transformation is injective (i.e., one-to-one). We will use object linearization at the end of this chapter to reduce the problem of testing equivalence of queries returning arbitrary complex objects to the equivalence of queries returning trivial or complete chain objects.

\(^1\)Our reason for handling trivial objects as a special case will become clear in Section 2.2 when we encode objects as flat relations. Trivial objects correspond to an empty encoding relation.
Object Completion

The first step in linearizing an object is to make it complete (cf. Section 2.1.1) by padding with placeholder nodes. Object completion is necessary to ensure that the second step remains lossless.

Given an object $o$ that conforms to sort $\tau$, we define the completion of $o$ with respect to $\tau$, denoted $\text{COMPLETE}_\tau(o)$, as the new object created from $o$ by the following process. If $o$ is either trivial or already complete, then $\text{COMPLETE}_\tau(o) = o$. Otherwise, let $o'$ be any descendant empty collection within $o$, and let $\tau'$ be the descendant sort within $\tau$ corresponding to $o'$. We know that $\tau'$ is a collection sort, and so let $\tau''$ be the child sort of $\tau'$ (i.e. $\tau'$ is a collection of $\tau''$s). Then, we insert into the empty collection $o'$ a “shaded copy” of $\tau''$. Repeating this process until $o$ does not contain any more empty collections, we obtain $\text{COMPLETE}_\tau(o)$.

Example 6 Consider the (non-trivial) incomplete objects $o_7$, $o_8$, and $o_9$ from Example 5 that conform to sorts $\tau_7$, $\tau_8$, and $\tau_9$, respectively, from Example 4. Figure 2.2 illustrates the completion of each object with respect to its associated sort.

Object $\text{COMPLETE}_\tau(o)$ may contain “shaded” delimiter pairs $[||]$, $\{\}$, $\state{dom}$, or $\state{dom}$; it also may contain the symbol $\state{dom}$ which is used as an atomic value. If $\text{COMPLETE}_\tau(o)$ contains any of these symbols, it is technically not a complex object (as per Definition 2.1.5). Hence, we want to extend our definition of complex objects slightly to allow them to contain both shaded delimiter pairs and the atomic value $\state{dom}$. Our intention is that shaded delimiter pairs retain the same semantics as their unshaded counterparts, but affect object equality (that is, two objects with
identical structure but different shading of nodes are not considered equal). To accomplish this, we modify our interpretation of sorts (Definition 2.1.4) as follows.

\[
[[\text{dom}]] = \text{dom} \cup \{\text{dom}\}
\]

\[
[[\{\tau\}]] = \{\{v_1, \ldots, v_j\} \mid j \geq 0, v_i \in [[\tau]], i \in [1, j]\}
\cup \{\{v_1, \ldots, v_j\} \mid j \geq 0, v_i \in [[\tau]], i \in [1, j]\}
\]

\[
[[\{\tau_1, \ldots, \tau_k\}]] = \{\{v_1, \ldots, v_k\} \mid v_j \in [[\tau_j]], j \in [1, k]\}
\cup \{\{v_1, \ldots, v_k\} \mid v_j \in [[\tau_j]], j \in [1, k]\}
\]

The interpretations \([[\{\tau\}]]\) and \([[\{\tau, \tau\}]]\) are modified analogously to \([[\{\tau\}]]\). Under this modified interpretation of sorts, completion with respect to sort \(\tau\) is a function

\[
\text{COMPLETE}_\tau: [[\tau]] \rightarrow [[\tau]]
\]

and so, by Definition (2.1.5), \(\text{COMPLETE}_\tau(o)\) is a complex object.

**Observation 2.1.6** Given an initial object \(o\) that does not contain any shaded nodes and a sort \(\tau\) to which \(o\) conforms, object \(\text{COMPLETE}_\tau(o)\) obeys the following properties:

1. The root node is always unshaded.
2. The only shaded atomic value is \(\text{dom}\).
3. No unshaded node occurs beneath a shaded node.
4. Any unshaded node that has a shaded child must be a collection type.
5. Any collection node
   - (a) has at least one child node;
   - (b) has at most one shaded child node; and
   - (c) cannot have both shaded and unshaded child nodes.

Finally, we note that completion with respect to a sort \(\tau\) is both idempotent and invertible (by deleting shaded nodes).

**Removing Branching**

Once an object \(o\) is completed, the next step in linearizing it is to transform it into a chain object. A recursive procedure for performing this transformation is shown in Algorithm 1. The critical step in calculating \(\text{CHAIN}(o)\) is the invocation of \(\text{DISTRIBUTE}\) on line 10 to remove tuple sorts with arity of two or more. The \(\text{DISTRIBUTE}\) function combines two chain objects \(o_a\) and \(o_b\) into a single chain object by distributing \(o_b\) across each leaf tuple \(t\) in \(o_a\) and then distributing the atomic values from \(t\) across the leaves of its copy of \(o_b\).

Algorithm 1 can be applied to any complete complex object. By temporarily treating the symbol \(\text{dom}\) as an atomic value, any sort \(\tau\) can be viewed as a complete complex object (\(\tau\) conforms to itself, and \(\text{COMPLETE}_\tau(\tau) = \tau\)), allowing Algorithm 1 to be applied to sorts. This leads us to the following observation.
Algorithm 1 Transforming objects into chains

CHAIN(o)
▷ Input: complete or trivial object o
▷ Output: chain object formed from o
1   if o is atomic
2      then return \{ o \}
3   elseif o = [ o_1, \ldots, o_n ] \hspace{1em} \triangleright \text{Where } [ \cdot ] \text{ is any (possibly shaded) collection type}
4      then return [ CHAIN(o_1), \ldots, CHAIN(o_n) ] \hspace{1em} \triangleright \text{preserving collection type } [ \cdot ]
5   elseif o = \{ \} or o = \{ \}
6      then return \{ \}
7   elseif o = \{ o_1 \} or o = \{ o_1 \}
8      then return CHAIN(o_1)
9   elseif (o = \{ o_1, \ldots, o_n \}) or o = \{ o_1, \ldots, o_n \} and n > 1
10     then return Distribute(CHAIN(o_1), CHAIN(\{ o_2, \ldots, o_n \}))

Distribute(o_a, o_b)
▷ Input: chain object o_a of sort $\langle \frac{a}{k}, k \rangle$
         Assume that o_a is a tree whose m leaves are
         the k-ary tuples \{ a_{i1}^1, \ldots, a_{ik}^1 \}, \ldots, \{ a_{i1}^m, \ldots, a_{ik}^m \}
▷ Input: chain object o_b of sort $\langle \frac{b}{l}, l \rangle$
         Assume that o_b is a tree whose n leaves are
         the l-ary tuples \{ b_{j1}^1, \ldots, b_{jl}^1 \}, \ldots, \{ b_{j1}^n, \ldots, b_{jl}^n \}
▷ Output: chain object of sort $\langle \frac{a}{k} \cdot \frac{b}{l}, k + l \rangle$
         formed by distributing o_b over each leaf of o_a
         and pushing down atomic values
1   o ← copy of o_a
2   foreach $i \in [1, m]$
3      do $o^i ←$ copy of o_b
4         foreach $j \in [1, n]$
5            do substitute tuple \{ a_{i1}^j, \ldots, a_{ik}^j, b_{j1}^j, \ldots, b_{jl}^j \}
                 for tuple \{ b_{j1}^1, \ldots, b_{jl}^1 \} within $o^i$
6               substitute $o^i$ for tuple \{ a_{i1}^j, \ldots, a_{ik}^j \} within o
7   return o
Observation 2.1.7 If \( o \) is a complete object conforming to sort \( \tau \), then \( \text{Chain}(o) \) is a complete chain object conforming to sort \( \text{Chain}(\tau) \).

Definition 2.1.8 (Object Linearization) Given an object \( o \) conforming to sort \( \tau \), the linearization of \( o \) with respect to \( \tau \) is the complete chain object given by the following function \( \text{Linearize}_\tau : \tau \rightarrow \text{Chain}(\tau) \).

\[
\text{Linearize}_\tau(o) := \text{Chain} \circ \text{Complete}_\tau(o)
\]  

(2.4)

Given an arbitrary sort \( \tau \), we may at times refer to the chain sort \( \text{Chain}(\tau) \) as the linearization of \( \tau \). This is consistent with Definition 2.1.8, because if we treat \( \tau \) as an object of sort \( \tau \) to be linearized, we deduce the following.

\[
\text{Linearize}_\tau(\tau) = \text{Chain} \circ \text{Complete}_\tau(\tau) = \text{Chain}(\tau)
\]

Example 7 Sorts \( \tau_7 \), \( \tau_8 \), and \( \tau_9 \) are shown in Figure 2.3 along with their linearizations \( \text{Chain}(\tau_7) \), \( \text{Chain}(\tau_8) \), and \( \text{Chain}(\tau_9) \). Collection types have been numbered to help illustrate the effect of Algorithm 1. Observe that in each case the ordering of collections in the chain corresponds to a pre-order traversal of the original sort.

Example 8 Figure 2.4 shows the linearization of objects \( o_7 \), \( o_8 \), and \( o_9 \) with respect to sorts \( \tau_7 \), \( \tau_8 \), and \( \tau_9 \). The reader is encouraged to step through the application of Algorithm 1 to the completed objects in Figure 2.2 to yield the objects shown here.
Figure 2.4: Linearization of objects $o_7$, $o_8$, and $o_9$
With certain restrictions (to be explained shortly), Algorithm 1 has a retraction (i.e., it is reversible). The retraction operation depends upon the sort of the original object; we don’t go into detail because we never have need to perform the retraction operation. It suffices to point out that from the result of \(\text{Distribute}(o_a, o_b)\) we can reconstruct both \(o_a\) and \(o_b\) because either both are trivial chain objects (in which case \(\text{Distribute}(o_a, o_b)\) is also a trivial chain object), or neither contains an empty collection and so the output has redundancy akin to a relational multi-value dependency. There is a minor difficulty when dealing with shaded tuples because lines 6, 8, and 10 of procedure \(\text{Chain}\) discard the delimiters of the original tuples (and hence the bit of information denoting whether a tuple is shaded). For example, in the transformation of \(\text{COMPLETE}_{r_0}(o_9)\) (cf. Figure 2.2) to \(\text{LINEARIZE}_{r_0}(o_9)\) (cf. Figure 2.4) thirteen pairs of tuple delimiters were discarded, three of which were shaded. If the original object contained arbitrarily-shaded tuples, then information has been lost. However, if we assume that all shaded nodes in \(\text{COMPLETE}_{\tau}(o)\) were inserted by the completion of \(o\) with respect to some sort \(\tau\), and \(\tau\) does not contain any zero-ary tuple sorts, then the properties in Observation 2.1.6 are sufficient for deducing which tuple delimiters require shading.

**Theorem 2.1.9** Given sort \(\tau\) not containing any zero-ary tuple sorts, \(\text{LINEARIZE}_{\tau}\) has a retraction.

**Corollary 2.1.10** Given sort \(\tau\) not containing any zero-ary tuple sorts, any two objects \(o, o' \in [[\tau]]\) satisfy the following.

\[ o = o' \iff \text{LINEARIZE}_{\tau}(o) = \text{LINEARIZE}_{\tau}(o') \]

Finally, we note that the properties in Observation 2.1.6 may be violated by application of Algorithm 1. For example, linearized objects \(\text{LINEARIZE}_{r_0}(o_8)\) and \(\text{LINEARIZE}_{r_0}(o_9)\) in Figure 2.4 both violate the first property by having unshaded nodes occurring beneath shaded nodes.

**Observation 2.1.11** Given an initial object \(o\) that does not contain any shaded nodes and a sort \(\tau\) to which it conforms, linearized object \(\text{LINEARIZE}_{\tau}(o)\) obeys the following properties:

1. The root node is always unshaded.
2. The only shaded atomic value is \(\text{dom}\).
3. Any collection node
   (a) has at least one child node;
   (b) has at most one shaded child node; and
   (c) cannot have both shaded and unshaded child nodes.
4. Tuple nodes are never shaded.

The first three properties in Observation 2.1.11 hold because they hold in the input object \(\text{COMPLETE}_{\tau}(o)\) (by Observation 2.1.6) and are preserved by Algorithm 1 (although preservation of the third property is not obvious and relies upon
\( \text{COMPLETE}_r(o) \) satisfying all of the properties in Observation 2.1.6. The fourth property holds because procedure \text{CHAIN} deletes all tuple nodes from its input and inserts unshaded tuples of atomic values at the leaves.

### 2.1.3 Object-Constructing Queries

In this subsection we propose the “Conjunctive Object-Constructing Query Language” \( \text{COCQL}^2 \) which is intended to model a specific subclass of SQL-92 [63] queries containing aggregation. As such, \( \text{COCQL}^2 \) uses a data model based on “tables,” which differ from traditional relations both by allowing duplicate tuples and by allowing attribute values to range over complex objects. After first defining the data model we then describe \( \text{COCQL} \), which is an algebraic language for specifying functions (queries) that map a database of flat tables to a complex object. We also define a subclass of \( \text{COCQL} \) that is incapable of constructing non-trivial incomplete objects (i.e., objects containing empty nested collections), which we will denote as \( \text{COCQL}^2^- \).

**Data Model**

Let \( \text{aname} \) and \( \text{tname} \) denote countably infinite sets of attribute names and table names, respectively, disjoint from each other and from the underlying domain \( \text{dom} \). (Our convention will be to use uppercase letters to denote elements of \( \text{aname} \) and \( \text{tname} \).) Our data model is defined as follows.

**Definition 2.1.12 (Table Schema)** A table schema of arity \( k \) has the form

\[
T(A_1 : \tau_1, \ldots, A_k : \tau_k)
\]

for some \( k \geq 0 \), where \( T \) is a table name from \( \text{tname} \), \( A_1, \ldots, A_k \) are distinct attribute names from \( \text{aname} \), and \( \tau_1, \ldots, \tau_k \) are sorts. If \( \tau_1, \ldots, \tau_k \) are all \( \text{dom} \), then we say that the table schema is flat.

**Definition 2.1.13 (Relation Schema)** A relation schema is a flat table schema.

**Definition 2.1.14 (Table)** A table \( T \) is a pair \( (S(T), T(T)) \) where \( S(T) \) is a table schema \( T(A_1 : \tau_1, \ldots, A_k : \tau_k) \) and \( T(T) \) is a finite bag-instance of \( S(T) \).

\[
T(T) \in [\{ \{ \tau_1, \ldots, \tau_k \} \}] \]

We call \( T \) flat if \( S(T) \) is a flat schema.

**Definition 2.1.15 (Relation)** A relation \( R \) is a table with a flat schema and whose instance \( S(R) \) does not contain duplicate tuples. In other words, if \( S(R) \) has arity \( k \), then

\[
T(R) \in [\{ \{ \text{dom}, \ldots, \text{dom} \} \}]^k
\]

\(^2\)Pronounced “cockle".
We often omit the sorts and write a table schema as \(T(A_1, \ldots, A_k)\). If necessary, we use \(\text{sort}(A_i)\) to denote sort \(\tau_i\) corresponding to attribute \(A_i\) (within the schema of table \(T\), which will be obvious from the context). We use \(\text{dom}(A_i)\) to denote the domain of values for attribute \(A_i\),

\[
\text{dom}(A_i) := [[[\text{sort}(A_i)]]] = [[[\tau_i]]] \tag{2.6}
\]

and we use \(\text{adom}(A_i, T)\) to denote the active domain of attribute \(A_i\) within \(T\), which is the finite subset of \(\text{dom}(A_i)\) that appears within the projection of attribute \(A_i\) from table instance \(T(T)\) (when table instance \(T(T)\) is obvious from the context, we simply write \(\text{adom}(A_i)\)). Similarly, given a sequence \(\overline{A} = A_{i_1} \ldots A_{i_j}\) of \(j\) (not necessarily distinct) attributes of \(T\), we use \(\text{sort}(\overline{A})\) to denote the sort of \(\tau_j\) tuples that can be projected from \(T\), with \(\text{dom}(\overline{A})\) denoting the corresponding domain of tuple values,

\[
\text{sort}(\overline{A}) := \{\tau_{i_1}, \ldots, \tau_{i_j}\} \tag{2.7}
\]

\[
\text{dom}(\overline{A}) := \{\{v_{i_1}, \ldots, v_{i_j}\} | \{v_{i_1}, \ldots, v_{i_j}\} \in [[[\text{sort}(\overline{A})]]] \land \forall l, m \in [1, j](A_{i_l} = A_{i_m} \implies v_{i_l} = v_{i_m})\} \tag{2.8}
\]

and we use \(\text{adom}(\overline{A}, T)\) (or \(\text{adom}(\overline{A})\)) to denote the active domain of \(\overline{A}\) within table instance \(T(T)\). Finally, we use \(\text{sort}(S(T))\) to denote the sort of tuples within \(T\)

\[
\text{sort}(S(T)) := \text{sort}(A_1 \ldots A_k) = \{\tau_1, \ldots, \tau_k\} \tag{2.9}
\]

with \(\text{dom}(S(T))\) denoting the domain of possible tuples.

\[
\text{dom}(S(T)) := \text{dom}(A_1 \ldots A_k) = [[[\{\tau_1, \ldots, \tau_k\}]]] = [[[\text{sort}(S(T))]]] \tag{2.10}
\]

Equations 2.8 and 2.10 are consistent because \(A_1, \ldots, A_k\) are distinct names.

Observe from Definition 2.1.14 that each table instance \(T(T)\) is a bag of tuples, which is a complex object. However, our data model does not define “nested tables” per se, because table schemas are not recursive and so attribute names are only assigned to the elements of the outermost tuple—a distinction that becomes important for the design of our query language.

**Definition 2.1.16 (Relational Database)** A relational database \(D\) is a finite set of table names

\[
\{T_1, \ldots, T_m\} \in \text{tname}
\]

that have associated with each table name \(T_i\) a relation \((S(T_i), T(T_i))\). We call the tables in \(D\) base tables (or base relations); the set of base table schemas are collectively called the database schema, denoted \(S(D)\).

We define \(\text{adom}(D) \subset \text{dom}\) to be the active domain of database \(D\), which is the set of values that occur within any attribute of any tuple within \(D\).
COQL\textsuperscript{\textregistered} and COQL\textsuperscript{\textregistered}- Specification

Traditional conjunctive queries can be specified by relational algebra expressions combining base relations with the operators selection, projection, Cartesian product (or join), and attribute renaming (sometimes merged into the projection operator) [6, Ch. 4]. Standard approaches for extending relational algebra to bags of tuples include introducing two variations of projection (namely, duplicate-preserving and duplicate-eliminating), or defining projection as duplicate-preserving and introducing a new duplicate-elimination operator [42]. Well-formed algebra queries do not contain variable names; atomic values are only referenced either explicitly (as constants) or via attribute names occurring within the schema of the operator’s input. As its name implies, COQL\textsuperscript{\textregistered} is intended as an extension of conjunctive queries that allows for the construction of complex objects.

**Definition 2.1.17 (COQL\textsuperscript{\textregistered} Query)** A COQL\textsuperscript{\textregistered} query \( Q \) is an expression of the form

\[
\text{Eval}(T, f, \mathbf{E}(T))
\]

where

- \( T \) is an algebraic expression composed of the operators in Figure 2.5;
- \( f \in \mathcal{F} \) is an aggregation function that takes a table as input and returns a complex object as output;
- \( \mathbf{E}(T) \) is a query evaluation environment; and
- \( \text{Eval}() \) is the query evaluation function which evaluates algebra expression \( T \) within the context of \( \mathbf{E}(T) \) to yield the table to which \( f \) is applied.

Given a relational database \( D \), the result of evaluating \( Q \) over \( D \) — denoted \( (Q)^D \) — is a complex object of sort \( \text{sort}(Q) := \text{sort}(f(T)) \).

If for every database instance \( D \) the object \( (Q)^D \) is trivial, then \( Q \) is called unsatisfiable; otherwise \( Q \) is called satisfiable. If the environment \( \mathbf{E}(T) \) is empty, then \( Q \) is called unparameterized; otherwise \( Q \) is called parameterized.

Although a COQL\textsuperscript{\textregistered} query contains an algebraic expression, it also contains some additional non-algebraic syntax. These non-algebraic constructs lead to several features that distinguish COQL\textsuperscript{\textregistered} from the usual language of relational algebra expressions over bags.

1. **Environment Variables:** Associated with the evaluation of table-generating expression \( T \) is an environment \( \mathbf{E}(T) \), which is a set of assignments of the form

\[
\mathbf{E}(T) = \{ V_1 \leftarrow v_1, \ldots, V_n \leftarrow v_n \}
\]

for some \( n \geq 0 \), where for each \( i \in [1, n] \), \( V_i \) is a distinct name from \textit{name} with which a sort \( \text{sort}(V_i) \) is implicitly associated, and \( v_i \in \text{dom}(V_i) = [\text{sort}(V_i)] \). Environment \( \mathbf{E}(T) \) is inherited down the algebraic operator tree so that any reference to \( V_i \) anywhere within expression \( T \) is interpreted as a reference to the value \( v_i \).
<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Query Expression</th>
<th>Output Table T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Relation</td>
<td>$T(A_1, \ldots, A_k)$ where $A_1, \ldots, A_k \in \text{aname}$ are “fresh” attribute names</td>
<td>$T(T) \in D$</td>
</tr>
<tr>
<td>Cartesian Product</td>
<td>$T_1 \times T_2$</td>
<td>$T(A_1, \ldots, A_{k_1}, B_1, \ldots, B_{k_2}) \ { (t_1[A_1], \ldots, t_1[A_{k_1}], t_2[B_1], \ldots, t_2[B_{k_2}] \mid t_1 \in T_1, t_2 \in T_2 }$</td>
</tr>
<tr>
<td>Selection</td>
<td>$\sigma_{A \bowtie B}(T_1)$ where $\text{sort}(A) = \text{sort}(B) = \text{dom}$</td>
<td>$T(A_1, \ldots, A_{k_1}) \ { (t_1[A_1], \ldots, t_1[A_{k_1}] \mid t_1 \in T_1 }$</td>
</tr>
<tr>
<td>Duplicate-Preserving Projection</td>
<td>$\Pi_{A_{i_1}, \ldots, A_{i_k}}^\text{dup}(T_1)$ where $\text{sort}(A_{i}) = \ldots = \text{sort}(A_{i_k}) = \text{dom}$</td>
<td>$T(A_{i_1}, \ldots, A_{i_k}) \ { (t_1[A_{i_1}], \ldots, t_1[A_{i_k}] \mid t_1 \in T_1 }$</td>
</tr>
<tr>
<td>Duplicate-Eliminating Projection</td>
<td>$\Pi_{A_{i_1}, \ldots, A_{i_k}}^\text{dup}(T_1)$ where $\text{sort}(A_{i}) = \ldots = \text{sort}(A_{i_k}) = \text{dom}$</td>
<td>$T(A_{i_1}, \ldots, A_{i_k}) \ { (t_1[A_{i_1}], \ldots, t_1[A_{i_k}] \mid t_1 \in T_1 }$</td>
</tr>
<tr>
<td>Scalar Subquery</td>
<td>$\lambda_X f(T_2)(T_1)$ where $X \in \text{aname}$ is a “fresh” attribute name and $f \in \mathcal{F}$ where $\text{sort}(X) = \text{sort}(f(T_2))$</td>
<td>$T(A_1, \ldots, A_{k_1}, X) \ { (t_1[A_1], \ldots, t_1[A_{k_1}], \ \text{EVAL}(T_2, f, \text{E}(T)) \cup { A_1 \leftarrow t_1[A_1], \ldots, A_{k_1} \leftarrow t_1[A_{k_1}] } \mid t_1 \in T_1 }$</td>
</tr>
<tr>
<td>Conjunctive Selection</td>
<td>$\sigma_{p_{i_1} \bowtie p_{i_2}}(T_1)$</td>
<td>$\sigma_{p_2}(\sigma_{p_1}(T_1))$</td>
</tr>
<tr>
<td>Join</td>
<td>$T_1 \bowtie_{p_{i}} T_2$</td>
<td>$\sigma_{p}(T_1 \times T_2)$</td>
</tr>
<tr>
<td>Generalized Projection</td>
<td>$\Pi_{A_{i_1}, \ldots, A_{i_k}}^{f(T_{i_1}, \ldots, T_{i_k})}(T_1)$ where $X \in \text{aname}$ is a “fresh” attribute name and $\text{sort}(A_{i}) = \ldots = \text{sort}(A_{i_k}) = \text{dom}$</td>
<td>$\lambda_X f(T_{i_1}^{\text{dup}}, \ldots, T_{i_k}^{\text{dup}}(\sigma_{X_{i_1} \leftarrow A_{i_1} \wedge \ldots \wedge X_{i_k} \leftarrow A_{i_k}}(T_1)))(\Pi_{A_{i_1}, \ldots, A_{i_k}}(T_1))$ where $T_1'$ is a copy of $T_1$ with schema $T_1'(A_{i_1}', \ldots, A_{i_k}')$</td>
</tr>
</tbody>
</table>

Figure 2.5: COCQL$^\mathcal{F}$ algebra operators
2. **Aggregation Functions**: We assume a set \( \mathcal{F} \) of functions that each aggregate a table into a complex object. Given function \( f \in \mathcal{F} \) and a table (or table-generating expression) \( T \), we use \( \text{sort}(f(T)) \) to denote the sort of object \( f(T) \). For this thesis we will restrict our attention to the case where \( \mathcal{F} \) only contains three functions

\[
\mathcal{F} = \{ f_b, f_n, f_s \} \tag{2.11}
\]

that return the bag, normalized bag, and set projections, respectively, of the collection of tuples in the instance of table \( T \). That is:

\[
\begin{align*}
    f_b(T) & := \text{Bag}(T(T)) \quad \text{sort}(f_b(T)) = \{ \text{sort}(S(T)) \} \\
    f_n(T) & := \text{NormBag}(T(T)) \quad \text{sort}(f_n(T)) = \{ \text{sort}(S(T)) \} \\
    f_s(T) & := \text{Set}(T(T)) \quad \text{sort}(f_s(T)) = \{ \text{sort}(S(T)) \}
\end{align*}
\]

3. **Query Composability**: A \( \text{COCQL}^\mathcal{F} \) query \( Q = \text{Eval}(T, f, E(T)) \) takes as inputs a set of tables and outputs a complex object of sort \( \text{sort}(Q) = \text{sort}(f(T)) \). Hence, \( \text{COCQL}^\mathcal{F} \) is not a fully-composable query language, per se. However, the sub-language of algebraic expressions is fully-composable, and arbitrary nesting of queries is made possible by the \( \lambda \) algebraic operator described below.

The algebra operators for specifying expression \( T \) are defined in Figure 2.5, using the following notational conventions:

1. We use \( T_1 \) and \( T_2 \) to denote algebra expressions yielding tables with schema \( T_1(A_1, \ldots, A_{k_1}) \) and \( T_2(B_1, \ldots, B_{k_2}) \), respectively.
2. We use \( \{ t_1 \mid t_1 \in T_1 \} \) to denote a bag constructed by letting tuple variable \( t_1 \) range over tuple occurrences in \( T(T_1) \).\(^3\) This requires an inherently procedural (i.e. variable-binding) interpretation of \( t_1 \in T_1 \) that differs from the usual interpretation of \( t_1 \in T_1 \) as a predicate.
3. Given some \( A \in \text{aname} \cup \text{dom} \) and a binding of tuple variable \( t_1 \in T_1 \) to value \( t_1 = \langle a_1, \ldots, a_{k_1} \rangle \), we use \( t_1[A] \) to denote the following complex object:\(^3\)
   - If \( A = A_i \) for some \( i \in [1, k_1] \), then \( t_1[A] := a_i \).
   - If \( A \in \text{aname} \) but is not an attribute in \( S(T_1) \), then \( E(T) \) must contain some assignment \( A \leftarrow v \) and \( t_1[A] := v \).
   - If \( A \in \text{dom} \), then \( t_1[A] := A \) and \( \text{dom}(A) := \text{dom} \).

The definitions in Figure 2.5 for the core algebraic operators \textit{base relation}, \textit{Cartesian product}, \textit{selection}, and \textit{projection} (duplicate-preserving and duplicate-eliminating) are all fairly standard, as are the syntactic extensions \textit{conjunctive selection} and \textit{join}. In a minor break from tradition, we completely ignore the attribute names in the database schema and instead embed attribute renaming into the base relation operator—this avoids the need for attribute renaming at intermediate stages of the query. With one caveat, all of these algebraic operators are

---

\(^3\)Analogous for \( t_2 \in T_2 \).
blind to the sort of their attributes and so treat $\text{dom}^\infty$ as an abstract atomic domain.
The exception is a syntactic restriction on the selection, duplicate-eliminating projection, and generalized projection operators to prevent the (explicit or implicit) comparison of higher-order objects.

Example 9 Consider the following simple student enrolment database $D_1$.

<table>
<thead>
<tr>
<th>Prof</th>
<th>cid</th>
<th>Student</th>
<th>Enrol</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td></td>
<td>sid</td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following $\text{COCQL}^\{f_s, f_a, f_n\}$ query $Q_7$ returns the set of pairs $(p, s)$ such that professor $p$ teaches a course in which student $s$ is enrolled.

$$Q_7: \quad \text{Eval}(\Pi_{B,D}^{dup}(\text{Course}(A, B) \bowtie_{A=C} \text{Enrol}(C, D)), f_s, \emptyset)$$

The reader can verify that $Q_7$ returns the following sort

$$\text{sort}(Q_7) = \{ \{ \text{dom}(B), \text{dom}(D) \} \} = \{ \{ \text{dom}, \text{dom} \} \} = \tau_2 \quad \text{from Example 4}$$

and that $Q_7^{D_1}$—the result of evaluating $Q_7$ over database $D_1$—yields the following complex object.

$$Q_7^{D_1} = \{ \{ a, 1 \}, \{ a, 2 \}, \{ b, 2 \}, \{ b, 3 \} \} = o_2 \quad \text{from Example 5}$$

The most common method for extending relational algebra to handle aggregation is to explicitly define a generalized projection operator [42, 54], which is a unary operator that first partitions the input table and then applies an aggregation function to each partition. (If grouped tables are explicitly supported by the data model, generalized projection can be split into separate partition and aggregation steps [63, 92].) In our algebra, aggregation is instead handled by the scalar subquery operator, which is a pseudo-binary operator. A true binary relational algebra operator specifies a logical operation that can be performed on two independent inputs. In contrast, our scalar subquery operator requires a nested loops semantics in which algebra expression $T_2$—the “(correlated) subquery”—is re-evaluated for each tuple in $T_1$ to yield a new complex object $f(T_2)$; we use the symbol $\lambda$ to emphasize the obvious similarity to a lambda-calculus expression. Our $\lambda$ operator is similar to the $\text{Apply}$ operator that Galindo-Legaria and Joshi introduce to handle correlated subqueries [38, 39], but whereas $\text{Apply}$ returns a table for the subquery which is then crossed with $T_1$, $\lambda$ always returns a single complex object from the subquery. We define the generalized projection operator in Figure 2.5 as a syntactic extension that is specified in terms of $\lambda$. The following example illustrates the scalar subquery operator; the reader is referred to Example 13 for an example of generalized projection.
Example 10  Assuming the same database $D_1$ as in Example 9, consider the following COCQL$_{f_b,f_s,f_n}$ query $Q_8$ which returns the sets of students who have been taught by each professor.

$$Q_8: \text{Eval}(\lambda_{f_b, f_s, f_n}(\Pi^{\text{top}}_{B\rightarrow E}(\text{Course}(A, B) \land \text{Enrol}(C, D))))(\text{Prof}(E), f_b, \emptyset)$$

The reader can verify that $Q_8$ returns the following sort

$$\text{sort}(Q_8) = \{\{\text{dom}(E), \text{dom}(F)\}\} = \{\{\text{dom}, \{\text{dom}\}\}\} = \tau_7 \quad \text{from Example 4}$$

and that $Q_8^{D_1}$ yields the following complex object.

$$Q_8^{D_1} = \{\{a, \{\{1\}, \{2\}\}\}, \{b, \{\{1\}, \{3\}\}\}, \{c, \{\}\}\} = \sigma_7 \quad \text{from Example 5}$$

Query $Q_8$ is able to generate non-trivial incomplete object $\sigma_7$ because it is explicitly written using a scalar subquery. If the query were written to use generalized projection instead of a scalar subquery, professor $c$ would not appear in the output.

As mentioned in the above example, the scalar subquery operator is required in order to construct non-trivial incomplete objects. A very practical subclass of COCQL$^F$ queries is those that are known to always output complete objects.

**Definition 2.1.18 (COCQL$^F$- Query)** The language $\text{COCQL}^{F-}$ is the subclass of $\text{COCQL}^F$ queries that do not contain an explicit scalar subquery operator $\lambda$. Queries in $\text{COCQL}^{F-}$ may contain generalized projection.

We emphasize that none of the algebraic operators have the ability to look inside a complex object. All knowledge of the internal structure of objects is encapsulated within the aggregation functions in $F$, while the query algebra simply treats $\text{dom}^\infty$ as an abstract domain. This is why we christen $\text{COCQL}^F$ as an “object-constructing” query language, rather than an “object query language.”

**Relating COCQL$^F$ and SQL**

There are three main differences between the data model of the SQL standard [63] and the data model we have defined for the query language COCQL$^F$. First, the SQL data model allows the base tables stored by the RDBMS to contain duplicates, whereas COCQL$^F$ queries are expressed over databases containing relations. Second, the SQL data model designates a particular atomic value—NULL—to represent “unknown” data, and requires comparisons between atomic elements to be evaluated using three-valued logic. In contrast, we have assumed a standard boolean interpretation of the equality operator over elements of $\text{dom}$ (the underlying domain for the COCQL$^F$ data model). Third, SQL tables only allow atomic-valued attributes, whereas COCQL$^F$ queries can construct arbitrary complex-valued attributes. As
previously mentioned, this third difference is relatively minor since the \textit{COCQL} algebraic operators cannot look inside a complex-valued attribute, and so \textit{COCQL} essentially treats \textit{dom} as an atomic domain.

\textit{SQL} and \textit{COCQL} also differ in the aggregation functions that they support. For comparison purposes, let \textit{SQL} denote an “\textit{SQL}-like” query language operating over the data model we have defined for \textit{COCQL} (i.e. complex-valued table attributes; no \textsc{null} values; and databases containing relations) and in which the usual \textit{SQL} aggregation functions (\textsc{SUM}, \textsc{COUNT}, \textsc{AVG}, etc.) have been replaced with the set of functions \textit{F}. Furthermore, let \textit{COCQL}, denote the slight extension of the language \textit{COCQL} in which the aggregate function \textit{f} may be used in the root-level invocation of \textsc{Eval}, even if \textit{f} \notin \textit{F} (clearly \textit{COCQL}, = \textit{COCQL} if \textit{f} \in \textit{F}).

\textbf{Proposition 2.1.19} \textit{COCQL}, corresponds to a restricted class of queries that can be expressed in \textit{SQL} using only the following constructs:

1. \textsc{SELECT} clauses composed of attribute names, constants, scalar subqueries, and aggregation expressions of the form \(f(A_1, \ldots, A_k), f \in \textit{F};\)
2. \textsc{FROM} clauses composed of a list of base relation names and nested \textit{SQL} table expressions;
3. \textsc{WHERE} clauses composed of conjunctions of equality comparisons, with each comparison containing only constants and/or references to attributes originating from base relations; and
4. \textsc{GROUP BY} clauses composed only of constants and attributes originating from base relations.

We denote this subset of \textit{SQL} expressions as \textit{CSQL} (for “Conjunctive \textit{SQL}”).

\textbf{Proposition 2.1.20} \textit{COCQL}, corresponds to the restricted class of \textit{CSQL} queries in which

- every nested \textsc{SELECT-FROM-WHERE-GROUPBY} block that contains an aggregation function in the \textsc{SELECT} clause also has as \textsc{GROUPBY} clause, and
- the \textsc{SELECT} clause does not contain scalar subqueries

(i.e., the subset of \textit{CSQL} that permits scalar aggregation only at the root block). We denote this subset of \textit{CSQL} expressions as \textit{CSQL}.

A \textit{COCQL}, query always returns a single complex object (treated as a scalar value from abstract domain \textit{dom}). In contrast, a \textit{CSQL} query always returns a table which—per Definition 2.1.14—is a table schema paired with a bag-object. Corresponding to any \textit{CSQL} query is a \textit{COCQL}, query whose root invocation of \textsc{Eval} applies the function \textit{f} to construct the same bag-object as the table instance returned by the \textit{CSQL} query.

\textbf{Example 11} Assume that a database schema contains relation \textit{Person}(id, \textit{fname}, \textit{lname}, phone). \textit{CSQL} query \(Q_9\) returns a table of (\textit{fname}, phone) pairs for persons with the last name DeHaan.
Query $Q_{10}$ shows a corresponding COCQL$^{\mathcal{F}, f_b}$ query which returns a bag of binary tuples.

$$Q_{10}: \quad \text{Eval}(\Pi^{\text{dup}}_{B,D}(\sigma_{C='DeHaan'}(\text{Person}(A,B,C,D))), f_b, \emptyset)$$

Given an arbitrary COCQL$^{\mathcal{F}}$ (or COCQL$^{\mathcal{F}, f_b}$) query whose root invocation of Eval applies the function $f$, the translation of the query into CSQL$^{\mathcal{F}}$ depends upon $f$. If $f = f_b$, then the corresponding CSQL$^{\mathcal{F}}$ query returns a table whose instance is the same bag-object as the output of the COCQL$^{\mathcal{F}}$ query. If $f \neq f_b$, then the corresponding CSQL$^{\mathcal{F}}$ query uses scalar aggregation in the root block to apply the function $f$ and the resulting object is wrapped within a single-row, single-column table.

Example 12 Assuming the same Person relation as the previous example, COCQL$^{\mathcal{F}}$ query $Q_{11}$ returns the result of applying an arbitrary function $f \in \mathcal{F}$ to a collection of (fname, phone) pairs for persons with the last name DeHaan.

$$Q_{11}: \quad \text{Eval}(\Pi^{\text{dup}}_{B,D}(\sigma_{C='DeHaan'}(\text{Person}(A,B,C,D))), f, \emptyset)$$

If $f = f_b$, then the corresponding CSQL$^{\mathcal{F}}$ query is $Q_9$ from Example 11. Otherwise, the corresponding CSQL$^{\mathcal{F}}$ query is the scalar aggregation query $Q_{12}$ below.

$$Q_{12}: \quad \text{SELECT } f(\text{fname}, \text{phone}) \text{ FROM Person WHERE lname = 'DeHaan' }$$

We have defined the COCQL$^{\mathcal{F}}$ generalized projection operator as a syntactic extension for a correlated scalar subquery. In contrast, the semantics of the GROUP BY construct in SQL are usually described in terms of partitioning—and for good reason, because physical implementations of grouping/partitioning are typically much more efficient than nested-loop-based evaluation of correlated subqueries. Nevertheless, a logical rewriting of GROUP BY in terms of correlated subqueries is also possible within CSQL$^{\mathcal{F}}$ as illustrated in the next example.

Example 13 Using the same Person(id, fname, lname, phone) relation as the previous example, consider CSQL$^{\mathcal{F}}$ query $Q_{13}$, which returns the result of applying arbitrary aggregation function $f \in \mathcal{F}$ to the collection of first names for each group of persons with the same last name. This corresponds to COCQL$^{\mathcal{F}, f_b}$ query $Q_{14}$.

$$Q_{13}: \quad \text{SELECT } f(\text{fname}) \text{ FROM Person GROUP BY lname }$$

$$Q_{14}: \quad \text{Eval}(\Pi^{\text{dup}}_{B}(\Pi^{E=f(B)}_{C}(\text{Person}(A,B,C,D))), f_b, \emptyset)$$

Expanding the generalized projection in $Q_{14}$ yields COCQL$^{\mathcal{F}, f_b}$ query $Q_{15}$, which corresponds to CSQL$^{\mathcal{F}}$ query $Q_{16}$. 

32
\[ Q_{15}: \text{EVAL}(\Pi_{E}(\lambda_{E}(\Pi_{d}(\sigma_{C}(A,B,C,D))))(\Pi_{C}(\text{Person}(A,B,C,D))))), \]

\[ Q_{16}: \text{SELECT} (\text{SELECT} f(fname) \]
\[ \text{FROM Person} \]
\[ \text{WHERE lname = t.lname} ) \]
\[ \text{FROM (SELECT lname FROM Person GROUP BY lname) AS t} \]

CQL\textsuperscript{F} queries \(Q_{13}\) and \(Q_{16}\) are equivalent; however, this equivalence relies upon the fact that attribute \textit{lname} does not contain \texttt{NULL} values (as per our definition of language CQL\textsuperscript{F}). If the data model were extended with \texttt{NULL} values, the \texttt{WHERE} clause of the subquery in \(Q_{16}\) would need minor adjustment to account for SQL’s three-valued logic.

Although we do not prove it formally here, the correspondence between \texttt{COCQL}\textsuperscript{F}\texttt{,fb} and CQL\textsuperscript{F} posited in Proposition 2.1.19 can be verified by

1. defining a canonical form for \texttt{COCQL}\textsuperscript{F}\texttt{,fb} algebra expressions,
2. demonstrating the correspondence between CQL\textsuperscript{F} expressions and \texttt{COCQL}\textsuperscript{F}\texttt{,fb} expressions in canonical form, and
3. showing that every \texttt{COCQL}\textsuperscript{F}\texttt{,fb} can be converted to the canonical form by a sequence of equivalence-preserving algebraic transformations.

The correspondence between \texttt{COCQL}\textsuperscript{F}\texttt{,fb} and CQL\textsuperscript{F} posited in Proposition 2.1.20 can be verified in a similar fashion.

**Deciding Equivalence of CQL\textsuperscript{F} Queries**

In the introduction to this thesis we purport to study the equivalence of SQL queries containing nested aggregation functions. Having just introduced the algebraic query language \texttt{COCQL}\textsuperscript{F} and shown its relationship to a specific class of SQL queries, we briefly define formally what it means for two \texttt{COCQL}\textsuperscript{F} queries to be equivalent.

**Definition 2.1.21 (COCQL\textsuperscript{F} Query Equivalence)** Two \texttt{COCQL}\textsuperscript{F} queries \(Q, Q'\) over database schema \(S(D)\) with the same output sort \(\tau\) are equivalent—denoted \(Q \equiv Q'\)—if for each relational database \(D\) with schema \(S(D)\) and finite instance \(T(D)\) the complex objects \((Q)D\) and \((Q')D\) are equal.

**Proposition 2.1.22** The query equivalence problems for \texttt{COCQL}\textsuperscript{F}\texttt{,fb} and CQL\textsuperscript{F} are polynomial-time reducible to each other.

**Proposition 2.1.23** The query equivalence problems for \texttt{COCQL}\textsuperscript{F}\texttt{,fb} and CQL\textsuperscript{F} are polynomial-time reducible to each other.

Much of the remainder of this thesis concerns how to test \texttt{COCQL}\textsuperscript{F} query equivalence. The main goal of the next two sections is to reduce \texttt{COCQL}\textsuperscript{F} query equivalence to testing a special form of equivalence between queries that return flat relations.
2.2 Encoding Objects as Relations

Section 2.1.2 presented a process, called linearization, for transforming an arbitrary complex object into a complete chain object that possibly contains “shaded” nodes. In this section we describe a method for converting between a complete chain object and its representation as a set of flat tuples. In Section 2.2.1 we only consider complete chain objects that do not have shaded nodes—a restriction that is lifted in Section 2.2.2. Combined with Section 2.1.2, this provides a method for encoding arbitrary objects as a single relation. (This encoding technique is similar to one used previously by Suciu [102]; see Section 2.4.2 for a brief comparison.) Although each object has a unique linearization, the same cannot be said for relational encodings—there are an infinite number of different encodings of the same object. Therefore, Section 2.2.3 defines a binary relationship between encoding relations that characterizes when two different relations encode the same object.

2.2.1 Encoding and Decoding Chain Objects

Figure 2.6 illustrates the basic technique for encoding a complete chain object as a set of tuples, showing both the complete chain object $o_4$ from Example 5 and its encoding as relation $R_1$. Starting with the tree representation of the object, we label each outgoing edge from a collection node with a locally-unique value (shown) and then create a tuple for each root-to-leaf path down the tree, incorporating the edge labels as attribute values. Note that edge labels could be drawn from a composite domain that corresponds to multiple relational attributes. For example, relation $R_2$ shows an alternative encoding that could be formed by replacing labels $y_1$ and $y_2$ with $y\cdot a$ and $y\cdot b$, respectively. We now give a more formal definition of encoding relations.

Definition 2.2.1 (Encoding Schema) An encoding schema of depth $d \geq 0$ is a relation schema $R(A_1, \ldots, A_k)$ annotated with the following additional information:

- For each $i \in [1, d]$, the “tuple of index attributes at level $i$” $\overline{I}_i$ is a sequence $\overline{I}_i := I_{i,1} \cdots I_{i,k_i}$
where \( k_i \geq 0 \) and for \( j \in [1, k_i] \) each \( I_{i,j} \in \{A_1, \ldots, A_k\} \). Because sequences allow repeated elements, we use \( I_i \) to denote the set of attributes occurring in \( I_i \). For convenience, we use \( I_{[i,j]} \) to denote the sequence \( I_i, I_{i+1}, \ldots, I_j \), and \( I_{[i,j]} \) to denote the corresponding set \( \bigcup_{i \in I_{[i,j]}} I_i \).

- The “tuple of output attributes” \( \vec{V} \) is a sequence

\[
\vec{V} := V_1 \ldots V_{k_v}
\]

where \( k_v \geq 0 \) and for \( j \in [1, k_v] \) each \( V_j \in \{A_1, \ldots, A_k\} \). We use \( \mathcal{V} \) to denote the set of output attributes occurring in \( \vec{V} \).

Furthermore, the sets of index and output attributes must together cover all the attributes of relation \( R \).

\[
I_1 \cup \ldots \cup I_d \cup \mathcal{V} = \{A_1, \ldots, A_k\} \tag{2.12}
\]

**Definition 2.2.2 (Encoding Relation)** An encoding relation \( R \) of depth \( d \) is a relation whose schema is an encoding schema of depth \( d \) and whose instance satisfies the following constraints.

1. \( T(R) \) satisfies the functional dependency \( \overline{I}_{[1,d]} \rightarrow \mathcal{V} \).
2. If \( d = 0 \), then \( T(R) \) contains precisely one tuple.

For convenience, we use the notation

\[
R(\overline{I}_1; \ldots; \overline{I}_d; \mathcal{V})
\]

to represent the schema of encoding relation \( R \). When depicting an encoding relation graphically, we separate index levels with a single vertical line and separate output variables from index variables with a double vertical line. For example, Figure 2.6 depicts encoding relation \( R_1 \) with schema \( R_1(X; Y; Z) \), while Figure 2.7 depicts an encoding relation \( R_3 \) with schema\(^4\) \( R_3(A, B; C, D; C) \).

Given an encoding relation \( R = (S(R), T(R)) \) and a value \( \bar{a} \in \text{dom}(\overline{I}_{[1, l-1]}), R \) we use \( R[\bar{a}] \) to denote the sub-relation indexed by \( \bar{a} \), which is an encoding relation \((S(R[\bar{a}]), T(R[\bar{a}]))\) defined as

\[
S(R[\bar{a}]) := R(\overline{I}_1; \ldots; \overline{I}_d; \mathcal{V}) \tag{2.13}
\]

\[
T(R[\bar{a}]) := \Pi_{\overline{I}_{[1,l]}}(\sigma_{\overline{I}_{[1,l-1]}=\bar{a}}(R)) \tag{2.14}
\]

where \( \Pi \) and \( \sigma \) are relational projection and selection, respectively. Intuitively, if \( R \) encodes an object of depth \( d \), then \( R[\bar{a}] \) encodes a sub-object of depth \( d - l + 1 \) that

---

\(^4\)At times we may refer to encoding relation \( R \) as having “relation schema \( R(\overline{I}_{[1,d]}; \mathcal{V}) \)” or containing “tuples over \( \text{dom}(\overline{I}_{[1,d]}; \mathcal{V}). \)” This is technically incorrect because tables (and hence relations) are required to have distinct attribute names, while the sequence \( \overline{I}_{[1,d]}; \mathcal{V} \) allows repetitions. For example, \( R_3 \) in Figure 2.7 appears to be a quinary relation with schema \( R_3(ABCDC) \), but it is properly a quaternary relation over the attributes \( \{A, B, C, D\} \). This distinction is moot, however, because we can treat \( R_3 \) as a set of quinary tuples over \( \text{dom}(\overline{I}_{[1,d]}; \mathcal{V}) = \text{dom}(ABCDC) \) since agreement of values for repeated attributes is guaranteed by equation 2.8.

35
occurs as a descendant within the object encoded by $R$. For example, Figure 2.7 also shows the encoding relations $R_3[a_2b_1]$ and $R_3[a_2b_1d_3]$ derived from $R_3$.

We say that an encoding relation and a signature are compatible if the length of the signature matches the depth of the encoding schema. Each combination of an encoding relation and a compatible signature decodes to a unique complete chain object. Given an encoding relation $R$ of depth $d$ and a signature $\bar{s}$ of length $d$, the function $\text{DecodeSimple}(R, \bar{s})$ shown in Algorithm 2 computes a complete chain object of sort $\bar{s}$. If $R$ is non-empty, then $\text{DecodeSimple}(R, \bar{s})$ returns a hierarchy of $d$ (possibly zero) levels of nested collections with tuples of arity $\bar{V}$ at the leaves. If $R$ is empty, then Definition 2.2.2 requires that $d > 0$, and so $\text{DecodeSimple}(R, \bar{s})$ returns a single empty collection of type $\bar{s}$._1. The structuring of Algorithm 2 into two procedures $\text{DecodeSimple}$ and $\text{InnerDecodeSimple}$ is not necessary, but it facilitates our extending of the algorithm to handle shading in the next subsection.

Example 14 Decoding relation $R_1$ (cf. Figure 2.6) with signature $\text{ns}$ yields object $o_4$.

\[
\text{DecodeSimple}(R_1, \text{ns}) = \text{InnerDecodeSimple}(R_1, \text{ns})
\]
\[
= \{ \text{InnerDecodeSimple}(R_1[x_1], s), \text{InnerDecodeSimple}(R_1[x_2], s), \text{InnerDecodeSimple}(R_1[x_3], s) \}
\]
\[
= \{ \text{InnerDecodeSimple}(R_1[x_1y_1], \varnothing), \text{InnerDecodeSimple}(R_1[x_2y_1], \varnothing), \text{InnerDecodeSimple}(R_1[x_2y_2], \varnothing) \}, \text{InnerDecodeSimple}(R_1[x_3y_1], \varnothing), \text{InnerDecodeSimple}(R_1[x_3y_2], \varnothing) \}
\]

Figure 2.7: Encoding relation $R_3$ and select sub-relations
Algorithm 2 Reconstructing an encoded chain object

**DecodeSimple**($R, \bar{s}$)

▷ **Input:** encoding relation $R(\bar{x}_1; \ldots; \bar{x}_d; \bar{V})$
▷ **Input:** compatible signature $\bar{s}$
▷ **Output:** complete or trivial chain object of sort $(\bar{s}, [\bar{V}])$

1. if $R$ is non-empty
2. then return **InnerDecodeSimple**($R, \bar{s}$)
3. else ▷ $d > 0$ and $R$ encodes a trivial empty collection
4. if $\bar{s} = s\bar{Y}$
5. then return $\{\}$
6. elseif $\bar{s} = b\bar{Y}$
7. then return $\emptyset$
8. else ▷ $\bar{s} = n\bar{Y}$
9. return $\emptyset$

**InnerDecodeSimple**($R, \bar{s}$)

▷ **Input:** non-empty encoding relation $R(\bar{x}_1; \ldots; \bar{x}_d; \bar{V})$
▷ **Input:** compatible signature $\bar{s}$
▷ **Output:** complete or trivial chain object of sort $(\bar{s}, [\bar{V}])$

1. if $d = 0$ ▷ $R$ contains singleton tuple $\{a_1, \ldots, a_{|\bar{V}|}\}$
2. then return $\{a_1, \ldots, a_{|\bar{V}|}\}$
3. elseif $\bar{s} = s\bar{Y}$
4. then return $\{**InnerDecodeSimple**(R[\bar{a}], \bar{V}) \mid \bar{a} \in \Pi_{\bar{X}_1}(R)\}$
5. elseif $\bar{s} = b\bar{Y}$
6. then return $\emptyset$
7. elseif $\bar{s} = n\bar{Y}$
8. return $\emptyset$ $**InnerDecodeSimple**(R[\bar{a}], \bar{V}) \mid \bar{a} \in \Pi_{\bar{X}_1}(R)$
We started this subsection by informally describing how to generate an encoding relation from a complete chain object (not containing shaded nodes), using \( o_4 \) and \( R_1 \) as an example. Clearly, our method could generate an infinite number of unique encoding relations for \( o_4 \) simply by choosing different edge labels; however, all of these relations would be isomorphic to \( R_1 \) modulo renaming of index values. Because there is a one-to-one mapping between tuples in \( R_1 \) and the leaves of \( o_4 \), there cannot exist another relation that contains fewer tuples than \( R_1 \) and still decodes to \( o_4 \); hence, \( R_1 \) could be considered a “minimal encoding” of \( o_4 \). There are also infinitely many encodings of \( o_4 \) that are not isomorphic to \( R_1 \) and hence contain varying amounts of redundancy with respect to sort \( \tau_4 \).

Example 15 Relation \( R_3 \) in Figure 2.7 also yields object \( o_4 \) when decoded with signature \( ns \).

\[
\text{DECODESIMPLE}(R_3, ns) = \emptyset \{(1),(2)\},\{(1),(2)\},\{(1),(1),(2)\},
\{(1),(2)\},\{(1),(3),(3)\},\{(1),(3),(3)\} \emptyset
\]

\[
= \emptyset \{(1),(2)\},\{(1),(2)\},\{(1),(2)\},
\{(1),(2)\},\{(1),(3),(3)\},\{(1),(3)\} \emptyset
\]

\[
= \emptyset \{(1),(2)\},\{(1),(2)\},\{(1),(3)\} \emptyset
\]

\[
= o_4
\]

The reader can verify that when relation \( R_3 \) is instead decoded with signature \( sb \) it yields object \( o_5 \).

### 2.2.2 Encoding and Decoding Linearized Objects

The *linearization* process of Section 2.1.2 outputs complete chain objects that may contain special “shaded” nodes (cf. Figure 2.4). The encoding and decoding procedures described in the previous subsection must be extended if we want to reconstruct linearized objects. From Observation 2.1.11 we know that the root node of a linearized object is never shaded; therefore, we can store information about a node’s shading within the label we assign to its *incoming* edge.

One approach would be to use an extra boolean attribute for each index level of an encoding schema to indicate shading. An alternative would be to assume the existence of a new domain of “shaded labels” distinguishable from unshaded labels. Because labels are only required to be locally-unique (i.e. unique among siblings), we can improve upon this second approach— from Observation 2.1.11 we know that a node can have at most one shaded child node, and so we need only one shaded label.

We extend the definition of an encoding relation as follows. Let the symbol \( \square \) denote a designated atomic element not occurring in the atomic domain \( \text{dom} \).
Similar to how we extended the underlying domain of complex objects to allow for shaded nodes, we slightly extend the underlying domain of encoding relations \(^5\) to include the atomic value \(\equiv\). The encoding process described in Section 2.2.1 now requires two minor adjustments:

1. The label for any given edge should be composed solely of (one or more occurrences of) the \(\equiv\) symbol if and only if the edge leads into a shaded node.
2. Each occurrence of the atomic value \(\text{dom}\) should be replaced by \(\equiv\) within the output attributes of the encoding relation.

Example 16 Applying the modified encoding algorithm to objects \(\text{Linearize}_{R_T}(o_7)\), \(\text{Linearize}_{R_S}(o_8)\), and \(\text{Linearize}_{R_5}(o_9)\) from Figure 2.4 yields the encoding relations \(R_4\), \(R_5\), and \(R_6\), respectively, shown in Figure 2.8.

Algorithm 3 shows the modified decoding algorithm \(\text{Decode}\). The key modification is that in the recursive invocation to decode sub-relation \(R[\overline{a}]\), we test the predicate \(\overline{a} = \equiv\overline{a}\) which returns TRUE when \(\overline{a}\) is a tuple composed entirely of the \(\equiv\) symbol. The result of this test is passed to the decoding of \(R[\overline{a}]\) to indicate whether shading needs to be performed on the constructed sub-object. At the leaf level \((d = 0)\) this boolean parameter is ignored because tuple nodes of linearized objects are never shaded (see Observation 2.1.11); however, at this level the \(\equiv\) symbols within the output are translated into the atomic value \(\text{dom}\) that occurs within the leaves of linearized objects.

\(^5\)Technically, an encoding relation is defined as a special case of a flat table, and so we actually need to extend the underlying domain of tables. The important point is that \(\equiv\) may not occur within the base relations of a database, or as an explicit constant within an object-constructing COCQL\(^7\) query.
Algorithm 3 Reconstructing an encoded \textit{linearized} object

\textbf{DECODE}(R, \bar{\delta})

\textbf{Input:} encoding relation \(R(\bar{T}_1; \ldots; \bar{T}_d; \bar{V})\)
\textbf{Input:} compatible signature \(\bar{\delta}\)
\textbf{Output:} complete or trivial chain object of sort \((\bar{\delta}, |\bar{V}|)\), possibly containing shaded nodes

1. \textbf{if} \(R\) is non-empty
2. \textbf{then} \textbf{return} \textbf{INNERDECODE}(\(R, \bar{\delta}, \text{false}\))
3. \textbf{else}
4. \hskip 20pt \textbf{if} \(\bar{\delta} = s\bar{V}\)
5. \hskip 40pt \textbf{then} \textbf{return} \{ \}
6. \hskip 20pt \textbf{elseif} \(\bar{\delta} = b\bar{V}\)
7. \hskip 40pt \textbf{then} \textbf{return} \{ \}
8. \hskip 20pt \textbf{else} \(\bar{\delta} = n\bar{V}\)
9. \hskip 40pt \textbf{return} \{ \}

\textbf{INNERDECODE}(R, \bar{\delta}, \text{shade})

\textbf{Input:} non-empty encoding relation \(R(\bar{T}_1; \ldots; \bar{T}_d; \bar{V})\)
\textbf{Input:} compatible signature \(\bar{\delta}\)
\textbf{Input:} boolean \textit{shade} indicating if shading is required
\textbf{Output:} complete or trivial chain object of sort \((\bar{\delta}, |\bar{V}|)\), possibly containing shaded nodes

1. \textbf{if} \(d = 0\) \(\triangleright\) \(R\) contains singleton tuple \(\{a_1, \ldots, a_{|\bar{\delta}|}\}\)
2. \textbf{then} \(t \leftarrow \{a_1, \ldots, a_{|\bar{\delta}|}\}\)
3. \textbf{Replace each occurrence of} \(\equiv\) \textbf{within} \(t\) \textbf{with} \(\text{dom}\)
4. \textbf{return} \(t\)
5. \textbf{elseif} \(\bar{\delta} = s\bar{V}\)
6. \textbf{then if} \(\text{shade} = \text{true}\)
7. \hskip 20pt \textbf{return} \{ \ \text{INNERDECODE}(R[\bar{\alpha}], \bar{V}, (\bar{\alpha} = \equiv_{\bar{\delta}})) \mid \bar{\alpha} \in \Pi_{\bar{T}_1}(R) \} \}
8. \textbf{else return} \{ \ \text{INNERDECODE}(R[\bar{\alpha}], \bar{V}, (\bar{\alpha} = \equiv_{\bar{\delta}})) \mid \bar{\alpha} \in \Pi_{\bar{T}_1}(R) \} \}
9. \textbf{elseif} \(\bar{\delta} = b\bar{V}\)
10. \textbf{then if} \(\text{shade} = \text{true}\)
11. \hskip 20pt \textbf{return} \{ \ \text{INNERDECODE}(R[\bar{\alpha}], \bar{V}, (\bar{\alpha} = \equiv_{\bar{\delta}})) \mid \bar{\alpha} \in \Pi_{\bar{T}_1}(R) \} \}
12. \textbf{else return} \{ \ \text{INNERDECODE}(R[\bar{\alpha}], \bar{V}, (\bar{\alpha} = \equiv_{\bar{\delta}})) \mid \bar{\alpha} \in \Pi_{\bar{T}_1}(R) \} \}
13. \textbf{else} \(\bar{\delta} = n\bar{V}\)
14. \textbf{if} \(\text{shade} = \text{true}\)
15. \hskip 20pt \textbf{return} \{ \ \text{INNERDECODE}(R[\bar{\alpha}], \bar{V}, (\bar{\alpha} = \equiv_{\bar{\delta}})) \mid \bar{\alpha} \in \Pi_{\bar{T}_1}(R) \} \}
16. \textbf{else return} \{ \ \text{INNERDECODE}(R[\bar{\alpha}], \bar{V}, (\bar{\alpha} = \equiv_{\bar{\delta}})) \mid \bar{\alpha} \in \Pi_{\bar{T}_1}(R) \} \}
Example 17 The reader can verify the following for the relations shown in Figure 2.8.

\[ \text{DECODE}(R_4, bs) = \text{LINEARIZE}_{r_7}(o_7) \]
\[ \text{DECODE}(R_5, sbss) = \text{LINEARIZE}_{r_8}(o_8) \]
\[ \text{DECODE}(R_6, sbnsbn) = \text{LINEARIZE}_{r_8}(o_8) \]

Decoding relation \( R_7 \) in Figure 2.9 with signature \( sbss \) also yields object \( \text{LINEARIZE}_{r_8}(o_8) \).

\[ \text{DECODE}(R_7, sbss) = \text{INNERDECODE}(R_7, sbss, \text{false}) \]
\[ = \{ \text{INNERDECODE}(R_7[s_1], bs, \text{false}), \]
\[ \quad \text{INNERDECODE}(R_7[s_2], bs, \text{false}) \} \]
\[ = \{ \{ \text{INNERDECODE}(R_7[s_1 = w_1], s, \text{false}) \}, \]
\[ \quad \{ \text{INNERDECODE}(R_7[s_2 = v_1, w_1], s, \text{false}) \} \}
\[ = \{ \{ \{ \text{INNERDECODE}(R_7[s_1 = w_1, x_1], \O, \text{false}) \} \}, \]
\[ \quad \{ \{ \text{INNERDECODE}(R_7[s_2 = v_1, w_1, x_1], \O, \text{false}) \} \}, \]
\[ \quad \{ \text{INNERDECODE}(R_7[s_2 = v_1, x_1], \O, \text{false}) \}, \]
\[ \quad \{ \text{INNERDECODE}(R_7[s_2 = v_1], \O, \text{true}) \} \} \}
\[ = \{ \{ \{ \{ \text{dom}, \text{dom} \} \} \}, \]
\[ \quad \{ \{ \{ \text{dom}, \text{dom} \} \}, \{ \text{dom}, \text{dom} \} \} \} \}
\[ = \{ \{ \{ \{ \text{dom}, \text{dom} \} \} \}, \]
\[ \quad \{ \{ \{ \text{dom}, \text{dom} \} \} \} \} \}
\[ = \text{LINEARIZE}_{r_8}(o_8) \]

Algorithm 3 is clearly backwards-compatible with Algorithm 2. That is, for any encoding relation \( R \) and compatible signature \( \bar{s} \) such that \( R \) does not contain the symbol \( \equiv \), \( \text{DECODE}(R, \bar{s}) = \text{DECODESIMPLE}(R, \bar{s}) \). Therefore, we will not discuss Algorithm 2 and any future mention of a decoding algorithm is implicitly a reference to Algorithm 3.

### 2.2.3 Encoding-Equality

In the previous section we defined a decoding algorithm that takes an encoding relation and a compatible signature as inputs and returns a complete chain object.
Examples 14 and 15 illustrate two encoding relations $R_1$ and $R_3$ that, when decoded with signature $\mathbf{ns}$, each yield object $o_4$. We would like to recognize that $R_1$ and $R_3$ are in a sense “equal” because they encode the same object; however, this notion of equality is dependent upon the signature $\mathbf{ns}$ used in the decoding. When decoded with signature $\mathbf{sb}$, $R_3$ yields $o_5$ but $R_1$ does not.

**Definition 2.2.3 (Encoding-Equality)** Given two encoding relations $R$, $R'$ and a compatible signature $\mathbf{s}$, if

$$\text{Decode}(R, \mathbf{s}) = \text{Decode}(R', \mathbf{s})$$

then we say that $R$ and $R'$ are encoding-equal w.r.t. signature $\mathbf{s}$ (abbreviated $\mathbf{s}$-equal, and denoted $R \equiv_\mathbf{s} R'$).

We use $R \equiv_\mathbf{s} R'$ to denote $\mathbf{s}$-equality, as distinct from $R = R'$ which—as an extension of traditional relational equality—implies that $S(R)$ and $S(R')$ have identical arities of index attributes at each index level and of output attributes, and that $T(R) = T(R')$. Observe that $R \equiv_\mathbf{s} R'$ implies that the encoding schemas $S(R)$ and $S(R')$ both have depth $|\mathbf{s}|$ and the same arity of output variables; however, it does not require attribute names to be the same, nor does it require the arity of corresponding index levels to be the same. For example, $\mathbf{s}$-equality of encoding relations $R(\overline{Z}_1; \overline{V})$ and $R'(\overline{Z}_1, \overline{V})$ corresponds to relational equality of projections onto the output attributes—that is, $\Pi_{\overline{V}}(R) = \Pi_{\overline{V}}(R')$, but it does not imply $T(R) = T(R')$ because index values could differ.

While Definition 2.2.3 captures the desired semantics of $\mathbf{s}$-equality, reasoning with it is awkward due to the reliance on the `Decode` procedure. We next introduce a mechanism called a $\mathbf{s}$-certificate that allows us to characterize $\mathbf{s}$-equality in a more declarative fashion. A $\mathbf{s}$-certificate is essentially a proof that the decoding of two relations with signature $\mathbf{s}$ yields the same object.

**Definition 2.2.4 ($\mathbf{s}$-Certificate)** Given encoding relations $R$ and $R'$ and signature $\mathbf{s}$, a $\mathbf{s}$-certificate is a tree rooted by a node $n_{(R,R')}$ that proves $R \equiv_\mathbf{s} R'$. When $\mathbf{s}_1 = \mathbf{s}$ then $n_{(R,R')}$ is a set node; when $\mathbf{s}_1 = \mathbf{b}$ then $n_{(R,R')}$ is a bag node; when $\mathbf{s}_1 = \mathbf{n}$ then $n_{(R,R')}$ is a normalized bag node; and when $\mathbf{s} = \emptyset$ then $n_{(R,R')}$ is a tuple node.

**Definition 2.2.5 (Set Node)** A set node $n^s_{(R,R')}$ proves $R \equiv_{\mathbf{s}\overline{V}} R'$ for some $R$, $R'$, and $\overline{V}$. It contains a function $f : \text{dom}({\overline{Z}_1}, R') \rightarrow \text{dom}({\overline{Z}_1}, R)$ satisfying

$$\forall \overline{x} \in \text{dom}(\overline{Z}_1) \left[ (R[f(\overline{x})] \equiv_{\overline{Z}} R'[\overline{x}]) \land (\overline{x} = \overline{f(x)} \iff f(\overline{x}) = \overline{f'(x)}) \right] \quad (2.15)$$

and a function $f' : \text{dom}(\overline{Z}_1) \rightarrow \text{dom}(\overline{Z}_1)$ satisfying

$$\forall \overline{x} \in \text{dom}(\overline{Z}_1) \left[ (R[\overline{x}] \equiv_{\mathbf{s}\overline{V}} R'[f'(\overline{x})]) \land (\overline{x} = \overline{f(x)} \iff f'(\overline{x}) = \overline{f'(x)}) \right] \quad (2.16)$$

42
For each pair \((\overline{x}, \overline{x}')\) such that either \(\overline{x}' = f'(\overline{x})\) or \(\overline{x} = f(\overline{x}')\), node \(n^b_{(R,R')}\) has a child \(Y\)-certificate that proves \(R[\overline{x}] \models_Y R'[\overline{x}']\).

A set node needs to guarantee that each child object in one of the decodings matches a child object in the other decoding; it does this as follows. For each (non-empty) sub-relation \(R'[\overline{x}']\) of \(R'\), the first conjunct in formula 2.15 enforces that

\[
\text{Decode}(R'[\overline{x}'], Y) = \text{InnerDecode}(R'[\overline{x}'], Y, \text{false})
= \text{InnerDecode}(R[f(\overline{x}')], Y, \text{false})
= \text{Decode}(R[f(\overline{x}')], Y)
\]

for some (non-empty) sub-relation \(R[f(\overline{x}')]\) of \(R\). Observe that in the decoding of \(R\) and \(R'\) by Algorithm 3, function Decode is never invoked on either \(R'[\overline{x}']\) or \(R[f(\overline{x}')]\); the actual invocations would be \(\text{InnerDecode}(R'[\overline{x}'], Y, (\overline{x}' = \equiv_Y^1))\) and \(\text{InnerDecode}(R[f(\overline{x}')], Y, (f(\overline{x}') = \equiv_Y^1))\), respectively. Therefore, the second conjunct in formula 2.15 is needed to ensure that the predicates \(\overline{x}' = \equiv_Y^1\) and \(f(\overline{x}') = \equiv_Y^1\) agree, so that the roots of the decoded sub-objects are either both or neither shaded. Formula 2.15 thus guarantees \(\text{Decode}(R', sY) \subseteq \text{Decode}(R, sY)\). Formula 2.16 likewise guarantees \(\text{Decode}(R, sY) \subseteq \text{Decode}(R', sY)\).

**Definition 2.2.6 (Bag Node)** A bag node \(n^b_{(R,R')}\) proves \(R \equiv_Y R'\) for some \(R\), \(R'\), and \(Y\). It contains a bijective function \(f : \text{adom}(\overline{L}_1, R') \rightarrow \text{adom}(\overline{L}_1, R)\) satisfying

\[
\forall \overline{x}' \in \text{adom}(\overline{L}_1) \left[ (R[f(\overline{x}')] \equiv_Y R'[\overline{x}']) \land (\overline{x}' = \equiv_Y^1 \iff f(\overline{x}') = \equiv_Y^1) \right] \tag{2.17}
\]

For each pair \((\overline{x}, \overline{x}')\) such that \(\overline{x} = f(\overline{x}')\), node \(n^b_{(R,R')}\) has a child \(Y\)-certificate that proves \(R[\overline{x}] \models_Y R'[\overline{x}']\).

A bag node needs to guarantee that equal sub-objects occur with the same absolute frequency in both decodings. The bag node functions similarly to a set node; the main difference is that by requiring \(f\) to be a bijection between \(\text{adom}(\overline{L}_1, R)\) and \(\text{adom}(\overline{L}_1, R)\), we enforce that equal sub-objects of \(\text{Decode}(R, sY)\) and \(\text{Decode}(R', sY)\) can be paired up, guaranteeing identical absolute frequencies.

**Definition 2.2.7 (Normalized Bag Node)** A normalized bag node \(n^s_{(R,R')}\) proves \(R \equiv_{nY} R'\) for some \(R\), \(R'\), and \(Y\). It contains two finite domains \(D_1\) and \(D_2\), and two surjective functions \(\rho : \text{adom}(\overline{L}_1, R) \rightarrow D_1\) and \(\varphi : \text{adom}(\overline{L}_1, R') \rightarrow D_2\) that satisfy

\[
\forall p \in D_1, \forall q \in D_2 [(R)^p \equiv_{nY} (R')^q]
\]

where \((R)^p\) is the (non-empty) encoding relation defined by \(S((R)^p) := S(R)\) and \(T((R)^p) := \sigma_{(\overline{L}_1), \rho}(R)\), and \((R')^q\) is defined analogously using \(\varphi\). For each pair \((p, q)\) such that \(p \in D_1\) and \(q \in D_2\), node \(n^s_{(R,R')}\) has a child \(bY\)-certificate that proves \((R)^p \equiv_{bY} (R')^q\).
A normalized bag node needs to guarantee that equal sub-objects occur with the same relative frequency in both decodings. To understand how this is enforced, let $B_R$ and $B_{R'}$ denote the bags (of sub-objects) obtained by decoding relations $R$ and $R'$, respectively, with signature $bY$. Let $B \otimes n$ denote the $n$-expansion of bag $B$, which is the bag obtained from $B$ by multiplying the cardinality of each element by $n$ [23]. Definition 2.2.7 ensures that

$$B_R \otimes |D_2| = B_{R'} \otimes |D_1|$$

which implies that $B_R$ and $B_{R'}$ have the same normalized bag projection.

**Definition 2.2.8 (Tuple Node)** A tuple node $n^t_{(R,R')}$ proves $R \simeq R'$ for some $R$ and $R'$ with depth zero. By Definition 2.2.2, an encoding relation of depth zero contains precisely one tuple (of only output values). Therefore, node $n^t_{(R,R')}$ contains a single comparison showing that $R$ and $R'$ contain the same tuple.

The main result of this subsection is summarized by the next theorem, which is a result of how Definitions 2.2.3–2.2.8 have been constructed to mirror the semantics of Algorithm 3.

**Theorem 2.2.9** Given two encoding relations $R$ and $R'$ and a compatible signature $\bar{f}$, $R \simeq R'$ if and only if there exists a $\bar{f}$-certificate between $R$ and $R'$.

**Example 18** Consider encoding relations $R_1$ (cf. Figure 2.6) and $R_3$ (cf. Figure 2.7). Figure 2.10 depicts a (partial) ns-certificate proving that $R_1 \nsimeq R_3$.

Observe that the nodes within a $\bar{f}$-certificate are organized into levels of identical node type. For any index level $i$, if $\bar{f}_i = b$ then “the nodes in (bag) level $i$” are the equi-depth bag nodes that each prove $bY_{[i+1,d]}$-equality of some pair of sub-relations. Similarly, if $\bar{f}_i = s$ then “the nodes in (set) level $i$” are the equi-depth set nodes that prove $s_{[i+1,d]}$-equality. If $\bar{f}_i = n$ then “the nodes in (normalized bag) level $i$” are the equi-depth normalized bag nodes that prove $nY_{[i+1,d]}$-equality—note, however, that immediately underneath a normalized bag level is an extra layer of bag nodes proving $bY_{[i+1,d]}$-equality, and so when we refer to “nodes at level $i$” it is not the same as tree depth $i$. For example, in Figure 2.10 the “nodes at level 1” refers to the normalized bag node at the root (tree depth 0), while the “nodes at level 2” refers to the set nodes at tree depth 2. Finally, the leaves of a certificate are all equi-depth and guaranteed to be tuple nodes.

### 2.3 Encoding Queries

An “encoding query” is a query that outputs a (flat) encoding relation. The goal of this section is to reduce the problem of deciding $\text{COCQL}^\{f_b,f_s,f_n\}$ query equivalence (see Definition 2.1.21) to deciding “encoding-equivalence” between encoding queries. In
Figure 2.10: ns-certificate proving $R_1 \equiv_{as} R_3$
Section 2.3.1 we define a language for specifying encoding queries. In Section 2.3.2 we define a mapping from each \texttt{COCQL} query $Q$ to an encoding query $Q'$ such that over any relational database $D$, the output of $Q'$ is an encoding of the linearization of the object output by $Q$.

### 2.3.1 Hierarchical Conjunctive Encoding Queries

Our language for specifying encoding queries is based on the traditional rule-based syntax of conjunctive queries. Our primary extension is to allow the right-hand side of the rule to be a hierarchy of conjunctive query bodies. Although there is a connection between our hierarchical syntax and the well-known \textit{left outer join} algebra operator, we defer that discussion until Chapter 4 and restrict this section to defining our language for encoding queries. In the following definitions, let $D$ be a relational database with schema $S(D)$.

**Definition 2.3.1 (Conjunctive Query with Equality (CQ$^=$))** A conjunctive query with equality $Q$ is a rule of the form:

$$Q(\overline{W}) \triangleright R_1(\overline{X}_1), \ldots, R_m(\overline{X}_m), Y_1 = Z_1, \ldots, Y_n = Z_n$$

where $\overline{W}$ is a tuple composed of constants and variables, each $R_i$ is a relation name in $S(D)$, each $\overline{X}_i$ is a tuple composed of constants and variables, and each $Y_i$ and $Z_i$ is either a constant or a variable. When $n = 0$ query $Q$ is simply called a conjunctive query (CQ).

Expression $Q(\overline{W})$ is termed the query head, each expression $R_i(\overline{X}_i)$ is called an atom, and each expression $Y_i = Z_i$ is called a predicate. The body of $Q$—denoted $\text{body}_Q$—is the set of atoms $\text{atoms}_Q$ and predicates $\text{pred}_Q$ in $Q$. We define the following sets of constants and variables:

- $\mathcal{B}$ denotes the set of variables occurring anywhere in $Q$.
- $\mathcal{W}$ denotes the set of variables and constants occurring within head tuple $\overline{W}$.
- $\mathcal{C}$ denotes an infinite set of constants, and $\mathcal{C}_Q \subseteq \mathcal{C}$ is the set of constants occurring anywhere in $Q$.
- $\mathcal{P} \subseteq \mathcal{B}$ is a distinguished set of variables called the parameters.
- $\mathcal{L} := \mathcal{B} \setminus \mathcal{P}$ denotes the local variables. To ensure query safety, we require that each local variable must occur within at least one atom.

Query $Q$ is called \textit{parameterized} when $|\mathcal{P}| > 0$. Query $Q$ is called \textit{boolean} if $|\overline{W}| = 0$. Query $Q$ is called \textit{unsatisfiable} if $\text{pred}_Q$ transitive imply a predicate of the form $c_i = c_j$, where $c_i$ and $c_j$ are distinct constants in $\mathcal{C}$, can be derived from $\text{pred}_Q$ using transitivity. It is well-known that any parameter-free, satisfiable CQ with equality can be converted to an equivalent CQ (without equality) by a process of variable substitution. We therefore assume without loss of generality that that predicates within $\text{pred}_Q$ only reference constants and parameter variables.
An embedding of $\text{body}_Q$ into $D$—denoted $\gamma : \text{body}_Q \to D$, but for conciseness we often write $\gamma : Q \to D$—is a mapping $\gamma : B \to \text{dom}$ (extended with identity on the constants in $C$) such that $\gamma$ satisfies every atom in $\text{atoms}_Q$ and every predicate in $\text{pred}_Q$. Atom $R_i(X_i)$ is satisfied by $\gamma$ if relation $R_i \in D$ and tuple $\gamma(X_i) \in T(R_i)$. Predicate $Y_j = Z_j$ is satisfied by $\gamma$ if $\gamma$ maps $Y_j$ and $Z_j$ to the same element of $\text{dom}$.

The result of evaluating unparameterized conjunctive query $Q$ over database $D$—denoted $Q^D$—is a flat relation $(S(Q^D), T(Q^D))$. Schema $S(Q^D)$ is deduced from the query head by associating with each element of $\overline{W}$ a unique attribute name from aname. Instance $T(Q^D)$ is the set of tuples for which there exists a supporting embedding of $\text{body}_Q$ into $D$.

$$T(Q^D) := \{ \gamma(\overline{W}) \mid \gamma : Q \to D \}$$

(2.20)

If $Q$ is parameterized, then it can only be evaluated within the context of a specific assignment of values to the parameter variables. Let $v : P \to \text{dom}$ be any such assignment; then, the parameterized evaluation of $Q$ over $D$ relative to $v$ restricts the supporting embeddings to those that are extensions of $v$.

$$T(Q^D|_v) := \{ \gamma(\overline{W}) \mid \gamma : Q \to D \land \gamma(\overline{P}) = v(\overline{P}) \}$$

(2.21)

(Within equation 2.21, $\overline{P}$ denotes an arbitrary sequencing of set $P$ into a tuple.)

**Definition 2.3.2 (Hierarchical Conjunctive Query (HCQ))** A hierarchical conjunctive query $Q$ is a generalization of a parameter-free $CQ$, defined as follows.

1. $\text{body}_Q$ is a rooted (unordered) tree in which each vertex has a unique label, the root vertex contains an unparameterized $CQ$ body, and the non-root vertices each contain a parameterized $CQ^*$ body. For each vertex $v \in \text{body}_Q$, we use $\text{body}_v$ to denote the $CQ^*$ body occurring within vertex $v$, $\text{body}^\text{anc}_v$ to denote the $CQ^*$ body formed by conjoining the $CQ^*$ bodies of all of the ancestors of $v$, and $\text{body}^+_v$ to denote the $CQ^*$ body formed by conjoining $\text{body}_v$ with $\text{body}^\text{anc}_v$; the sets $\text{atoms}_v$, $\text{atoms}^\text{anc}_v$, $\text{atoms}^+_v$, $\text{pred}_v$, $\text{pred}^\text{anc}_v$, and $\text{pred}^+_v$, are defined analogously.

2. Sets $B$, $C_Q \in C$, $P$, and $L$ are defined the same as for $CQ^*$ s. Because the overall query $Q$ is parameter-free, set $P = \emptyset$ and therefore $L = B$.

3. We use $B_v$ to denote the set of variables occurring anywhere in $\text{body}_v$, with $B_v^\text{anc}$ and $B_v^+$ likewise corresponding to $\text{body}^\text{anc}_v$ and $\text{body}^+_v$. We use $P_v$ to denote the set of designated parameter variables in $\text{body}_v$, and we require that $P_v := B_v \cap B_v^\text{anc}$. Similarly, we use $L_v := B_v \setminus B_v^\text{anc}$ to denote the set of local variables of $\text{body}_v$, and for any two distinct vertices $u$, $v$ in $\text{body}_Q$, we require that local variables $L_u$ and $L_v$ be disjoint. As a consequence, for each variable

---

6Technically, the attribute names in a relation schema must be unique (per Definitions 2.1.12 and 2.1.13) and so if $\overline{W}$ contains repeated variables or constants, then each occurrence must be assigned a different attribute name. When discussing containment or equivalence of queries we will generally ignore attribute names, relying instead upon positional alignment of columns in the depiction of the schema or instance.
$Q_{17}(A, B, D, A, B, A) := \begin{array}{c}
\exists R(A, B) \\
\exists S(B, C) \quad \exists R(B, E) \\
\exists S(C, D)
\end{array}$

Figure 2.11: Hierarchical conjunctive query $Q_{17}$

- $X \in B$ there exists precisely one node $u$ such that $X \in L_u$, and $X$ only occurs within the subtree rooted at $u$. We use $\mathcal{P}_v^{\text{anc}}$ and $\mathcal{P}_v^+$ to denote the sets of "inter-query parameters" along the path from the root to $v$, with $\mathcal{P}_Q^+$ denoting the set of all inter-query parameters within $\text{body}_Q$.

4. The head tuple $\overline{W}$ is composed from elements of $\mathcal{B} \cup \mathcal{C} \cup \{\sqsubseteq\}$ (that is, variables, constants, and the special $\sqsubseteq$ constant). We use $\mathcal{W}$ to denote the set of elements occurring within $\overline{W}$.

5. Each component of head tuple $\overline{W}$ that is not the $\sqsubseteq$ constant is labelled with a vertex $v \in \text{body}_Q$. We write $W_i \sqsubseteq v$ to denote that the $i$th component of $\overline{W}$ is labelled with vertex $v$. If $W_i \in \mathcal{C}$, then $v$ is allowed to be any vertex in $\text{body}_Q$. If $W_i \in \mathcal{B}$, then $v$ must satisfy $W_i \in \mathcal{B}^v$.

6. We define the head label tuple $\overline{L}$ as the tuple with the same arity as head tuple $\overline{W}$ in which each position $L_i$ contains the label of the corresponding head component $W_i$ (or is empty if $W_i = \sqsubseteq$). We define the head label set $L$ as the set of labels occurring in sequence $\overline{L}$.

We again adopt the (non-restrictive) syntactic convention that for each vertex $v$ in $\text{body}_Q$ the equality predicates in $\text{body}_v$ only reference $\mathcal{C} \cup \mathcal{P}_v$ (that is, constants and variables that are parameters within $v$).

Example 19 Hierarchical conjunctive query $Q_{17}$ is shown in Figure 2.11. $Q_{17}$ has head label tuple $\overline{L} = (\sqsubseteq, \sqsubseteq, \sqsubseteq, \sqsubseteq, \sqsubseteq, \sqsubseteq)$ and head label set $L = \{\sqsubseteq, \sqsubseteq, \sqsubseteq, \sqsubseteq\}$.

The result of evaluating hierarchical conjunctive query $Q$ over database $D$ is a relation $Q^D = (\mathcal{S}(Q^D), \mathcal{T}(Q^D))$ whose instance may contain the special atomic value $\sqsubseteq$ not occurring in the domain of $D$ (see Section 2.2.2). Schema $\mathcal{S}(Q^D)$ is defined the same as for conjunctive queries. Similar to conjunctive queries, instance $\mathcal{T}(Q^D)$ is defined in terms of terminal embeddings of $\text{body}_Q$ into $D$ which generalize the definition of CQ embeddings given above.

Let $\text{trees}(T)$ denote the set of all non-empty trees that can be formed from some non-empty tree $T$ by deleting zero or more of its strict subtrees; note that
every tree in \( \text{trees}(T) \) contains at least the root of \( T \). We now restrict our attention to trees in the set \( \text{trees}(\text{body}_Q) \). For each such tree \( T \), define \( \mathcal{B}_T := \bigcup_{v \in T} \mathcal{B}_v \) and define \( \mathcal{L}_T, \text{atoms}_T, \) and \( \text{pred}_T \) analogously; it follows from Definition 2.3.2 that \( T \) is a parameter-free query body (i.e., \( \mathcal{L}_T = \mathcal{B}_T \)).

Given any \( T \in \text{trees}(\text{body}_Q) \), let \( \text{body}_T^+ \) denote the CQ\(^*\) body formed by conjoining the CQ\(^*\) bodies of all vertices within \( T \). An embedding of \( T \in \text{trees}(\text{body}_Q) \) into \( \mathbf{D} \)—denoted \( \gamma : T \rightarrow \mathbf{D} \)—is a mapping \( \gamma : \mathcal{B}_T \rightarrow \text{dom} \) (extended with identity on the constants in \( \mathcal{C} \)) that is an embedding of \( \text{body}_T^+ \) into \( \mathbf{D} \). Observe that in the degenerate case where \( \text{body}_Q \) is a single vertex, this definition of an embedding corresponds to the definition given earlier for CQ\(^*\)s. We use \( \text{body}_Q^+ \) to denote the single CQ\(^*\) formed by conjoining the CQ\(^*\)s from all of the vertices in \( \text{body}_Q \), which corresponds to the maximal case where \( T \) contains all of the vertices within \( \text{body}_Q \).

A terminal embedding of \( T \in \text{trees}(\text{body}_Q) \) into \( \mathbf{D} \)—denoted \( \gamma : T \rightarrow \mathbf{D} \)—is an embedding \( \gamma : T \rightarrow \mathbf{D} \) for which there does not exist some other tree \( T' \in \text{trees}(\text{body}_Q) \) and some other embedding \( \gamma' : T' \rightarrow \mathbf{D} \) such that \( \gamma' \) is an extension of \( \gamma \). Note that for \( \gamma' \) to be an extension of \( \gamma \), tree \( T' \) must be an extension of tree \( T \).

Given any embedding \( \gamma : T \rightarrow \mathbf{D} \) of some \( T \in \text{trees}(\text{body}_Q) \) and any component \( W_i \) of the head tuple \( \overline{W} \), the partial application of \( \gamma \) to \( W_i \)—denoted \( \hat{\gamma}(W_i) \)—is defined as follows.

\[
\hat{\gamma}(W_i) := \begin{cases} 
\gamma(W_i) & \text{if } \exists v \in T \text{ such that } W_i \text{ is labelled with } v \\
\equiv & \text{otherwise (includes case } W_i = \equiv) 
\end{cases}
\] (2.22)

Instance \( T(Q^D) \) is then defined as the set of tuples for which there exists a supporting partial application of some terminal embedding.

\[
T(Q^D) := \{ \hat{\gamma}(\overline{W}) \mid \exists T \in \text{trees}(\text{body}_Q), \gamma : T \rightarrow \mathbf{D} \} 
\] (2.23)

As a shorthand notation, we use \( \gamma : Q \rightarrow \mathbf{D} \) to denote that \( \gamma \) is a terminal embeddings of \( \text{body}_Q \), and so Equation 2.23 can be rewritten as follows.

\[
T(Q^D) := \{ \hat{\gamma}(\overline{W}) \mid \gamma : Q \rightarrow \mathbf{D} \} 
\] (2.24)

**Example 20** Consider evaluating HCQ \( Q_{17} \) from Figure 2.11 over a database instance \( \mathbf{D}_2 \) containing the following two relations \( R \) and \( S \).

\[
R \quad \begin{array}{cc}
\text{col}_1 & \text{col}_2 \\
1 & 2 \\
2 & 3 \\
2 & 4 \\
3 & 4 \\
3 & 5 \\
5 & 7 \\
6 & 8 \\
\end{array} 
\]

\[
S \quad \begin{array}{cc}
\text{col}_1 & \text{col}_2 \\
2 & 4 \\
4 & 5 \\
5 & 6 \\
7 & 8 \\
\end{array} 
\]
The set $\text{trees}(Q_{17})$ contains the six trees $T_1, \ldots, T_6$ shown in Figure 2.12. For each $i \in [1, 6]$, let $\Gamma^i$ denote the set of embeddings of $T_i$ into $D_2$. In Figure 2.13 we tabularly depict each $\Gamma^i$ and we have shaded rows corresponding to terminal embeddings. Finally, let $\Gamma$ denote the set of terminal embeddings of members of $\text{trees(body}_Q)$ into $D_2$. Relation $(Q_{17})^{D_2}$ is obtained by duplicate-eliminating projection of columns $A$, $b$, $D$, $A$, $-$, and $A$ from “relation” $\Gamma$ shown in Figure 2.14.

**Definition 2.3.3 (Encoding Query (CEQ or HCEQ))** A hierarchical conjunctive encoding query $Q$ of depth $d \geq 1$ is an HCQ in which the head tuple $\overline{W}$ is partitioned into $d + 1$ sub-tuples as follows:

$$Q(\overline{T}_1; \ldots; \overline{T}_d; \overline{V})$$

In the degenerate case where $\text{body}_Q$ is a single CQ body and $\overline{W}$ does not contain the special constant $\equiv$, we call $Q$ a conjunctive encoding query.

The result of evaluating encoding query $Q$ over database $D$ is a relation $Q^D = (S(Q^D), T(Q^D))$ whose schema $S(Q^D)$ is an encoding schema (see Definition 2.2.1) deduced by associating a unique attribute name from $\text{name}$ with each occurrence of a variable or constant in the query head. Instance $T(Q^D)$ is defined the same as for HCQs.

We call $\overline{V}$ the “output tuple” because it corresponds to the output attributes in encoding schema $S(Q^D)$; we use $\hat{V}$ to denote the set of labelled constants/variables composing $\overline{V}$, and we use $\mathcal{V}$ to denote the set of unlabelled constants/variables composing $\overline{V}$ (i.e., for each $X \in \mathcal{V}$ there exists some $v$ such that $X \equiv v$ occurs in both $\hat{V}$ and $\overline{V}$). Likewise, for each $i \in [1, d]$ we call $\overline{T}_i$ the “tuple of indexes at level $i$” because it corresponds to the tuple of index attributes at level $i$ in $S(Q^D)$; we use

![Figure 2.12: The six trees in $\text{trees}(Q_{17})$](image-url)
Figure 2.13: Embeddings of trees(Q_{17}) into database D_2

Figure 2.14: Terminal embeddings and their partial applications to W̄
\( \hat{\mathcal{I}}_i \) to denote the set of labelled constants/variables composing \( \hat{\mathcal{I}}_i \), and we use \( \mathcal{I}_i \) to denote the set of unlabelled constants/variables composing \( \hat{\mathcal{I}}_i \). We further define tuple \( \hat{\mathcal{I}}_{[i,j]} \) and sets \( \hat{\mathcal{I}}_{[i,j]} \) and \( \mathcal{I}_{[i,j]} \) as in Definition 2.2.1.

**Definition 2.3.4 (Validity)** We call encoding query \( Q \) valid if for any database \( D \) the relation \( Q^D \) is a valid encoding relation (i.e., satisfies Definition 2.2.2).

We are only interested in valid encoding queries because our definition of “encoding-equivalence” in the next sub-section presupposes that encoding queries always output encoding relations. The following lemma yields a sufficient condition for query validity that meets our needs for this chapter; we re-visit query validity in Chapters 3 and 5 when we address the problem of encoding-equivalence.

**Lemma 2.3.5** Encoding query \( Q \) is valid if it satisfies the following conditions.

1. For every component \( V_i\Box \) of \( \hat{V} \) where \( V_i \in \mathcal{B} \) (i.e., \( V_i \) is a variable), index set \( \hat{\mathcal{I}}_{[1,d]} \) contains \( V_i\Box \).
2. For every component \( V_i\Box \) of \( \hat{V} \) where \( V_i \in \mathcal{C} \) (i.e., \( V_i \) is a constant) and \( v \) is a non-root vertex of \( \text{body}_Q \), index set \( \hat{\mathcal{I}}_{[1,d]} \) contains an index labelled with vertex \( v \) (i.e., \( \exists X. (X\Box \in \hat{\mathcal{I}}_{[1,d]}) \)).

**Proof.** Suppose that \( Q \) satisfies the two conditions above. Since Definition 2.3.3 requires that \( d \geq 1 \), by Definition 2.2.2 we must show that for any database \( D \) the relation \( Q^D \) satisfies the functional dependency \( \hat{\mathcal{I}}_{[1,d]} \rightarrow \hat{V} \).

Consider any two terminal embeddings \( \gamma_1, \gamma_2 : Q \rightarrow D \) that satisfy \( \gamma_1(\hat{\mathcal{I}}_{[1,d]}) = \gamma_2(\hat{\mathcal{I}}_{[1,d]}) \). Consider in turn each element \( V_i\Box \) of tuple \( \hat{V} \). If \( V_i \) is a variable then \( \gamma_1(V_i\Box) = \gamma_2(V_i\Box) \) follows trivially from \( V_i\Box \in \hat{\mathcal{I}}_{[1,d]} \). Otherwise, \( V_i \) must be a constant. If \( v \) is the root vertex of \( \text{body}_Q \), it follows from the definition of partial application that \( \gamma_1(V_i\Box) = \gamma_2(V_i\Box) = V_i \). Otherwise \( v \) is a non-root vertex of \( \text{body}_Q \), and so \( \hat{\mathcal{I}}_{[1,d]} \) must contain some index \( X\Box \). It again follows from the definition of partial application that if \( \gamma(X\Box) \in \text{dom} \) then \( \gamma_1(V_i\Box) = \gamma_2(V_i\Box) = V_i \), whereas if \( \gamma(X\Box) = \mathbb{O} \) then \( \gamma_1(V_i\Box) = \gamma_2(V_i\Box) = \mathbb{O} \).

**2.3.2 Reducing Equivalence to Encoding-Equivalence**

In this section we reduce the problem of deciding equivalence of satisfiable unparameterized \( \text{COCQL}^{f_h f_s f_k} \) queries to deciding “encoding-equivalence” of HCEQs. We will illustrate this process with a running example using the following queries and database instance.

**Example 21** Consider the two \( \text{COCQL}^{f_h f_s f_k} \) queries shown in Figure 2.15. Following our convention, uppercase letters denote variables while integers and lowercase letters denote atomic constants; for reasons that will become clear shortly, we have also numbered the query nodes in which an aggregation function is applied. Over any database,
Figure 2.15: **COCQL**\{f_b, f_s, f_n\} queries Q_{18} and Q_{19}
both $Q_{18}$ and $Q_{19}$ output objects conforming to the sort $\tau$ shown in Figure 2.16. We have labelled the atomic sorts in $\tau$ with the corresponding output variables/constants from $Q_{18}$ (above) and $Q_{19}$ (below). We have labelled the collection sorts in $\tau$ with the corresponding node numbers from $Q_{18}$ (left) and $Q_{19}$ (right). Now consider evaluating $Q_{18}$ and $Q_{19}$ over database $D_3$ containing the relations $R$ and $S$ below. To aid in visualizing the database instance, we also provide a graphical depiction of $D_3$.

Both $Q_{18}$ and $Q_{19}$ return the object $o_{12}$ depicted in Figure 2.17. The figure also shows the linearization of $o_{12}$ with respect to $\tau$; observe that it conforms to the chain sort $\text{Chain}(\tau) = (\text{snsb}, 5)$ shown in Figure 2.16.

We will now describe a function $\text{Encode}$ that maps satisfiable unparameterized $\text{COCQL}(f_b, f_s, f_n)$ queries to hierarchical conjunctive encoding queries. Given an arbitrary satisfiable $\text{COCQL}(f_b, f_s, f_n)$ query $Q$, the encoding query $\text{Encode}(Q)$ is constructed by the following steps. For now, assume that $Q$ does not contain the generalized projection operator (this assumption is not restrictive since generalized projection can always be rewritten using scalar aggregation as per Figure 2.5).

1. Number each of the nodes in $Q$ that applies an aggregation function (i.e., the root $\text{Eval}$ as well as each $\lambda$), as we have done to query $Q_{18}$ in Figure 2.15.

2. For each node numbered $\downarrow$, create an algebraic expression $E_i$ as follows:
Figure 2.17: Query result $o_{12} = (Q_{18})^{D_3} = (Q_{19})^{D_3}$ and its linearization w.r.t. $\tau$
(a) Initialize $E_i$ with the algebraic expression whose result is the input to the aggregation function at node $\bullet$ (i.e., the right input if node $\bullet$ is an \( \lambda \) operator).

(b) Traverse $E_i$ depth-first starting from its root.
   - Whenever a scalar subquery operator (\( \lambda \)) is encountered, delete that operator and its entire right input expression, and continue traversing the left input expression.
   - Whenever a duplicate-preserving projection operator (\( \Pi^{dup} \)) is encountered
     - delete from its projection list all attributes of non-atomic sort,
     - add to its projection list any constants or attributes of atomic sort that are currently projected away,
     - change it to duplicate-eliminating projection, and
     - continue traversing its input expression.

(c) $E_i$ now contains only base relations, selection, Cartesian product, join, and duplicate-eliminating projection. Convert $E_i$ to a canonical form having a single duplicate-eliminating projection operator above a conjunctive selection operator above the Cartesian product of all of the relations.

For example, query $Q_{18}$ yields the following expressions.

\[
\begin{align*}
E_1 &= \Pi_{a,A,B,C,D}(\sigma_{B=C}(R(A,B) \times R(C,D))) \\
E_2 &= \Pi_{B,F}(\sigma_{E=B}(R(E,F))) \\
E_3 &= \Pi_{L,M,N,O,P}(\sigma_{K=D,O=B,A=L=M}(S(K,L) \times S(M,N) \times R(O,P))) \\
E_4 &= \Pi_{c,G,H,I,J}(\sigma_{G=F,H=I}(S(G,H) \times S(I,J)))
\end{align*}
\]

3. To construct the body of $\text{ENCODE}(Q)$:
   (a) From each expression $E_i$ above, create a vertex labelled $\square$ containing a CQ* body constructed from the set of relations and predicates occurring in $E_i$.
   (b) For each pair of expressions $E_i$ and $E_j$, if the traversal of $E_i$ visited (and deleted) the \( \lambda \) node labelled $\square$, then draw an edge from vertex $\square$ to vertex $\square$.

4. To construct the head of $\text{ENCODE}(Q)$:
   (a) Let $\tau$ be the output sort of $Q$. Label the atomic sorts in $\tau$ with the corresponding output variables/constants and the collection types with the corresponding node numbers (see Example 21).
   (b) The depth $d$ of the encoding query head is the depth of the chain sort $\text{CHAIN}(\tau)$ (e.g., $d = 4$ for $Q_{18}$). We calculate the index variables at each level $i \in [1, d]$ as follows.
      i. Enumerate the collection types in $\text{CHAIN}(\text{sort})$ starting at the root.
      ii. If the $i^{th}$ collection type enumerated has label $\square$, form $\mathcal{I}_i$ from the output list of $E_j$ by labelling each constant/variable with $\square$. 
For example, the collection types in \( \text{CHAIN}(\tau) \) from query \( Q_{18} \) are enumerated in the order \( \mathbb{A}, \mathbb{B}, \mathbb{C}, \mathbb{D} \), yielding the following indexes.

\[
\begin{align*}
\bar{T}_1 &= \mathbb{A}, \mathbb{B}, \mathbb{C}, \mathbb{D} \\
\bar{T}_2 &= \mathbb{L}, \mathbb{M}, \mathbb{N}, \mathbb{O}, \mathbb{P} \\
\bar{T}_3 &= \mathbb{B}, \mathbb{F} \\
\bar{T}_4 &= \mathbb{C}, \mathbb{G}, \mathbb{H}, \mathbb{I}, \mathbb{J}
\end{align*}
\]

(c) The output list is constructed from \( \tau \) by a pre-order traversal that emits at each atomic domain the associated variable/constant, labelled with the label of the closest containing collection type. For example, for query \( Q_{18} \) we obtain \( \bar{\mathcal{V}} = \mathbb{A}, \mathbb{C}, \mathbb{L}, \mathbb{F}, \mathbb{C} \).

5. Finally, we merge variables to simplify the query representation and to conform to our syntactic convention for HCQs.

(a) For every vertex \( v \) in the HCQ body and every variable \( X_i \in \mathcal{L}_v \), if \( \text{pred}_v \neq X_i = Y \) then replace all occurrences of \( X_i \) in \( Q \) with \( Y \).

This causes the following substitutions for our running example: \( C/B, E/B, G/F, I/H, K/D, O/B, \) and \( M/L \).

(b) Delete any predicates that are trivial.

(c) For every component \( X_i \in \bar{\mathcal{T}}_{[1,d]} \), delete all but the left-most occurrence of \( X_i \) from the index sequence \( \bar{\mathcal{T}}_{[1,d]} \).

Figure 2.18 shows the encoding query \( Q_{20} = \text{ENCODE}(Q_{18}) \).
When numbering nodes that apply aggregation functions, generalized projection nodes also need to be numbered. For example, see query \( Q_{19} \) in Figure 2.15.

During the traversal of each expression \( E_i \), whenever a generalized projection operator is encountered, delete from the attribute list the aggregation expression \( X = f(A_{j1}, \ldots, A_{ji}) \) (effectively converting the operator into duplicate-eliminating projection) and continue traversing its input expression. For example, query \( Q_{19} \) yields the following expressions.

\[
\begin{align*}
E_5 &= \Pi_{a, C'}, D'(\sigma_{B' = C' \land A' = E'}(R(A', B') \times R(C', D') \times R(E', F'))) \\
E_6 &= \Pi_{U', b}(\sigma_{V' = c \land U' = D'}(R(U', V'))) \\
E_7 &= \Pi_{K', U', M', N'}(\sigma_{K' = D' \land U' = M'}(S(K', L') \times S(M', N'))) \\
E_8 &= \Pi_{A', B', C', D', E', F'}(\sigma_{B' = C' \land A' = E'}(R(A', B') \times R(C', D') \times R(E', F'))) \\
E_9 &= \Pi_{c, H', p}(\sigma_{C' = F' \land H' = p}(S(G', H') \times S(I', J'))) \\
\end{align*}
\]

When creating the body of \( \text{ENCODE}(Q) \), we ignore the expressions corresponding to generalized projection nodes (i.e., don’t create \( \text{CQ}^* \) bodies for them and don’t consider them when constructing edges); the rest of the algorithm remains unchanged. For example, the traversal yielding expression \( E_8 \) above was a sub-traversal of the one yielding expression \( E_5 \). Because of this, all of the base relations (with their local variables) and predicates occurring in \( E_8 \) already occur in \( E_5 \).

The head of \( \text{ENCODE}(Q) \) is calculated from \( \tau \) and each expression \( E_j \) almost as before, except that when constructing index tuple \( \overline{I}_i \) for collection type \( \mathbb{C} \), we label the components of \( \overline{I}_i \) with the label of the nearest enclosing collection type within \( \tau \) that is constructed by an \( \lambda \) operator. For example, the collection types in query \( Q_{19} \) are enumerated in the order \( \mathbb{B}, \mathbb{C}, \mathbb{D}, \mathbb{E} \), but type \( \mathbb{B} \) is constructed by a generalized projection operator whose nearest enclosing type within \( \tau \) that is constructed by an \( \lambda \) operator is type \( \mathbb{E} \).

\[
\begin{align*}
\overline{I}_1 &= a, C, D, \overline{I}_2 = K, L, M, N, \overline{I}_3 = A, B, C, D, E, F, \overline{I}_4 = c, H, J
\end{align*}
\]

Output attributes are labelled similarly, yielding \( \overline{V} = a, C, L, F, c \).

A final syntax simplification is performed as before. This causes the following variable substitutions for our running example: \( C'/B', E'/B', G'/F', I'/H', K'/D', M'/L', U'/D', \) and \( V'/c \).

Figure 2.19 shows the encoding query \( Q_{21} \) which is the final result of our running example \( \text{ENCODE}(Q_{19}) \). Although expression \( E_8 \) was not used when creating the body of \( Q_{21} \), it was needed to determine the index tuple \( \overline{I}_3 \) in the query head.
Given any satisfiable unparameterized \( \text{COCQL}\{I_1, J_2, J_3\} \) query \( Q \), the query \( \text{Encode}(Q) \) is a valid encoding query.

**Proof.** When \( \text{body}_{\text{Encode}}(Q) \) is constructed by the algorithm \( \text{Encode}(Q) \), the \( \lambda \) operators effectively partition the traversals of the original algebra tree (ignoring the traversals corresponding to generalized projection operators, since they don’t introduce new nodes into \( \text{body}_{\text{Encode}}(Q) \)). This means that each labelled node in \( \text{body}_Q \) is only ever visited once, and hence every non-root node of \( \text{body}_{\text{Encode}}(Q) \) has exactly one incoming edge. This partitioning also guarantees that the sets of local variables in each CQ body are pairwise disjoint. Syntactic validity of query \( Q \) requires that every attribute referenced in a selection predicate must either be visible in the schema of the current algebra expression or must be bound within the evaluation environment inherited by the nearest \( \lambda \) operator. The fact that \( Q \) is unparameterized then guarantees that any non-local variable used in a CQ body occurs as a local variable in one of its ancestor CQ bodies. Hence, \( \text{body}_{\text{Encode}}(Q) \) is a proper HCQ body (as per Definition 2.3.2).

When \( \text{head}_{\text{Encode}}(Q) \) is constructed by algorithm \( \text{Encode}(Q) \), the variables in the head originate from attribute references in \( Q \). Because \( Q \) is unparameterized, every variable in \( \text{head}_{\text{Encode}}(Q) \) will be local to \( \text{body}_{\text{Encode}}(Q) \). In order for \( \text{head}_{\text{Encode}}(Q) \) to contain some variable \( W_i \), it follows from the algorithm that expression \( E_v \) must contain some variable \( W_j \) such that variables \( W_i \) and \( W_j \) were merged during the final step. Syntactic validity of query \( Q \) requires that either \( W_j \) is local to expression \( E_v \), or \( W_j \) was inherited from the parent evaluation environment in \( \text{body}_Q \). The construction of \( \text{body}_{\text{Encode}}(Q) \) entails that either \( W_i \in \mathcal{L}_v \) or \( W_i \in \mathcal{L}_u \) for some vertex \( u \) that is an ancestor of \( v \) in \( \text{body}_{\text{Encode}}(Q) \). Hence, \( \text{head}_{\text{Encode}}(Q) \) satisfies Definitions 2.3.2 and 2.3.3 and so \( \text{Encode}(Q) \) is a proper HCQ.

Finally, we need to establish that \( \text{Encode}(Q) \) is valid in that it satisfies Definition 2.3.4. Algorithm \( \text{Encode}(Q) \) causes each component of the output tuple to also appear in one of the index tuples (because it necessarily occurs within the projection list of one of the expressions \( E_i \)). Therefore, by Lemma 2.3.5, \( \text{Encode}(Q) \) is valid. \( \square \)

---

**Figure 2.19:** Encoding query \( Q_{21} = \text{Encode}(Q_{19}) \)
Theorem 2.3.7  Given any satisfiable unparameterized COCQL\(\{f_b,f_s,f_n\}\) query \(Q\), the query \(\text{ENCODE}(Q)\) is a valid conjunctive encoding query.

Proof. Similar to the proof of Theorem 2.3.6, but relies on the fact that the extended definition of \(\text{ENCODE}(Q)\) avoids introducing hierarchical edges into the query body for generalized projection operators.

Example 22 Consider the encoding queries \(Q_{20} = \text{ENCODE}(Q_{18})\) from Figure 2.18 and \(Q_{21} = \text{ENCODE}(Q_{19})\) from Figure 2.19. Figures 2.20 and 2.21 shows the encoding relations \(R_8\) and \(R_9\) that result from evaluating \(Q_{20}\) and \(Q_{21}\), respectively, over database \(D_3\) from Example 21. The reader can verify that decoding either relation with the signature \(\text{snsb}\) (cf. Algorithm 3) yields the object \(o_{13}\) shown in Figure 2.22.

Observe that object \(o_{13}\) in Figure 2.22 differs from object \(\text{LINEARIZE}_\tau(o_{12})\) in Figure 2.17 only by the additional shading of one of the collection nodes. This “spurious shading” is a side-effect of the decoding process, but it is entirely predictable. The spuriously shaded set within chain object \(o_{13}\) corresponds to an (unshaded) set within object \(o_{12}\) that was linearized underneath an instance of an empty normalized bag. By adding additional bookkeeping to Algorithm 1, it is possible to create a modified function \(\text{CHAIN}\) that adds spurious shadings whenever a collection object is linearized beneath an instance of an empty collection. Analogous to Definition 2.1.8, we can then define a modified linearization function \(\text{LINEARIZE}_\tau\) as follows

\[
\text{LINEARIZE}_\tau(o) := \text{CHAIN} \circ \text{COMPLETE}_\tau(o)
\]

which then satisfies \(o_{13} = \text{LINEARIZE}_\tau(o_{12})\). It follows from Observation 2.1.6 that for each tuple sub-object within \(\text{COMPLETE}_\tau(o)\), the roots of the tuple components are either all shaded or all unshaded. Given sort \(\tau\) and object \(\text{LINEARIZE}_\tau(o)\), we can therefore identify precisely the shadings that are spurious. The following theorem and corollary are suitably modified versions of Theorem 2.1.9 and Corollary 2.1.10.

Theorem 2.3.8  Given sort \(\tau\) not containing any zero-ary tuple sorts, \(\text{LINEARIZE}_\tau\) has a retraction.

Corollary 2.3.9  Given sort \(\tau\) not containing any zero-ary tuple sorts, any two objects \(o, o' \in [\tau]\) satisfy the following.

\[
o = o' \iff \text{LINEARIZE}_\tau(o) = \text{LINEARIZE}_\tau(o')
\]

Theorem 2.3.10  Given any finite database \(D\) and any COCQL\(\{f_b,f_s,f_n\}\) query \(Q\) with output sort \(\tau\) not containing any zero-ary tuples, let \(\bar{\tau}\) be the signature corresponding to the linearization of \(\tau\). Then, the complex object \((Q)^D\) and the encoding relation \((\text{ENCODE}(Q))^D\) are related as follows.

\[
\text{DECODE}((\text{ENCODE}(Q))^D,\bar{\tau}) = \text{LINEARIZE}_\tau((Q)^D)
\]

(2.26)
| $R_8$ | $a_1$ | $A_1$ | $B_1$ | $D_1$ | $F_2$ | $N_3$ | $B_3$ | $P_3$ | $B_2$ | $F_2$ | $d_1$ | $F_1$ | $H_3$ | $J_1$ | $a_1$ | $B_1$ | $I_3$ | $F_2$ | $d_1$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| a     | a     | 2     | 3     | 4     | 5     | 2     | 3     | 2     | 3     | c     | 3     | 4     | 5     | a     | 2     | 4     | 3     | c     |
| a     | a     | 2     | 3     | 4     | 5     | 2     | 3     | 2     | 3     | c     | 3     | 4     | 6     | a     | 2     | 4     | 3     | c     |
| a     | a     | 2     | 3     | 4     | 5     | 2     | 3     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 5     | 2     | 7     | 2     | 3     | c     | 3     | 4     | 5     | a     | 2     | 4     | 3     | c     |
| a     | a     | 2     | 3     | 4     | 5     | 2     | 7     | 2     | 3     | c     | 3     | 4     | 6     | a     | 2     | 4     | 3     | c     |
| a     | a     | 2     | 3     | 4     | 5     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 3     | 2     | 3     | c     | 3     | 4     | 5     | a     | 2     | 4     | 3     | c     |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 3     | 2     | 3     | c     | 3     | 4     | 6     | a     | 2     | 4     | 3     | c     |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |
| a     | a     | 2     | 3     | 4     | 6     | 2     | 7     | 2     | 7     |       |       |       |       | a     | 2     | 4     | 7     |       |       |

Figure 2.20: Encoding query result $R_8 = (Q_{20})^{D_3}$ satisfying $\text{DECODE}(R_8, \text{snsb}) = o_{13}$
Figure 2.21: Decoding query result $D^6_9(s) = 0^1_9$.

<table>
<thead>
<tr>
<th></th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
<th>1 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
<tr>
<td>a 2 3</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
<td>6 4</td>
</tr>
</tbody>
</table>
Figure 2.22: Query result $o_{13} = \text{DECODE}((Q_{20})^D, \text{snsb}) = \text{DECODE}((Q_{21})^D, \text{snsb})$

**Proof (sketch).** For conciseness of presentation, let $o$ denote the object $(Q)^D$, $Q'$ denote the encoding query $\text{ENCODE}(Q)$, and $R$ denote the encoding relation $(\text{ENCODE}(Q))^D$.

If $o$ is a trivial object, then $\text{LINEARIZE}_r(o)$ is one of $\{\} \cup \{\}$, or $\{\} \cup \{\}$ (depending upon the first character of $\overline{\xi}$). In this case it is easy to show that $R$ will be an empty encoding relation, and so the result follows from the definition of $\text{DECODE}$ (Algorithm 3). Therefore, assume that $o$ is not trivial.

Consider the simple case where $o$ is a chain object that does not contain empty collections, and so object linearization (cf. Section 2.1.2) is idempotent.

$$\text{LINEARIZE}_r(o) = \text{CHAIN} \circ \text{COMPLETE}_r(o) = o \quad (2.27)$$

The procedure $\text{ENCODE}$ essentially partitions the algebraic tree of $Q$ based upon the $\lambda$ operators andreshapes the hierarchy of these partitions into a hierarchy of CQ bodies. It is straightforward to verify that because $o$ does not contain any empty collections, $R$ does not contain the $\Xi$ symbol. In this simple case, procedure $\text{DECODE}$ (Algorithm 3) is equivalent to procedure $\text{DECODESIMPLE}$ (Algorithm 2). Because $\text{DECODESIMPLE}$ essentially performs the same collection-construction operations as would be required to evaluate $Q$ over $D$, it is tedious but straightforward to verify the following.

$$\text{DECODE}(R, \overline{\xi}) = \text{DECODESIMPLE}(R, \overline{\xi}) = o \quad (2.28)$$

The result then follows from equations 2.26, 2.27, and 2.28.

Now suppose that $o$ is a chain object but contains an empty collection generated by some $\lambda$ operator.

$$\text{LINEARIZE}_r(o) = \text{CHAIN} \circ \text{COMPLETE}_r(o) = \text{COMPLETE}_r(o) \quad (2.29)$$
In this case, function $\text{COMPLETE}_\tau$ introduces shaded nodes to complete the object. Appealing to the semantics of HCQs, we can show that there exist a partial application of some terminal embedding of $Q'$ into $D$ that assigns the value $\mathbf{x}$ to all of the variables in the index level corresponding to the shaded subobject. These embeddings yield index tuples satisfying the predicate $\overline{a} = \overline{a}^{|l|}$ in Algorithm 3, causing the constructed nodes to be shaded. The result then follows from equations 2.26 and 2.29.

Finally, suppose that $o$ is not a chain object;

$$\text{LINEARIZE}_\tau(o) = \text{CHAIN} \circ \text{COMPLETE}_\tau(o) \quad (2.30)$$

in this case, the effect of the $\text{COMPLETE}_\tau$ function is mimicked by $\mathbf{x}$ values in $R$ and in the decoding process as described above, but the effect of the $\text{CHAIN}$ function is non-trivial. The net effect of the $\text{CHAIN}$ function is three-fold: first, to linearize sibling collection types consistent with a pre-order traversal of $\tau$ (see Example 7); second, to push atomic values down by distributing them across all of the leaves; and third, to add spurious shadings to certain collection nodes. The $\text{ENCODE}$ and $\text{DECODE}$ functions together mirror all three of these behaviours.

The type linearization occurs both when the indexes within encoding head of $Q'$ are chosen based upon a pre-order traversal of $\tau$ and when the signature $\overline{S}$ used in the decoding process is calculated via a pre-order traversal of $\tau$. The push-down of atomic values is implicit in the semantics of HCQs—specifically, that the value for a particular attribute originating from a parent block is repeated within all of the tuples originating from different terminal embeddings that agree on the variable bindings of the parent block but differ on the bindings of the child block.

Lastly, consider any two sibling collection types within sort $\tau$ that correspond to collection types in $\text{CHAIN}(\tau)$ at depths $i, j$ with $i < j$. Given any tuple that satisfies $\overline{I}_{i-1} = \overline{I}_{i-1}^{|l|}$ but $\overline{I}_j \neq \overline{I}_j^{|l|}$, the predicate test $\overline{I}_{i-1} = \overline{I}_{i-1}^{|l|}$ in Algorithm 3 will cause a spurious shading of the child collection object constructed at depth $i$.

**Definition 2.3.11 (Encoding-Equivalence)** Given two encoding queries $Q$, $Q'$ over database schema $\mathcal{S}(D)$ and a signature $\overline{S}$, if for each relational database $D$ with schema $\mathcal{S}(D)$ and finite instance $T(D)$ the encoding-equality $(Q)^D \equiv_{\overline{S}} (Q')^D$ holds, then we say that $Q$ and $Q'$ are encoding-equivalent w.r.t. signature $\overline{S}$ (abbreviated $\overline{S}$-equivalent, and denoted $Q \equiv_{\overline{S}} Q'$).

**Theorem 2.3.12** Given two satisfiable, unparameterized COCQL($f_b, f_s, f_n$) queries $Q$, $Q'$ with the same output sort $\tau$ not containing zero-ary tuples,

$$Q \equiv Q' \iff \text{ENCODE}(Q) \equiv_{\overline{S}} \text{ENCODE}(Q')$$

where $\overline{S}$ is the signature corresponding to the linearization of $\tau$.

**PROOF.** Follows from the relationship between the output of queries $Q$ and $\text{ENCODE}(Q)$ (Theorem 2.3.10) and the existence of a retraction for transformation $\text{LINEARIZE}_\tau$ (Theorem 2.3.8 and Corollary 2.3.9).
Note that the ban on zero-ary tuples in Theorem 2.3.12 is easily overcome. If the output sort $\tau$ of $Q$ and $Q'$ contains a collection of zero-ary tuples, we can convert this to a collection of unary tuples by modifying both $Q$ and $Q'$ so that the corresponding query operators project a constant into the collection constructor. This modification does not change the equivalence relation between $Q$ and $Q'$.

**Corollary 2.3.13** Deciding equivalence of satisfiable $\text{CSQL}^{(f_b,f_s,f_n)}$ queries is polynomial-time reducible to $\approx_{\text{S}}$-equivalence of hierarchical conjunctive encoding queries.

**Proof.** Follows from Proposition 2.1.22.

**Corollary 2.3.14** Deciding equivalence of satisfiable $\text{CSQL}^{(f_b,f_s,f_n)}$ queries is polynomial-time reducible to $\approx_{\text{S}}$-equivalence of conjunctive encoding queries.

**Proof.** Follows from Proposition 2.1.23 and Theorem 2.3.7.

**Example 23** The equivalence of $\text{COCQL}^{(f_b,f_s,f_n)}$ queries $Q_{18}$ and $Q_{19}$ can be decided by testing the condition $Q_{20} \approx_{\text{S}} Q_{21}$ (which we show to be true in Example 59, Chapter 5).

Now suppose that $Q_{18}$ and $Q_{19}$ were each modified by replacing the aggregate function $f_b$ with $f_n$. The modified queries would not be equivalent, because $Q_{20}$ and $Q_{21}$ are not $\approx_{\text{S}}$-equivalent. (The reader can easily verify from Figures 2.20 and 2.21 that $R_8 \neq_{\text{S}} R_9$ because object $\text{Decode}(R_8)$ contains a sub-bag with cardinality four, while object $\text{Decode}(R_9)$ does not.)

### 2.4 Relevant Literature

Here we briefly survey literature relevant to deciding equivalence between queries that output complex objects.

#### 2.4.1 Query Languages for Complex Objects

Much of the previous work on complex objects has been performed within the context of nested relations. Because a relation is by definition a set of tuples, a nested relation corresponds to a complex object whose sort contains only set collections. The literature surrounding nested relations and (set-based) complex objects is vast; Abiteboul, Hull, and Vianu provide a good summary in their textbook [6, Ch. 20].

Some of the earliest work on nested structures includes the quotient relations of Furtado and Kerschberg, in which tuples are organized into blocks via the partition and departition operators [35]. The partitioning of quotient relations is not recursive, however, and so the nested relational model along with the restructuring operators nest and unnest of Jaeschke and Schek [65] can be viewed as a recursive generalization of quotient relations. Jaeschke and Schek’s nest/unnest operators
are limited to nesting/unnesting of a single attribute, however; the generalization of nest/unnest to handle nested relations of arbitrary arity is due to subsequent work by Thomas and Fischer [103].

Gyssens, Paradaens, and Van Gucht analyze the space of possible nested relations with the goal of identifying useful subclasses, since they contend that not all nested relations correspond to real world situations [55]. They organize subclasses of nested relations (NR) into the following hierarchy.

\[
\text{HNR} < \text{NFR} < \text{NL} < \text{NR}
\]

The class of normalization-lossless relations (NL) consists of all relations obtainable from flat relations using arbitrary sequences of nest and unnest operators. Note that relations containing empty nested sub-relations are members of NR but not of NL. The class of nested flat relations (NFR) consists of relations obtainable from flat relations using only consecutive nest operations. Finally, the class of hierarchical nested relations (HNR) is the subset of NFR in which at each level of nesting the atomic attributes form a superkey. Relations within HNR are also known as Verso-relations [5, 13], and have the useful property that the size of an instance is polynomially-bounded by the number of atomic elements in its active domain [6, Ch. 20].

Whereas Gyssens et al.’s taxonomy only considers restructuring performed by sequences of nest and unnest operators, the query language usually called the nested relational algebra (NRA) is formed by adding Thomas and Fischer’s nest and unnest operators to the relational algebra. The expressive power of NRA is very limited compared to many complex object query languages; not only is it unable to create empty sub-relations, Paredaens and Van Gucht showed that its expressive power does not exceed the standard relational algebra for queries that both input and output flat relations [89]. A flurry of research in late 1980s and early 1990s yielded a variety of query languages for complex objects with expressive power exceeding NRA (in some cases, far exceeding NRA by including various combinations of operators such as empty collection constructors, transitive closure, fixed points, powersets, etc.). For example, pairs of algebraic and calculus-based languages for complex objects along with a proof of equivalent expressive power are proposed by Ozsoyoglu et al. [88], Buneman et al. [14], and Abiteboul and Beeri [4]. Grumbach considers a calculus for bags [49], while Libkin and Wong consider a variety of calculus-based languages that include nested sets, bags, and various aggregation functions [78, 79, 112]. A common theme throughout this body of literature is the focus on the expressive power of the query languages. Since all of these languages meet or exceed the expressive power of the traditional relational algebra/calculus, the query equivalence problem is undecidable for any of these languages.

To situate our work within the context of this body of literature, our query language $\text{COCQL}_{\{f_b, f_s, f_n\}}$ can be described informally as follows. The restricted subclass $\text{COCQL}_{\{f_b, f_s, f_n\}}$ that we study in Chapter 3—queries that do not contain scalar sub-queries but may contain generalized projection—is essentially an extension of the
bag semantic conjunctive relational algebra with three variants of the nest operator (for constructing different collection types), but with no unnest operator and with certain restrictions on the operations permitted on non-atomic attributes. If the three variants of nest are replace with analogous variants of the outer-nest operator described by Levy and Suciu [77], then the full language COCQL\(^{f_b,f_s,f_n}\) is obtained.

Even though language COCQL\(^{f}\) does not contain an unnest operator, it is straightforward to show that any object in Gyssens et al.’s class NR can be constructed by language COCQL\(^{f_b}\) out of some database of flat relations. Similarly, language COCQL\(^{f_s}\) is capable of constructing any object in class NL. This apparent discrepancy is due to the simple fact that COCQL\(^{f_s}\) allows duplicate-eliminating projection to appear above generalized projection, whereas Gyssens et al. restrict their attention to operator sequences containing only nest/unnest. While on one hand this seems to call into question the usefulness of Gyssens et al.’s taxonomy when considering more powerful query languages, there is an interesting connection between their taxonomy and the equivalence problem that we study in this thesis. Because encoding relations rely on atomic-valued index attributes to distinguish the encoded objects, encoding relations resemble “flattened” Verso-relations. Our main result in Chapter 3 can then be construed as saying that equivalence of COCQL\(^{f_b,f_s,f_n}\) expressions can be reduced to equivalence of queries that return objects in the simplest class HNR. This does not, however, prove the decidability of the open problem of equivalence between two nest/unnest sequences, because such a sequence may not be equivalent to any COCQL\(^{f_b,f_s,f_n}\) query.

2.4.2 Relational Encodings of Complex Objects

Our approach of encoding arbitrary complex objects into flat relations and subsequently translating queries over objects into queries over flat relations is not novel. Suciu proposes a query language that extends NRA with a bounded fixed point operator, and his proof that this language is a conservative extension of non-nested relational algebra plus bounded fixed points—that is, that the languages have the same expressive power over flat inputs and outputs—uses a similar technique [102]. Suciu’s encoding method uses a separate relation for each collection type in the object’s sort (he only considers nested sets), and uses index values to distinguish the elements of each collection. Suciu’s encoding handles arbitrary typed objects, whereas our encoding method only handles chain objects, and thus necessitates prior application of the linearization transformation described in Section 2.1.2. The two encoding methods are fundamentally related: our relational representation obtained by object linearization followed by subsequent encoding of the chain object is equivalent to the left outer join of the various relations within Suciu’s encoding. Our encoding requires a special symbol because it corresponds to the NULL values introduced by an outer join operator.
2.4.3 Algebraic Transformations

Algebraic query optimization is a well-studied problem. The literature pertaining to algebraic optimization of SQL queries can be divided into two different veins.

One line of research focuses on algebraic transformations that change the “shape” of the query, where a query’s “shape” is loosely defined as the nesting structure of the SELECT–FROM–WHERE–GROUPBY blocks (which corresponds visually to the shape of the associated Query Graph Model representation [56]). Early work in this vein includes work by Kim on flattening of nested query blocks [69], and subsequent work by Dayal on both merging and decorrelating nested blocks [28]. The advent of rule-based optimization paradigms pioneered by systems like STARBURST [56] and VOLCANO [47] caused a focus on local manipulations of algebra trees. Various algebraic transformation rules for commuting aggregation blocks, partitioning or merging aggregation operations, avoiding duplicate elimination, or commuting aggregation with join were proposed by Yan and Larson [113, 114], Paulley and Larson [91], Gupta, Harinarayan, and Quass [54], and Khizder and Weddell [68].

The second vein of research generalizes the well-known heuristic of pushing down predicates. Starting with the idea of “magic sets”—an optimization technique proposed for bottom-up evaluation of recursive Datalog programs [9]—Mumick and various co-authors apply the idea to nested SQL queries. As a result, they derive conditions for introducing or eliminating redundant (semi)joins into existing query blocks, or moving selection or join predicates around the algebra tree [76, 85, 86, 87].

Algebraic transformation rules have also been proposed for nested relational algebra queries. Scholl proposes a set of algebraic equivalences that hold over NRA expressions, with the stated goal of eliminating redundant join operations [100]. More recently, Liu and Yu propose an optimization algorithm for NRA expressions based upon heuristic application of algebraic transformations that also attempt to reduce the execution cost of certain types of nested join operations [80].

While algebraic transformations have proved to be a powerful and flexible optimization technique, many of the transformation rules appearing in the literature are given with only sound conditions characterizing their applicability. Complete characterizations for a given transformation rule are difficult to prove because applicability could easily depend upon global features of the query plan, whereas algebraic transformations by their nature perform very localized query rewriting. The query transformation literature as a whole fails to provide a systematic understanding of underlying principles governing global interactions between different nested query blocks. As such, this body of work does not furnish a coherent characterization of query equivalence that would enable the development of a provably complete decision procedure.
2.4.4 Query Equivalence

The equivalence of conjunctive queries that input and output flat relational structures is a well-studied problem which we will survey in Chapter 3 when we consider encoding-equivalence of CEQs. We restrict our attention here to the small amount of literature that considers the query equivalence problem for queries that output nested structures.

When a query language allows the ability to both create and explicitly test for empty collections, it can simulate the full relational algebra and query equivalence becomes undecidable. Van den Bussche et al. recently proved a more interesting undecidability result. They show that the query equivalence problem is undecidable for the Positive-Existential fragment of the Nested Relational Calculus [107]. NRC is a conservative extension of relational algebra, and it includes the ability to both create and test for empty sets. PENRC is the fragment of NRC formed by removing relational difference and the emptiness test. Although PENRC lacks the ability to explicitly test set-emptiness, Van den Bussche et al.’s proof of undecidability relies on the ability to both construct objects containing empty subsets and to union sets, but does not appear to require the ability to unnest sets. As such, their undecidability result would likely transfer to any extension of our language COCQL that includes the ability to union collections, but not necessarily to a similar extension to COCQL since that language does not have the ability to create empty sets.

Containment and equivalence of queries returning set-based complex objects is studied by Levy and Suciu [77], who consider “conjunctive OQL” (COQL) queries (a language with expressive power equivalent to extending our language COCQL with an unnest operator). Whereas containment of flat relations indisputably corresponds to set inclusion, there is no single definition for containment of nested sets. Levy and Suciu use an inductive definition of containment previously proposed for Verso relations [13], and they reduce containment (under this definition) of COQL queries constructing objects with nesting depth \( d \) to testing a relationship between CQs that they call “simulation to depth \( d \)” defined as follows. Let \( Q(\overline{I_1}; \ldots; \overline{I_d}; \overline{V}) \) be a CQ whose head has been annotated to distinguish \( d \) sets of index variables, and define \( \overline{I} := (\overline{I_1}; \ldots; \overline{I_d}) \). Given two such queries, \( Q \) simulates \( Q' \) to depth \( d \)—denoted \( Q \preceq_d Q' \)—iff over every database instance the following equation holds:

\[
\forall \overline{I_1}. \exists \overline{I_1} \ldots \forall \overline{I_d}. \exists \overline{I_d}. \forall \overline{V} \left[ Q(\overline{I}; \overline{V}) \implies Q'(\overline{I'}; \overline{V}) \right]
\]

a condition characterized by the existence of a simulation mapping, and hence NP-complete to decide [77]. For COQL queries that cannot construct empty sets (i.e., our language COCQL plus unnest), containment reduces to a single simulation test. Levy and Suciu claim that arbitrary COQL containment reduces to testing an exponential number of simulation conditions; however, Dong et al. point out that this is insufficient for implying containment [32]. Dong et al. consider containment of a restricted class of COQL queries (corresponding to XQuery), showing it to be
Figures 2.23: COQL queries $Q_{22}$, $Q_{23}$, and $Q_{24}$

in co-NEXPTIME, but NP-complete or co-NP-complete for a variety of further restrictions. To the best of our knowledge, the complexity of the general COQL containment problem remains open.

The containment relationship used by Levy and Suciu is not antisymmetric (mutual containment does not imply equivalence) and so they define a separate “strong simulation” relationship between CQs for testing COQL equivalence. Query $Q$ strongly simulates $Q'$ to depth $d$—denoted $Q \preceq_d Q'$—iff:

$$\forall \mathcal{I}_1, \exists \mathcal{I}_1', \ldots, \forall \mathcal{I}_d, \exists \mathcal{I}_d', \forall \mathcal{V} \left[ Q(\mathcal{I}; \mathcal{V}) \iff Q'(\mathcal{I}_d'; \mathcal{V}) \right]$$

(2.32)

a condition which they claim is characterized by the existence of a strong simulation mapping [77], and hence still NP-complete to decide (although they define this mapping only for $d \leq 1$). While equivalence of general COQL queries is left open, they claim that equivalence for COQL queries that cannot construct empty sets (i.e., our COCQL[$\text{fs}$]—plus unnest) reduces to testing a single strong simulation condition in each direction (Proposition 6.3 [77]). We end this section with an example that demonstrates that this reduction of query equivalence to strong simulation is incorrect.

**Example 24** Consider a database containing a relation $E(P,C)$ that denotes parent-child relationships, along with the three queries shown in Figure 2.23 (written in an SQL-like syntax that corresponds to empty-set-free COQL). Query $Q_{22}$ returns sets of related grandchildren, grouped first into sets with a common parent, and then into sets with a common grandparent. Query $Q_{23}$ is similar to $Q_{22}$, but the outer aggregation groups by pairs of grandparents. Query $Q_{24}$ is also similar to $Q_{22}$, but the inner aggregation groups by both parent and grandparent. Levy and Suciu’s technique associates $Q_{22}$,
Consider the database $D_4$ in Figure 2.24 and the corresponding query results in Figure 2.25 (index groups have been visually separated for clarity). The reader can verify that over database $D_4$ all six strong simulation conditions $Q'_22 \preccurlyeq_2 Q'_23$, $Q'_23 \preccurlyeq_2 Q'_22$, $Q'_22 \preccurlyeq_2 Q'_24$, $Q'_24 \preccurlyeq_2 Q'_22$, $Q'_23 \preccurlyeq_2 Q'_24$, and $Q'_24 \preccurlyeq_2 Q'_23$ are satisfied (cf. equation 2.32); in fact, we can show that they are all satisfied over any database. However, the queries are not all equivalent since over $D_4$ queries $Q'_22$ and $Q'_24$ output the object $\{\{c_1, c_2\}, \{c_3\}\}$ while $Q'_23$ outputs $\{\{c_1, c_2\}, \{c_3\}, \{c_3\}\}$. Using the results from the next chapter, we can show that queries $Q'_22$ and $Q'_24$ are equivalent.
Chapter 3

Encoding-Equivalence of Conjunctive Encoding Queries

The purpose of this chapter is to present a necessary and sufficient condition for encoding-equivalence in the case where both queries are conjunctive encoding queries (see Section 2.3). Combined with Theorems 2.3.12 and 2.3.7 from Chapter 2, the results in this chapter yield a procedure for deciding query equivalence for the class of COCQL expressions. This subclass is of practical interest because COCQL expressions include queries that can be created by “stacking” views defined by SPJG algebra expressions [29] (see Proposition 2.1.20).

In Theorem 2.3.7 we showed that when the ENCODE algorithm is applied to a COCQL query, it yields a valid CEQ (cf. Definition 2.3.4). In this chapter, however, we consider all valid and satisfiable CEQs, independent of their origin. As per Definition 2.3.3, the body of any CEQ is a single CQ body; therefore, without loss of generality we dispense with the vertex labels in the query head, since all of the components of the head must have the same label. We restrict our attention to satisfiable CEQs, and so—by our syntactic convention for CQs—we can assume without loss of generality that the query bodies do not contain explicit equality predicates. The following lemma then characterizes CEQ validity.

Lemma 3.0.1 Assuming that the set of possible databases is not restricted by schema constraints, conjunctive encoding query $Q$ is valid if and only if every output variable also occurs as an index—that is, $(\mathcal{V} \cap \mathcal{B}) \subseteq \mathcal{I}_{[1, d]}$.

Proof. Sufficiency holds by Lemma 2.3.5. Conversely, if $(\mathcal{V} \cap \mathcal{B}) \not\subseteq \mathcal{I}_{[1, d]}$ then construct database $D$ by unioning two copies of $\text{body}_Q$ in which the variables in $(\mathcal{V} \cap \mathcal{B}) \setminus \mathcal{I}_{[1, d]}$ have been isomorphically renamed within one of the copies. (Because we have assumed that $Q$ does not contain explicit equality predicates, $\text{body}_Q$ is simply a set of atoms and can therefore be treated as a set of canonical tuples.) The query result $(Q)^D$ then violates the functional dependency $\overline{\mathcal{I}}_{[1, d]} \rightarrow \overline{\mathcal{V}}$, and so is not a valid encoding relation. \hfill \Box
The remainder of this chapter will proceed as follows. We propose a normal form for CEQs in Section 3.1 and prove that the transformation to the normal form preserves encoding-equivalence. In Section 3.2 we characterize equivalence between CEQs in normal form in terms of the existence of a special type of homomorphism. We survey related literature in Section 3.3.

### 3.1 A Normal Form for CEQs

The main contribution of this section is a normal form for conjunctive encoding queries that preserves encoding-equivalence relative to a given signature $\mathcal{S}$. Because calculating the normal form requires reasoning about dependencies—specifically, multivalued dependencies—that hold over the result of a conjunctive query, we start by discussing how to reason about multivalued dependencies that are implied by a query definition.

#### 3.1.1 Dependencies Over Query Results

We say that a query definition implies a particular dependency if that dependency is guaranteed to hold over the result of that query when evaluated over any possible database instance. Before we consider query-implied dependencies, we quickly review the concept of a query homomorphism. The reduction of the query equivalence problem to the existence of query homomorphisms is a fundamental result within database theory, and standard textbook material [6].

**Definition 3.1.1 (CQ Homomorphism)** Given two conjunctive queries $Q(\overline{W})$ and $Q'(\overline{W}')$, a homomorphism $h$ from $Q'$ to $Q$ is a mapping $h : \mathcal{B}' \to \mathcal{B} \cup \mathcal{C}$ (suitably extended to tuples, sets, and atoms, and with identity on constants) satisfying

1. $h(\text{body}_{Q'}) \subseteq \text{body}_Q$; and
2. $h(\overline{W}') = \overline{W}$.

We write $h : Q' \to Q$ to denote that $h$ is a homomorphism from $Q'$ to $Q$. A satisfiable CQ $Q(\overline{W})$ is contained under CQ $Q'(\overline{W}')$ if and only if there exists a homomorphism $h : Q' \to Q$. Two CQs are equivalent if and only if they are mutually contained. A conjunctive query $Q$ is minimal if there is no equivalent query $Q'$ such that $|\text{body}_{Q'}| < |\text{body}_Q|$. Each query $Q$ has a unique minimal form (up to isomorphic renaming of variables), but computing it is NP-complete [6, Sect. 6.2].\(^1\) The following theorem proves a property of minimal CQs that will be useful for later proofs.

\(^1\)Technically, the NP-complete decision problem is testing query equivalence/containment, while minimizing a query requires a linear number of such tests.
Theorem 3.1.2 Given any minimal conjunctive query $Q$, any mapping $h : B \rightarrow B \cup C$ (extended with identity on constants) that satisfies $h(W) = W$ and $h(body_Q) \subseteq body_Q$ forms a bijection between $B$ and itself.

Proof. Assume w.l.o.g. that $W$ does not contain repeated variables or constants. Suppose that mapping $h : B \rightarrow B \cup C$ satisfies $h(W) = W$ and $h(body_Q) \subseteq body_Q$ but does not form a bijection between $B$ and itself. Then, $h$ must satisfy $h(body_Q) \neq |body_Q|$. Furthermore, $h(W)$ must be a permutation of $W$. Because every permutation has a finite period, there exists some positive integer $j$ such that the composite mapping $h^j := h \circ \ldots \circ h$ satisfies $h^j(W) = W$; however, then $h^j$ is a homomorphism from $Q$ to itself that contradicts the minimality of $Q$. \hfill \square

Implied Multivalued Dependencies

A multivalued dependency (MVD) over some attribute set $W$ is an expression of the form $X \rightarrow Y$, where $X, Y$ are disjoint subsets of $W$. Let $Z$ denote the attribute set $W \setminus (X \cup Y)$. Then, a relation $R$ over $W$ satisfies $X \rightarrow Y$ if the following equation holds [6, Sect. 8.3].

$$R = \Pi_{XY}(R) \bowtie_X \Pi_{XZ}(R)$$

Given an CQ $Q$ that yields a relation over $W$, we say that $Q$ implies $X \rightarrow Y$—denoted $Q(W) \models X \rightarrow Y$—if for every database $D$ the relation $(Q)^D$ satisfies $X \rightarrow Y$.

$$(Q)^D = \Pi_{XY}((Q)^D) \bowtie_X \Pi_{XZ}((Q)^D)$$

Projection and natural join are conjunctive operations, and so the entire right-hand side of equation 3.2 can be perceived as a single conjunctive query evaluated over $D$. The following theorem formalizes this.

Theorem 3.1.3 $Q(W) \models X \rightarrow Y \iff Q \equiv \Pi_{XY}(Q) \bowtie_X \Pi_{XZ}(Q)$

Corollary 3.1.4 $Q(W) \models X \rightarrow Y$ iff there exists a homomorphism $h : Q \rightarrow Q$ (thereby satisfying $h(W) = W$) such that $h(atoms_Q)$ can be partitioned into two sets $S_Y$ and $S_Z$ satisfying

1. variables $Y$ only occur in $S_Y$,
2. variables $Z$ only occur in $S_Z$, and
3. all variables in common between $S_Y$ and $S_Z$ occur in $X$.

Corollary 3.1.5 If (satisfiable) conjunctive query $Q(W)$ is equivalent to a constant selection operation over conjunctive query $Q'(W)$

$$Q \equiv \sigma_{\mathbf{X}}(Q')$$

then $Q \models X \rightarrow Y$ iff $Q' \models X \cup A \rightarrow Y \setminus A$. 

75
Corollary 3.1.6 (Proposition 10.2.4 in [6]) If conjunctive query $Q(\overline{W})$ is equivalent to a projection operation over conjunctive query $Q'(\overline{W}')$

$$Q(\overline{W}) \equiv \Pi_{\overline{W}}(Q'(\overline{W}'))$$

then $Q \models X \rightarrow Y$ iff $Q' \models X \rightarrow Y \cup Y'$ for some $Y' \subseteq \mathcal{B} \setminus \mathcal{W}$, where $\mathcal{B}'$ is the set of variables in $\text{body}_{Q'}$.

Theorem 3.1.7 Deciding whether a given conjunctive query implies a given multivalued dependency is NP-complete.

Proof. Theorem 3.1.3 establishes that the problem is in NP. We establish NP-hardness using a reduction from testing containment of boolean CQs. Let $Q_a$ and $Q_b$ be two boolean CQs whose bodies contain the disjoint sets of variables $\mathcal{B}_a$ and $\mathcal{B}_b$, respectively. Let $A, Z$ be two fresh variables. Let $Q(\overline{W})$ be a new conjunctive query whose output tuple satisfies $\mathcal{W} = \mathcal{B}_a \cup \{A, Z\}$, and whose body is defined as follows.

$$\text{body}_Q = \text{body}_{Q_a} \cup \text{body}_{Q_b} \cup \bigcup_{x \in B_a \cup B_b} \{R(A, x), R(x, Z)\}$$

Then, $Q_a \subseteq Q_b$ iff there exists a homomorphism $h : B_b \rightarrow B_a$ such that $h(\text{body}_{Q_b}) \subseteq \text{body}_{Q_a}$ iff $Q$ implies $\mathcal{B}_a \rightarrow A$ (and $\mathcal{B}_a \rightarrow Z$). The last step follows from Corollary 3.1.6. □

Example 25 Assume a database schema containing relations $\text{Student}(\text{school}, \text{dept}, \text{name})$ and $\text{Prof}(\text{school}, \text{dept}, \text{name})$, and consider the following two conjunctive queries.

$$Q_{25}(A, B, C) := \text{Prof}(\text{'UW'}, A, B), \text{Student}(\text{'UW'}, A, C)$$
$$Q_{26}(A, B, C) := \text{Prof}(\text{'UW'}, A, B), \text{Student}(\text{'UW'}, A, C),$$
$$\text{Prof}(F, D, B), \text{Student}(F, E, C)$$

Figure 3.1 shows a sample database instance $D_5$, while Figure 3.2 shows the relation $R_{10}$ resulting from evaluating either query $Q_{25}$ or $Q_{26}$ over $D_5$. The reader can verify that relation $R_{10}$ satisfies $A \rightarrow C$. 

Figure 3.1: Database $D_5 = \{\text{Prof, Student}\}$
Query $Q_{27}$ embodies the appropriate query for testing directly—per Theorem 3.1.3—whether $Q_{25}$ satisfies the MVD $A \rightarrow C$.

$$Q_{27}(A, B, C) := \text{Prof('UW', A, B)}, \text{Student('UW', A, C')}, \text{Prof('UW', A, B'')}, \text{Student('UW', A, C)}$$

The following homomorphisms $h : Q_{25} \rightarrow Q_{27}$ and $h' : Q_{27} \rightarrow Q_{25}$ prove that $Q_{25}$ and $Q_{27}$ are equivalent, and so $Q_{25}$ satisfies $A \rightarrow C$.

$$h := \{A/A, B/B, C/C\} \quad h' := \{A/A, B/B, C/C, C'/C, B''/B\}$$

Similarly, query $Q_{28}$ embodies the appropriate query for testing directly—per Theorem 3.1.3—whether $Q_{26}$ satisfies the MVD $A \rightarrow C$.

$$Q_{28}(A, B, C) := \text{Prof('UW', A, B)}, \text{Student('UW', A, C')}, \text{Prof(F', D', B)}, \text{Student(F', E', C')}, \text{Prof('UW', A, B')}, \text{Student('UW', A, C)}, \text{Prof(F'', D'', B'')}, \text{Student(F'', E'', C)}$$

The following homomorphisms $h'' : Q_{26} \rightarrow Q_{28}$ and $h''' : Q_{28} \rightarrow Q_{26}$ prove that $Q_{26}$ and $Q_{28}$ are equivalent, and so $Q_{26}$ also satisfies $A \rightarrow C$.

$$h'' := \{A/A, B/B, C/C, D/A, E/A, F/'UW'\}$$
$$h''' := \{A/A, B/B, B''/B, C/C, C'/C, D'/D, D''/D, E'/E, E''/E, F'/F, F''/F\}$$

Alternatively, we could have deduced that $Q_{26}$ satisfies $A \rightarrow C$ by instead recognizing that $Q_{25}$ and $Q_{26}$ are equivalent ($Q_{25}$ is a minimal form of $Q_{26}$).

### A Hypergraph-based Characterization of Query-Implied MVDs

Corollary 3.1.4 provides a straightforward homomorphic condition for testing query-implied MVDs. However, homomorphisms are sometimes difficult to visualize for
large queries. We now provide an alternate characterization of query-implied dependencies based upon the topology of a query’s hypergraph; this characterization is both easier to visualize and provides more intuition into the role that MVDs play in the normal form we define later in this section. It also simplifies many of our proofs. As a simple example, we can see that $Q_{25}$ in Example 25 implies $A \leftrightarrow C$ because variables $C$ and $B$ are “separated” by variable $A$ within $\text{body}_{Q_{25}}$. We now formalize this idea of “separation” using graph-theoretic concepts.

Given an undirected graph $G = (U, E)$ and some subset of its vertices\(^2\) $U' \subseteq U$, the subgraph of $G$ induced by $U'$—denoted $G|_{U'}$—is formed from $G$ by deleting the vertices not in $U'$ as well as any edges incident to them. Given vertices $s, t \in U$, an $(s, t)$-\textit{path} in $G$ is a sequence of $k \geq 1$ adjacent edges starting at $s$ and ending at $t$. Given $X \subseteq U$ and $s, t \in (U \setminus X)$, we call $X$ an $(s, t)$-\textit{articulation set} of $G$ if the subgraph $G|_{U \setminus X}$ does not contain any $(s, t)$-paths.\(^3\) Given disjoint sets $X, S, T \subseteq U$, we call $X$ a weak $(S, T)$-\textit{articulation set} if there exists some pair $s \in S$ and $t \in T$ for which $X$ is an $(s, t)$-articulation set, and we call $X$ a strong $(S, T)$-\textit{articulation set} if it is an $(s, t)$-articulation set for every pair $s \in S$ and $t \in T$.

A (undirected) \textit{hypergraph} is a pair $H = (U, F)$ where $U$ is a set of vertices and $F$ is a set of non-empty subsets of $U$, called \textit{hyperedges}. Given some set $U' \subseteq U$, the induced subgraph $H|_{U'}$ is formed from $H$ by deleting the vertices not in $U'$ and then deleting any hyperedge that has become empty. Given vertices $s, t \in U$, an $(s, t)$-path in $H$ is a sequence of $k \geq 1$ hyperedges $f_1, \ldots, f_k$ such that $s \in f_1$, $t \in f_k$, and $f_j \cap f_{j+1} \neq \emptyset$ for $j \in [1, k-1]$.\(^6\) We define the $(s, t)$- and $(S, T)$-articulation sets for hypergraphs in terms of $(s, t)$-paths exactly as we did for standard graphs.

**Example 26** Consider the two hypergraphs depicted in Figure 3.3. The set $\{A\}$ is a $(B, C)$-articulation set in $H_{25}$ but not in $H_{26}$. Within $H_{26}$, the set $\{B, F\}$ is a weak (but not strong) $(E, AD)$-articulation set, while $\{C, F\}$ is a strong $(E, AD)$-articulation set.

Associated with each conjunctive query $Q$ we define the \textit{query hypergraph} $H^Q$ as follows. The query variables $B$ from $Q$ form the vertex set of $H^Q$. The hyperedges correspond to atoms of $\text{body}_Q$ such that for each atom $R_i(\overline{X}_i)$ there is a hyperedge $f$ containing precisely the set of variables in $\overline{X}_i$.

**Example 27** The query hypergraphs $H^{Q_{25}}$ and $H^{Q_{26}}$ are equal to the hypergraphs $H_{25}$ and $H_{26}$, respectively, from Figure 3.3. Observe that the query constant ‘UW’ does not appear in the hypergraphs.

\(^2\)To avoid confusion, we use $U$ rather than $V$ to denote the set of graph vertices since we already use $V$ for the output attributes of CEQs.

\(^3\)An articulation vertex (or set) is typically defined as vertex (or set of vertices) whose deletion increases the number of connected components in the graph [11, 58]. In network flow analysis, an $(s, t)$-cut is defined as a set of edges whose deletion disconnects $s$ from $t$ [84, 93]. Although our definition of an $(s, t)$-articulation set is similar in spirit to each of these concepts, it differs in the case where the initial graph is disconnected—specifically, we do not assume that $s$ and $t$ were connected in the original graph $G$. 

78
Given a hypergraph \( H = (U, F) \) and a database \( D \), an embedding of \( H \) into \( D \) is a mapping \( \gamma : U \to \text{dom} \) such that for each hyperedge \( f \in F \), there exists some tuple \( t \) in some relation in \( D \) such that all of the constants in \( \gamma(f) \) occur in \( t \).

**Lemma 3.1.8** Given a database \( D \) and conjunctive query \( Q \), any query embedding \( \gamma : \text{body}_Q \to D \), is also an embedding of hypergraph \( H^Q \) into \( D \).

**Lemma 3.1.9** Given a database \( D \), a hypergraph \( H = (U, F) \), and a set \( U' \subseteq U \), any hypergraph embedding \( \gamma : U \to D \) is also an embedding of the induced hypergraph \( H|_{U'} \) into \( D \).

**Theorem 3.1.10** Given a minimal conjunctive query \( Q(\overline{W}) \) and disjoint attribute sets \( X, Y \in \mathcal{W} \) query \( Q \) implies MVD \( X \rightarrow Y \) iff \( X \) is a strong \((Y,Z)\)-articulation set of the hypergraph \( H^Q \), where \( Z = \mathcal{W} \setminus (X \cup Y) \).

**Proof.** Follows from Corollary 3.1.4, because any homomorphism \( h : Q \rightarrow Q \) is necessarily isomorphic over \( B \) due to the minimality of \( Q \).

It is trivial to extend Theorem 3.1.10 (and Corollary 3.1.4) to allow constants in sets \( X \) and \( Y \); we assume this slight extension for the MVDs in the next section.

**Example 28** Queries \( Q_{25} \) and \( Q_{26} \) both imply \( A \rightarrow C \) because \( A \) is a \((B,C)\)-articulation set of \( H^{Q_{25}} \). In contrast, \( A \) is not a \((B,C)\)-articulation set of \( H^{Q_{26}} \).

### 3.1.2 Converting CEQs to Normal Form

We now define a normal form for CEQs based upon query-implied MVDs. Our normal form is defined by recursively identifying the core indexes \( \mathcal{I}_i^{\overline{V}} \subseteq \mathcal{I}_i \) at each level \( i \).

**Definition 3.1.11 (Core Indexes)** Given a CEQ \( Q(\overline{I}_1; \ldots ; \overline{I}_d; \overline{V}) \), a length-\( d \) signature \( \overline{W} \), and an integer \( i \in [1,d] \), define the core indexes at level \( i \) relative to \( \overline{W} \)—denoted \( \mathcal{I}_i^{\overline{W}} \)—as follows. Let \( Q_{(i)} \) be the following CQ.

\[
Q_{(i)}(\overline{I}_{1,i}^{\overline{W}}, \overline{I}_{i+1,d}^{\overline{W}}) \leftarrow \text{body}_Q
\] (3.3)
Set $\mathcal{I}_i^\mathcal{F}$ is a minimal subset of $\mathcal{I}_i$ satisfying both $\mathcal{V} \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{I}_{[i,d]}^\mathcal{F} \cup \mathcal{C}$ and the following signature-specific conditions.

<table>
<thead>
<tr>
<th>$\mathcal{I}_i^\mathcal{F}$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>$\mathcal{I}<em>i \subseteq \mathcal{I}</em>{[1,i-1]} \cup \mathcal{I}_{[i,d]}^\mathcal{F} \cup \mathcal{C}$</td>
</tr>
<tr>
<td>$s$</td>
<td>$Q(i) \vdash (\mathcal{I}<em>{[1,i-1]} \cup \mathcal{I}</em>{[i,d]}^\mathcal{F}) \rightarrow \mathcal{I}_{[i+1,d]}^\mathcal{F}$</td>
</tr>
<tr>
<td>$n$</td>
<td>$Q(i) \vdash \mathcal{I}<em>{[1,i-1]} \rightarrow \mathcal{I}</em>{[i,d]}^\mathcal{F}$</td>
</tr>
</tbody>
</table>

We denote by $\mathcal{I}_i^\mathcal{F}$ the tuple formed by deleting from $\mathcal{I}_i$ all attributes not in $\mathcal{I}_i^\mathcal{F}$.

Observation 3.1.12 *Constant columns are never core; that is, $\mathcal{I}_{[1,d]}^\mathcal{F} \subseteq \mathcal{C}$.*

Observation 3.1.13 *The sets of core indexes at different levels are disjoint.*

Observation 3.1.14 *If $\mathcal{I}_d^\mathcal{F} = \mathcal{I}_d \cap \mathcal{V} \cap \mathcal{B}$, then $\mathcal{I}_d^\mathcal{F} = (\mathcal{I}_d \cap \mathcal{V} \cap \mathcal{B}) \setminus \mathcal{I}_{[1,i-1]}$.*

Lemma 3.1.15 *Definition 3.1.11 determines a unique minimal set $\mathcal{I}_i^\mathcal{F}$.*

**Proof.** First, let $Q(i)(\mathcal{I}_{[1,i]} \setminus \mathcal{I}_{[i+1,d]}^\mathcal{F})$ be a minimal CQ equivalent to $Q(i)$ from equation 3.3. Next, let a “candidate for $\mathcal{I}_i^\mathcal{F}$” denote any set $X \subseteq \mathcal{I}_i$ such that if we choose $\mathcal{I}_i^\mathcal{F} := X$ then all conditions in Definition 3.1.11 except for minimality are satisfied. We now show that if sets $X_1, X_2$ are both candidates for $\mathcal{I}_i^\mathcal{F}$, then $X_1 \cap X_2$ is also a candidate for $\mathcal{I}_i^\mathcal{F}$. If $\mathcal{V} \subseteq \mathcal{I}_{[1,i-1]} \cup X_1 \cup \mathcal{C}$ and $\mathcal{V} \subseteq \mathcal{I}_{[1,i-1]} \cup X_2 \cup \mathcal{C}$ then $\mathcal{V} \subseteq \mathcal{I}_{[1,i-1]} \cup (X_1 \cap X_2) \cup \mathcal{C}$ follows directly. Now consider the signature-specific conditions.

Case $\mathcal{I}_i^\mathcal{F} = b$:

$\mathcal{I}_i \subseteq \mathcal{I}_{[1,i-1]} \cup X_1$ and $\mathcal{I}_i \subseteq \mathcal{I}_{[1,i-1]} \cup X_2$ implies $\mathcal{I}_i \subseteq \mathcal{I}_{[1,i-1]} \cup (X_1 \cap X_2)$.

Case $\mathcal{I}_i^\mathcal{F} = s$:

The MVD $\mathcal{Q}(i) = \mathcal{I}_{[1,i-1]} \cup (X_1 \cap X_2) \rightarrow \mathcal{I}_{[i+1,d]}^\mathcal{F}$ cannot be derived from axioms for MVDs [6, Sect. 8.3], but can be reasoned from the query structure as follows.

1. By candidacy of $X_1$, $\mathcal{Q}(i) \vdash \mathcal{I}_{[1,i-1]} \cup X_1 \rightarrow \mathcal{I}_{[i+1,d]}^\mathcal{F}$. By Theorem 3.1.10, $\mathcal{I}_{[1,i-1]} \cup X_1$ is a strong $(\mathcal{I}_{[i+1,d]}, (\mathcal{I}_i \setminus X_1))$-articulation set of $H^{Q(i)}$.
2. By candidacy of $X_2$, $\mathcal{Q}(i) \vdash \mathcal{I}_{[1,i-1]} \cup X_2 \rightarrow \mathcal{I}_{[i+1,d]}^\mathcal{F}$. By Theorem 3.1.10, $\mathcal{I}_{[1,i-1]} \cup X_2$ is a strong $(\mathcal{I}_{[i+1,d]}, (\mathcal{I}_i \setminus X_2))$-articulation set of $H^{Q(i)}$.
3. The two articulation sets together imply that deleting $\mathcal{I}_{[1,i-1]} \cup (X_1 \cap X_2)$ from $H^{Q(i)}$ causes the two sets $\mathcal{I}_{[i+1,d]}^\mathcal{F}$ and $\mathcal{I}_i \setminus (X_1 \cap X_2)$ to occur in separate partitions of the remaining hypergraph. The MVD then follows from Theorem 3.1.10.
Case $\emptyset_i = n$:

Similar to case $\emptyset_i = s$. The candidates of $X_1$ and $X_2$ imply two MVDs which imply two strong articulation set of $H^{Q(i)}$. This time, the two articulation sets together imply that deleting $I_{[i, i-1]}$ from $H^{Q(i)}$ causes the four sets $X_1 \setminus X_2$, $X_2 \setminus X_1$, $I_i \setminus (X_1 \cup X_2)$, and $(X_1 \cap X_2) \cup I_{[i+1, d]}$ to occur in separate partitions of the remaining hypergraph. The MVD $Q(i) \vdash I_{[i, i-1]} \Rightarrow (X_1 \cap X_2) \cup I_{[i+1, d]}$ then follows from Theorem 3.1.10.

\[ \square \]

We say that CEQ $Q(\overline{I}_1; \ldots; \overline{I}_d; \overline{V})$ is in $\overline{\emptyset}$-normal form (\(\overline{\emptyset}$-NF) if $I_{[1, d]} = I_{[1, d]}$, and so an arbitrary CEQ is converted to $\overline{\emptyset}$-NF by deleting all non-core indexes from the query head. Assuming that the original query was valid (cf. Lemma 3.0.1), Definition 3.1.11 implies $(\mathcal{V} \cap \mathcal{B}) \subseteq I_{[1, d]}$ and so the new query is guaranteed to be valid.

**Theorem 3.1.16** Performing $\overline{\emptyset}$-normalization of CEQs is NP-complete.

**Proof.** NP-hardness follows directly from Theorem 3.1.7. Identifying the core indexes at each level can be done in NP time using an algorithm that traverses query hypergraphs (due to Observation 3.1.12, it is not a problem that query hypergraphs do not contain constants).

Case $\emptyset_i = b$: Trivial.

Case $\emptyset_i = s$: Minimize the body of $Q(i)$, then construct hypergraph $H^{Q(i)}$. Delete from $H^{Q(i)}$ all nodes corresponding to variables in the set $I_{[1, i-1]} \cup (I_i \cap \mathcal{V})$. Identify any non-output members of $I_i$ incrementally by traversing the connected components containing $I_{[i+1, d]}$ and deleting the “nearest” member of $I_i$.

Case $\emptyset_i = n$: Minimize the body of $Q(i)$, then construct hypergraph $H^{Q(i)}$. Delete from $H^{Q(i)}$ all nodes corresponding to variables in the set $I_{[1, i-1]}$. Identify $I_i$ by traversing the connected components containing any variable in $(I_i \cap \mathcal{V}) \cup I_{[i+1, d]}$.

\[ \square \]

**Example 29** Consider the following four CEQs.

\[
Q_{20}(A, B; C; C) := \text{ Prof('UW', A, B), Student('UW', A, C)}
\]
\[
Q_{30}(A; C; C) := \text{ Prof('UW', A, B), Student('UW', A, C)}
\]
\[
Q_{31}(A; B, C; C) := \text{ Prof('UW', A, B), Student('UW', A, C), Prof(F, D, B), Student(F, E, C)}
\]
\[
Q_{32}(A; C; C) := \text{ Prof('UW', A, B), Student('UW', A, C), Prof(F, D, B), Student(F, E, C)}
\]
Observe that $Q_{29}$ and $Q_{30}$ have the same body as $Q_{25}$ from Example 28, while $Q_{31}$ and $Q_{32}$ have the same body as $Q_{26}$. Therefore, both $Q_{29}$ and $Q_{31}$ imply $A \rightarrow C$.

Consider signature $sb$. The dependency $A \rightarrow C$ implies that $B$ is not a core index of $Q_{29}$ relative to $sb$. In contrast, queries $Q_{30}$, $Q_{31}$, and $Q_{32}$ are all in $sb$-NF.

Now consider signature $nn$. This time queries $Q_{29}$, $Q_{30}$, and $Q_{32}$ are already in $nn$-NF, but $Q_{31}$ is not because $A \rightarrow C$ implies that $B$ is not a core index.

The next theorem establishes that $\overline{F}$-normalization does not change the complex object obtained after decoding the CEQ result with signature $\overline{F}$. Before we prove this formally, however, consider the following example which we will use to illustrate the core ideas of the proof.

**Example 30** Figure 3.4 shows the encoding relations that result when the four queries from Example 29 are evaluated over the database $D_5$ from Figure 3.1. Query $Q_{30}$ is the $sb$-normal form of $Q_{29}$, and the reader can verify that both encoding relations $R_{11}$ and $R_{12}$ yield the object $\{ \{ \text{Dave}, \text{Xin} \}, \{ \text{Cara}, \text{Julia} \} \}$ when decoded with signature $sb$. Similarly, $Q_{32}$ is the $nn$-normal form of $Q_{31}$, and both encoding relations $R_{13}$ and $R_{12}$ yield the object $\{ \{ \text{Dave}, \text{Xin} \}, \{ \text{Cara}, \text{Julia} \} \}$ when decoded with signature $nn$.

**Theorem 3.1.17** $\overline{F}$-Normalization preserves $\overline{F}$-equivalence.
Proof. Let $Q(\overline{I}_1; \ldots; \overline{I}_d; \overline{V})$ CEQ, and let $Q'(\overline{I}_1^{\overline{f}}; \ldots; \overline{I}_d^{\overline{f}}; \overline{V})$ be its $\overline{f}$-normal form. For every $i \in [0, d]$, let $Q_i$ denote the following CEQ:

$$Q_i(\overline{I}_1; \ldots; \overline{I}_i^{\overline{f}}; \overline{I}_{i+1}; \ldots; \overline{I}_d^{\overline{f}}; \overline{V}) := \text{body}_{Q_i}$$ (3.4)

Because $Q' = Q_0$, we need to show that $Q_0 \in_{\overline{f}} Q$, which we do by induction.

Base Case: Equation 3.4 implies $Q_d = Q$, and so $Q_d \in_{\overline{f}} Q$ is trivial.

Induction Hypothesis: Suppose there exists some $i \in [1, d]$ such that $Q_i \in_{\overline{f}} Q$.

Inductive Step: We need to show that $Q_{i-1} \in_{\overline{f}} Q_i$. For any database $D$, let $\Gamma$ and $\Phi$ denote the sets of embeddings of $\text{body}_{Q_{i-1}}$ and $\text{body}_{Q_i}$ into $D$; because $Q_{i-1}$ and $Q_i$ have the same body, $\Gamma$ and $\Phi$ contain the same embeddings. W.l.o.g., assume that $\overline{I}_i = \overline{I}_i^\Phi \cdot \overline{J}_i$, where $\overline{J}_i$ is the set of non-core indexes in $\overline{I}_i$. The encoding relations $(Q_{i-1})^D$ and $(Q_i)^D$ can now be characterized as follows.

$$S((Q_{i-1})^D) = Q_{i-1}(\overline{I}_1; \ldots; \overline{I}_{i-1}; \overline{I}_i^{\overline{f}}; \overline{I}_{i+1}; \ldots; \overline{I}_d^{\overline{f}}; \overline{V})$$ (3.5)

$$T((Q_{i-1})^D) = \{ \gamma[\overline{I}_{[1,i-1]}^{\overline{f}}] : \gamma \in \Gamma \}$$ (3.6)

$$S((Q_i)^D) = Q_i(\overline{I}_1; \ldots; \overline{I}_{i-1}; \overline{I}_i^\Phi; \overline{I}_{i+1}; \ldots; \overline{I}_d^{\overline{f}}; \overline{V})$$ (3.7)

$$T((Q_i)^D) = \{ \phi[\overline{I}_{[1,i-1]}^{\overline{f}}] \overline{J}_i \overline{I}_{[i+1,d]}^{\overline{f}} : \phi \in \Phi \}$$ (3.8)

Abusing notation, we will now use $\Gamma$ and $\Phi$ in place of the encoding relations $(Q_{i-1})^D$ and $(Q_i)^D$ by implicitly associating schema $S((Q_{i-1})^D)$ with $\Gamma$ and $S((Q_i)^D)$ with $\Phi$. Then, we need to show that $\Gamma \in_{\overline{f}} \Phi$.

Case $\overline{s}_i = b$: 
By Definition 3.1.11, $\overline{J}_i \subseteq \overline{I}_{[1,i-1]} \cup \overline{C}$. Because $\overline{I}_{[1,i-1]}$ functionally determines $\overline{J}_i$, removing $\overline{J}_i$ from the head does not affect the cardinality or contents of any encoding sub-relations; hence, there is a trivial isomorphism between the tuples of $\Gamma$ and $\Phi$ that proves $\Gamma \in_{\overline{f}} \Phi$.

Case $\overline{s}_i = s$: 
For each value $\overline{a} \in \text{adom}(\overline{I}_{[1,i-1]}, \Gamma) = \text{adom}(\overline{I}_{[1,i-1]}, \Phi)$ let $C_{\overline{a}}$ be a $\overline{f}_{[i,d]}$-certificate rooted by an initially empty set node (see Definition 2.2.5). We will incrementally construct $C_{\overline{a}}$ until it proves that $\Gamma[\overline{a}] \in_{\overline{f}_{[i,d]}} \Phi[\overline{a}]$. Because $\text{adom}(\overline{I}_{[1,i-1]}, \Gamma) = \text{adom}(\overline{I}_{[1,i-1]}, \Phi)$, it is then trivial to construct the upper levels of a $\overline{f}$-certificate proving $\Gamma \in_{\overline{f}} \Phi$.

Let $\{\overline{b}_1, \ldots, \overline{b}_k\}$ be the set of values in $\text{adom}(\overline{I}_i, \Gamma[\overline{a}]) = \text{adom}(\overline{I}_i, \Phi[\overline{a}])$, and let $\{\overline{b}_1^1, \ldots, \overline{b}_1^{c_1}, \ldots, \overline{b}_k^1, \ldots, \overline{b}_k^{c_k}\}$ be the set of values in $\text{adom}(\overline{I}_i, \overline{J}_i, \Phi[\overline{a}])$. For each $\overline{b}_j$, the sub-relation $\Gamma[\overline{a} \overline{b}_j]$ encodes an object of sort $(\overline{f}_{[i+1,d]}; d - i)$ occurring in the set at level $i$. By Definition 3.1.11, relation $\Phi$ satisfies $\overline{I}_{[1,i-1]} \cup \overline{I}_i \rightarrow \overline{I}_{[i+1,d]}^{\overline{f}}$, which—by Corollaries 3.1.5 and 3.1.6—implies that
Φ[α] satisfies $I_i^\neg \rightarrow I_{i+1,d}^\neg$ (and $I_i^\neg \rightarrow J_i$). It follows from that all of the sub-relations $Φ[α\overline{b}_j\overline{c}_j^{\neg i}]$, ..., $Φ[α\overline{b}_j\overline{c}_j^{\neg i}]$ are identical, each other and to the sub-relation $Γ[α\overline{b}_j]$. For each $\overline{c}_j^{\neg i}$ add to $C_\pi$ the mapping $f(\overline{b}_j\overline{c}_j^{\neg i}) := \overline{b}_j$, as well as a child $\overline{s}_{i+1,d}^\neg$-certificate proving that $Φ[α\overline{b}_j\overline{c}_j^{\neg i}] = Γ[α\overline{b}_j]$ (which is trivial, because relations $Φ[α\overline{b}_j\overline{c}_j^{\neg i}] = Γ[α\overline{b}_j]$). Then, add the mapping $f'(\overline{b}_j) := \overline{b}_j \cdot \overline{c}_j$ (we could choose any $\overline{c}_j$). Certificate $C_\pi$ is complete when this has been performed for all values of $\overline{b}_j$.

Case $\overline{s}_i = n$:

The proof is almost identical to case $\overline{s}_i = s$. Because $I_{i+1,i-1} \rightarrow I_{i+1,d}^\neg$ implies $I_{i+1,i-1} \cup I_i^\neg \rightarrow I_{i+1,d}^\neg$, all of the relations $Φ[α\overline{b}_j\overline{c}_j^{\neg i}], ..., Φ[α\overline{b}_j\overline{c}_j^{\neg i}]$ are identical, each other and to the sub-relation $Γ[α\overline{b}_j]$. Furthermore, $I_{i+1,i-1} \rightarrow I_{i+1,d}^\neg$ also implies that across all values of $j$, the sets $\{\overline{c}_j, ..., \overline{c}_j\}$ are identical. Therefore, the degree of replication remains constant across all values of $\overline{b}_j \in \text{dom}(I_i, Γ[α])$.

\[\square\]

### 3.2 CEQ Encoding-Equivalence

In this section we fully characterize $\overline{s}$-equivalence of CEQs by tailoring the traditional homomorphism test for CQs to encoding queries.

**Definition 3.2.1 (Index-Covering Homomorphism)** Given two conjunctive encoding queries \(Q(\overline{I}_1; ..., \overline{I}_d; \overline{V})\) and \(Q'(\overline{I}_1; ..., \overline{I}_d; \overline{V}')\) of equal depth, an index-covering homomorphism from \(Q'\) to \(Q\) is a mapping \(h : B' \rightarrow B \cup C\) (suitably extended to tuples, sets, and atoms, and with identity on constants) satisfying

1. \(h(\text{body}_{Q'}) \subseteq \text{body}_Q\),
2. \(h(\overline{V}') = \overline{V}\), and
3. \(\forall i \in [1, d] : \overline{I}_i \subseteq h(\overline{I}_i')\).

Comparing Definitions 3.2.1 and 3.1.1, we note that if \(h\) is an index-covering homomorphism from CEQ \(Q'\) to CEQ \(Q\), then \(h\) is also a CQ homomorphism \(h : Q'(\overline{V}') \rightarrow Q(\overline{V})\), where \(Q'(\overline{V}')\) and \(Q(\overline{V})\) denote the CQs formed from \(Q'\) and \(Q\) by restricting the head tuples to \(\overline{V}'\) and \(\overline{V}\), respectively. Conversely, if \(h'\) is a CQ homomorphism from CQ \(Q''(\overline{W}'')\) to CQ \(Q''(\overline{V}'')\), then \(h'\) is also an index-covering homomorphism from the depth-one CEQ \(Q''(\overline{W}''; \overline{V}'')\) to the depth-one CEQ \(Q''(\overline{W}''; \overline{W}'')\). For this reason, we use the same notation \(h : Q' \rightarrow Q\) to denote index-covering homomorphisms as for CQ homomorphisms, and and we will clearly indicate within the context which type of homomorphism (and queries) we are
discussing. The next theorem and its corollary follow from well-known results for conjunctive queries.

**Theorem 3.2.2** Given conjunctive encoding queries $Q$ and $Q'$, each index-covering homomorphism from $Q'$ to $Q$ corresponds to a tuple in the result of evaluating $Q'$ over a database isomorphic to the body of $Q$.

**Corollary 3.2.3** Determining the existence of an index-covering homomorphism is NP-complete.

We now combine our tailored homomorphism with the normal form from the previous section to obtain a necessary and sufficient test for $\equiv^F$-equivalence. The necessity proof for the following theorem is intricate—as we will discuss further in Section 3.3, just considering nested sets is already complex [77], and handling arbitrary signatures requires integrating three different proof techniques (for sets, bags, and normalized bags). For readability, we give only a short sketch here and defer the full proof of necessity to Appendix A.

**Theorem 3.2.4** Two conjunctive encoding queries are $\equiv^F$-equivalent iff there exists index-covering homomorphisms in both directions between their $\equiv^F$-normal forms.

**Proof (sketch).** Assume without loss of generality that $Q$ and $Q'$ are already in $\equiv^F$-normal form (justified by Theorem 3.1.17).

If index-covering homomorphisms exist in both directions, then for any database $D$ the encoding relations $(Q)^D$ and $(Q')^D$ differ at most by permutation of attributes within each index level. A $\equiv^F$-certificate proving $(Q)^D \equiv^F (Q')^D$ is therefore straightforward to construct, since each node is simply an isomorphism between sub-relations; $Q \equiv^F Q'$ follows immediately.

We now sketch the argument for the existence of index-covering homomorphisms (see Appendix A for the full proof). We start by defining a degenerate class of “pseudo-trivial” $\equiv^F$-certificate nodes that imply certain regularities between the encoding relations that they relate. The overall goal is then to construct a canonical database $D$ out of the bodies of both $Q$ and $Q'$ such that if we can find a $\equiv^F$-certificate between $(Q)^D$ and $(Q')^D$ that contains these pseudo-trivial nodes, we can use it to generate the index-covering homomorphisms in each direction. We then argue for the existence of such a certificate by proving that any $\equiv^F$-certificate between $(Q)^D$ and $(Q')^D$ must contain these trivial nodes. The proof uses induction level-by-level from inside-out to prove triviality of certificate nodes, and the argument differs depending upon the semantics of the current level. Bag equivalence is based upon counting, and so for bag levels we rely upon certain algebraic properties of polynomials combined with a procedure to inflate the cardinalities of constants in any database in a regular way such that non-pseudo-trivial certificate nodes would imply differing element cardinalities, violating bag equivalence. Set equivalence is based upon existence, and so for set levels we use the query bodies to inject large amounts of symmetry into the construction of $D$, with the constants corresponding to the index
variables in $Q$ and $Q'$ forming the “axes” of the symmetry; we then show that non-pseudo-trivial certificate nodes would imply the existence (non-existence) of extra (missing) “canonical sub-objects,” violating set equivalence. Finally, normalized bag equivalence exhibits both count- and existence-based features, and so for these levels the argument synthesizes both the polynomial and the symmetry techniques.

Corollary 3.2.5 Testing $\overline{\Xi}$-equivalence of CEQs is NP-complete.

Corollary 3.2.6 Testing equivalence of $\text{COCQL}^{(f_b,f_s,f_n)}$-queries is NP-complete.

Corollary 3.2.6 is the culmination of this chapter, and one of the major results within this thesis. It is important because it demonstrates that as long as queries do not compare aggregated values, non-scalar forms of nesting do not increase the hardness of the query equivalence problem.

Example 31 Consider again queries $Q_{29}$–$Q_{32}$ from Example 29. Queries $Q_{30}$ and $Q_{32}$ are each in both sb-NF and nn-NF, and so the following index-covering homomorphisms $h : Q_{30} \rightarrow Q_{32}$ and $h' : Q_{32} \rightarrow Q_{30}$ prove $Q_{30} \approx_{sb} Q_{32}$ and $Q_{30} \approx_{nn} Q_{32}$.

$$
\begin{align*}
  h &= \{A/A, B/B, C/C\} \\
  h' &= \{A/A, B/B, C/C, F'/\text{UW}', D/A, E/A\}
\end{align*}
$$

Performing sb-normalization on $Q_{29}$ yields $Q_{30}$, and so $Q_{29} \approx_{sb} Q_{32}$. In contrast, $Q_{31}$ is in sb-NF and there clearly cannot exist an index-covering homomorphism from $Q_{32}$ to $Q_{31}$, so $Q_{31} \not\approx_{sb} Q_{32}$ follows by Theorem 3.2.4. By similar analysis we conclude $Q_{29} \not\approx_{nn} Q_{30} \approx_{nn} Q_{31} \approx_{nn} Q_{32}$. The encoding relations in Figure 3.4 illustrate a counter-example proving both $Q_{31} \not\approx_{sb} Q_{32}$ and $Q_{29} \not\approx_{nn} Q_{30}$.

3.3 Relevant Literature

In this section we survey literature relevant to deciding various forms of equivalence between conjunctive queries operating within a data model containing only flat relational structures. The reader is referred to the previous chapter for literature pertaining to query languages that construct nested objects. In particular, in Section 2.4.4 we discuss Levy and Suciu’s reduction of COQL equivalence to testing strong simulation between CQs [77], and so we do not discuss that work further here, even though it is in many ways the closest research to the CEQ encoding-equivalence problem which is our focus in this chapter.
3.3.1 Conjunctive Queries under Set Semantics

Equivalence and containment of conjunctive queries under set semantics was first studied by Chandra and Merlin [16] who show that the two problems are mutually reducible, and that they are each mutually reducible to the NP-complete problem of finding homomorphisms between queries. They also show that every query has a unique minimal form, and that the minimal forms of equivalent queries are isomorphic (hence minimization is also NP-complete). Chekuri and Rajaraman [20] take a previously-known result that homomorphisms are computable in PTIME when the query bodies correspond to acyclic hypergraphs [6], and they generalize the notion of acyclicity to query width. Their work furnishes algorithms for the containment, equivalence, and minimization problems that run in time polynomial in $n^k$, for queries of size $n$ and width $k$. These efficient algorithms are directly applicable to the problems of performing $\bar{\Sigma}$-normalization and finding index-covering homomorphisms that we discuss in this chapter.

3.3.2 Other Processing Semantics

Chaudhuri and Vardi [19] and Ioannidis and Ramakrishnan [64] independently propose bag/bag-set semantics as a way to model the input to cardinality-sensitive aggregation functions and proceed to study the query equivalence and containment problems under the new query processing semantics. Under bag semantics, conjunctive queries are equivalent precisely when they are isomorphic; the obvious corollaries are that every conjunctive query is already minimal under bag semantics, and that testing bag equivalence is GraphIsomorphism-complete. Under bag-set semantics (bag semantics where database relations are known to be sets), queries are equivalent if and only if they are variable-isomorphic; that is, query minimization simply entails removing duplicate relational atoms (linear time), and two queries are equivalent under bag-set semantics precisely when their minimized forms are equivalent under bag semantics. Chaudhuri and Vardi also show a simple reduction from bag equivalence to bag-set equivalence, which justifies our design of $\text{COCQL}^{\bar{\Sigma}}$ in Section 2.1.3 as a bag semantic algebra operating over a database of relations (sets of tuples).

Although both Chaudhuri and Vardi as well as Ioannidis and Ramakrishnan characterize the CQ equivalence problem under bag/bag-set semantics, the CQ containment problem under bag/bag-set semantics is still open, although known to be at least $\Pi_2^p$-hard [19]. Ioannidis and Ramakrishnan show bag containment to be linear time when queries do not contain repeated predicate names, and undecidable when extended to unions of CQs. More recently, Jayram, Kolaitis and Vee show that bag containment is undecidable for CQs extended with inequality predicates (i.e., $\neq$) [66].

$\Pi_2^p$ is a class within the second tier of the polynomial hierarchy. A decision problem is in $\Pi_2^p$ if its complement problem is in $\Sigma_2^p$, the class of problems that permit an NP-time reduction to a problem that is in NP [43, Sect. 7.2].
Variations of bag/bag-set semantics have also been proposed in the literature. Grumbach et al. relax bag-set semantics in order to model the input to certain normalizing aggregation functions such as \texttt{AVG}. Under their relaxed semantics, two queries are equivalent if over every database they return two bags containing the same elements with the same relative cardinalities. They find that CQs are equivalent under these semantics precisely when the query bodies are isomorphic “modulo a product” which roughly means that they differ at most by the addition of certain Cartesian product operations [50]. Cohen also relaxes bag-set semantics in order to study equivalence of SQL queries that contain a nested \texttt{SELECT DISTINCT} block. She proposes “combined semantics” as a generalization of bag-set semantics in which cardinality depends only on a specified subset of the query variables, and deduces that CQs are equivalent under these semantics precisely when there exist homomorphisms in each direction that are isomorphic with respect to the designated “multi-set variables” [22].

Our results in this chapter for characterizing encoding-equivalence between conjunctive encoding queries generalizes CQ equivalence under all of these previously-proposed processing semantics. More specifically, given two CQs $Q(\overline{V})$ and $Q'(\overline{V}')$, testing $Q \equiv Q'$ under

- \textit{set semantics} reduces to $Q(\overline{V}; \overline{V}) \equiv_s Q'(\overline{V'}; \overline{V'})$;
- \textit{bag-set semantics} reduces to $Q(\overline{B}; \overline{V}) \equiv_b Q'(\overline{B'}; \overline{V'})$ where $B$ and $B'$ are the query body variables;
- \textit{bag-set semantics modulo a product} reduces to $Q(\overline{B}; \overline{V}) \equiv_s Q'(\overline{B'}; \overline{V'})$; and
- \textit{combined semantics} reduces to $Q(\overline{V} \cup \overline{M}; \overline{V}) \equiv_b Q'(\overline{V'} \cup \overline{M'}; \overline{V'})$ where $M$ and $M'$ are the specified multi-set variables.

### 3.3.3 Aggregation Queries

The equivalence and containment problems for queries containing non-nested aggregation have been studied before. Cohen, Nutt, and Sagiv address the equivalence problem for conjunctive queries followed by a single aggregation operation applying one of the functions \texttt{SUM}, \texttt{COUNT}, \texttt{COUNT-DISTINCT}, \texttt{MAX}, or \texttt{MIN} [25]. When the query bodies do not contain comparison predicates, the equivalence problem for \texttt{SUM} and \texttt{COUNT} queries reduces to equivalence under bag semantics, while the equivalence problem for \texttt{MAX}, \texttt{MIN}, and \texttt{COUNT-DISTINCT} reduces to equivalence under set semantics. When the query bodies contain comparison predicates, they show the problem to be $\Pi^p_2$-complete for \texttt{MAX} queries, \texttt{MIN} queries, and \texttt{COUNT-DISTINCT} queries under a slight syntactic restriction that limits interaction between the aggregated variables and the variables involved in comparison predicates. For \texttt{SUM} and \texttt{COUNT} queries with comparisons they show the problem to be in PSPACE. These authors also extend their detailed study of aggregation functions to query bodies containing disjunction [26] and safe atomic negation [27]. In Cohen’s doctoral dissertation she uses abstract algebra concepts such as commutative semigroups, idempotency,
and abelian groups to define five properties of aggregation functions—\emph{shiftable}, \emph{order-decidable}, \emph{decomposable}, \emph{singleton-determining}, and \emph{expandable}—with which she taxonomizes commonly-used aggregation functions [21]. She then generalizes previous results for equivalence of queries with specific aggregation functions to black-box functions characterized by this taxonomy, and uses these equivalence results to characterize the problem of rewriting non-nested aggregation queries over non-nested aggregation views, an extension that is fleshed out further within a subsequent journal publication [24].

Grumbach, Rafanelli, and Tininini also consider the problem of rewriting non-nested aggregation queries over non-nested aggregation views, and as part of this work characterize the equivalence problem for queries that apply the \textsc{avg} or \textsc{percent} aggregation functions to the result of conjunctive queries [51]. Their characterization reduces this problem to equivalence of CQs under the relaxation of bag-set semantics discussed in Section 3.3.2. In related work, Grumbach and Tininini consider the view rewriting problem for the restricted case where the query and view languages are aggregation queries over a single relation (i.e., no joins) [53]; this case has very practical application to OLAP-style queries data warehouses with star schemas. Their main contribution is a polynomial-time rewriting algorithm which they prove to be complete for finding single-view rewritings of queries containing the \textsc{sum}, \textsc{count}, and \textsc{product} aggregation functions.

The various works by both Cohen et al. and Grumbach et al. surveyed above only considered \emph{non-nested} aggregation queries and views. More specifically, their various query and view languages all correspond to relational algebra expressions of the form

\begin{equation}
\Pi^{z_f(u)}(E)
\end{equation}

where $E$ is a relational algebra query not containing aggregation functions. Observe that when generalized projection only occurs as the top-most query operator, \emph{every grouping variable is also an output variable}. This syntactic restriction is significant, and highlights the orthogonality between our work and the work surveyed in this section. Whereas previous work focuses on understanding and characterizing specific aggregation operations, our work focuses on the interaction between aggregation and query nesting. It is precisely the ability to project away attributes that have been used for grouping at lower levels that makes the equivalence problem for nested queries a non-trivial extension of work on non-nested aggregation queries. Our abstraction of aggregation functions as collection constructors is comparatively primitive, but arguably necessary both as a simplifying assumption and in order to preserve decidability for the query equivalence problem. Other authors have shown that too much domain-specific knowledge about the functioning of particular aggregation functions quickly causes undecidable implication of aggregation constraints [75, 97]. Within our context, the obvious corollary is that the query equivalence problem becomes undecidable when queries contain aggregation functions whose results can be both arbitrarily nested and compared with explicit (and possibly, only implicit) higher-order comparisons.
Chapter 4

Equivalence of Conjunctive Queries with Outer Joins

Our goal in this thesis is to characterize equivalence of conjunctive object-constructing queries. In Chapter 2 we reduced this problem to deciding encoding-equivalence between hierarchical conjunctive encoding queries. In Chapter 3 we showed that encoding-equivalence between CEQs generalizes query equivalence between CQs. Therefore, before we address encoding-equivalence between hierarchical CEQs (HCEQs), it is helpful for us to first address query equivalence between hierarchical CQs (HCQs).

In Section 2.3.1 we defined the semantics of HCQs in terms of hierarchical applications of terminal embeddings of the query body into the database instance. There is a strong relationship between HCQs and relational algebra expressions containing the left outer-join (LOJ) operator. In Section 4.1 we show that extending the conjunctive algebra with LOJ yields a language whose expressive power corresponds to a generalization of HCQs that we call directed acyclic CQs (DACQs). In Sections 4.3–4.5 we address the DACQ equivalence problem (of which the HCQ equivalence problem is a special case). We end the chapter by surveying relevant literature in Section 4.6.

4.1 Conjunctive Queries with Outer Joins

In this section we propose two languages for conjunctive queries with outer joins. In Section 4.1.1 we formally define the SPCL algebra, which extends the conjunctive algebra with a left outer-join operator. Next, in Section 4.1.2 we define directed acyclic CQs, a straightforward generalization of hierarchical CQs, and we prove that DACQs have the same expressive power as SPCL expressions.
The SPCL Algebra

The conjunctive relational algebra—i.e., the operators base relation, selection, duplicate-eliminating projection, and Cartesian product along with the syntactic extensions conjunctive selection and join from Figure 2.5—is sometimes called the SPC algebra. We define the SPCL algebra to be language formed by extending the SPC algebra with the left outer-join operator \[42\].

The left outer join (LOJ) of relation \(R\) with \(S\)—denoted \(R \bowtie_p S\)—is equivalent to the outer union of the inner- and anti-joins of \(R\) and \(S\),

\[
R \bowtie_p S \equiv (R \bowtie_p S) \cup (R \bowrel_p S) \equiv (R \bowtie_p S) \cup (R - (R \bowcurlyvee_p S))
\] (4.1)

where the outer union operator \(\cup\) pads the tuples originating from the anti-join with a special null constant\(^1\) \(\bot\) so that they have the same arity as the tuples originating from the inner join. Alternatively, the LOJ operator can be defined using the notion of tuple subsumption. Given two tuples \(t_1\) and \(t_2\) of the same arity, \(t_2\) subsumes \(t_1\) if \(t_2\) contains strictly more \(\bot\) values than \(t_1\), and \(t_1\) and \(t_2\) coincide on all non-\(\bot\) attributes of \(t_1\). The removal of subsumed tuples from \(R\)—denoted \(R \downarrow\)—returns only the tuples in \(R\) that are not subsumed by another tuple. The LOJ expression \(R \bowtie_p S\) can then be characterized in terms of inner join, outer union, and removal of subsumed tuples, which can be combined into the minimal union operator \(\oplus\) [37].

\[
R \bowtie_p S \equiv ((R \bowtie_p S) \cup R) \downarrow \equiv (R \bowtie_p S) \oplus R
\] (4.2)

An advantage of equation 4.2 is that all of the non-monotonic behaviour is encapsulated within the \(\downarrow\) or \(\oplus\) operators, which can then be commuted above the inner join operators (although not above projection). This allows a tree of inner and outer joins to be abstracted as a tree of conjunctive and disjunctive operations, which can then be transformed into a canonical form; this is the conceptual basis for Galindo-Legaria’s join-disjunctive normal form (JDNF) [37], which we discuss further in Section 4.6.1.

In order to combine LOJ with the other conjunctive algebra operators, we must extend the definitions of the selection and duplicate-eliminating projection operators to handle input relations that contain nulls (introduced by lower LOJ operators). Following the three-valued logic of the SQL standard [63], we define the selection predicate \(A = B\) to evaluate to false if either attribute \(A\) or \(B\) is bound to the null constant; in other words, equality predicates are null-rejecting. Also following the SQL standard, we extend the definition of duplicate-eliminating projection to treat the null constant the same as any other database constant; in other words, identical tuples are always considered duplicate copies of the same tuple. Due to the interaction between LOJ and projection operators that introduce constants, we also need to slightly modify the syntax of projection attribute lists so

\(^1\)We intentionally use the symbol \(\bot\) rather than the SQL constant NULL to highlight our assumption that this special null constant does not occur within the input to an SPCL query, and therefore all occurrences of \(\bot\) within the query output are introduced by LOJ operators.
that constants are assigned fresh attribute names; for this we adopt the common “c AS $A_j$” syntax.

The SPCL algebra is capable of expressing any hierarchical CQ. The following example illustrates the basic technique behind the formal proof.

**Example 32** Consider the SPCL query $Q_{33}$ shown in Figure 4.1.\(^2\) Query $Q_{33}$ is equivalent to the hierarchical CQ $Q_{17}$ from Figure 2.11. For example, $E_1$ corresponds to the tree $T_6$ in Figure 2.12; similarly, $E_2$ corresponds to $T_1$, $E_3$ to $T_5$, and $E_4$ to $T_2$. The reader can verify that when $Q_{33}$ is evaluated over database instance $D_2$ from Example 20, the result—relation $R_{14}$ in Figure 4.2—is obtained from the relations shown in Figure 2.13 by projecting the appropriate attributes from the result of expression $(\Gamma_6 \Join \Gamma_4) \Join \Gamma_5 \Join \Gamma^2$.

**Theorem 4.1.1** Every HCQ can be expressed as an equivalent SPCL expression.

**Proof.** Given any HCQ $Q$, create an equivalent SPCL expression as follows.

1. For each node $v$ in $\text{body}_Q$, create an SPC expression $E_v$ from the CQ bodies of $v$ and all of $v$’s ancestors, with “fresh” copies of each attribute name. If the head of $Q$ contains a constant that is labelled with node $v$, then add a projection operator to the top of $E_v$ to introduce an appropriate constant attribute.

2. Arrange these conjunctive queries into a left-deep tree of LOJ operators whose left-to-right ordering corresponds to a breadth-first traversal of the

\(^2\)For convenience we have used a hybrid notation rather than pure SPCL; the subexpressions $E_1$, $E_2$, $E_3$, and $E_4$ represent CQ bodies that are easily translated into pure SPC expressions.
3. Create a top-most projection operator, and for each component $W_i$ labelled with $v$ in the head $Q(W)$, output the associated attribute from $E[v]$.

Consider any tree $T_i \in \text{trees}(\text{body}_Q)$, and let $u_1, \ldots, u_k$ denote the nodes of $T_i$ in breadth-first order. Given any database instance $D$, there is a one-to-one correspondence between embeddings of $T_i$ into $D$ and tuples in the inner join $(E[u_1] \Join \cdots \Join E[u_k])^D$. An embedding $\gamma : T_i \rightarrow D$ is terminal precisely when the corresponding tuple in $(E[u_1] \Join \cdots \Join E[u_k])^D$ is not subsumed by any other tuples in the result of the join tree (prior to the application of the top-most projection operator).

4.1.2 Directed Acyclic Conjunctive Queries

The following example demonstrates that SPCL is strictly more expressive than the language of hierarchical CQs.

Example 33 Let $Q_{34}$ be the SPCL query formed from $Q_{33}$ in Figure 4.1 by adding the predicate $C_2 = E_3$ to the join predicate of the LOJ operator immediately above $E_4$. Evaluating $Q_{34}$ over database instance $D_2$ from Example 20 yields the relation $R_{15}$ shown in Figure 4.2. There is no HCQ equivalent to $Q_{34}$.

The reason that SPCL query $Q_{34}$ cannot be expressed as an HCQ is that the variables $C_2$ and $E_3$ are inherited from separate blocks that are not themselves in an ancestor-descendant relationship. Syntactic well-formed-ness of HCQs requires that node $u$ must be an ancestor of $v$ if $P_v \cap L_u \neq \emptyset$; however, satisfying this requirement for this query requires a body that is not a tree. We now generalize the language of HCQs to encompass this case.
Definition 4.1.2 (Directed Acyclic Conjunctive Query (DACQ)) A directed acyclic conjunctive query is a generalization of an HCQ whose body is a rooted, connected DAG.

Let $\text{dags}(G)$ denote the set of all non-empty DAGs that can be formed from some non-empty DAG $G$ by zero or more iterations of selecting and deleting a leaf node. The evaluation of DACQs is defined analogously to the evaluation of HCQs (see Section 2.3.1), in terms of embeddings $\gamma : G \rightarrow D$ and terminal embeddings $\gamma : G \rightarrow D$ for $G \in \text{dags}(\text{body}_Q)$. The CQ $\text{body}_Q^+$ is defined analogously to $\text{body}_T^+$.

Example 34 Figure 4.3 depicts DACQ $Q_{35}$, and Figure 4.4 shows the terminal embeddings of $Q_{35}$ into database instance $D_2$.

We now show that every DACQ can be expressed as an SPCL expression and vice-versa. The basic technique behind the formal proof is illustrated by the following example.

Example 35 Consider the SPCL query $Q_{34}$ from Example 33. The DACQ $Q_{36}$ shown in Figure 4.5 is equivalent to $Q_{34}$. Observe the close correspondence between the vertices of $\text{body}_{Q_{36}}$ and the SPC blocks within $Q_{34}$.

Theorem 4.1.3 DACQs and SPCL expressions have the same expressive power.
Proof. The construction in the proof of Theorem 4.1.1 applies identically to DACQs, which proves that SPCL expressions are at least as expressive as DACQs.

To prove the other direction, it suffices to restrict our attention to a particular subclass of SPCL expressions. It follows from the work of previous authors on algebraic transformations for outer-join queries [12, 41] that given an arbitrary SPCL query, there exists an equivalent SPCL query that is rooted by a projection operator whose input is a left-deep tree of LOJ operators with SPC expressions occurring at its leaves. Let \( Q \) be any SPCL query with this restricted form, and let \( E_{u_1}, \ldots, E_{u_k} \) denote the SPC subqueries at the leaves of the left-deep tree, as they appear in left-to-right order. An equivalent DACQ can be constructed as follows.

1. Create a root node labelled \( u_1 \) containing all of the base relations and join/selection predicates occurring within SPC subquery \( E_{u_1} \).

2. Enumerate SPC queries \( E_{u_2}, \ldots, E_{u_k} \). For each query \( E_{u_i} \), let \( p_i \) denote the join predicate of the LOJ operator that has \( E_{u_i} \) as its right input.

   (a) Create a node labelled \( u_i \) containing all of the base relations and join/selection predicates occurring within SPC subquery \( E_{u_i} \), as well as the predicates in \( p_i \).

   (b) For each \( j \neq i \) such that \( p_i \) references an attribute originating from \( E_{u_j} \), add an edge from node \( u_j \) to node \( u_i \). Note that well-formed-ness of SPCL expressions guarantees \( j < i \), and so node \( u_j \) already exists.

3. Create the head of the DACQ from the projection list of the top projection operator of \( Q \). For each attribute, label it with the node label corresponding to the SPC expression from which it originates.

4. Use variable/constant substitution to eliminate any equality predicate that refers to a variable that is local to the node containing the predicate. Stop when the equality predicates in each node reference only constants and parameter variables.

Figure 4.5: DACQ \( Q_{36} \) equivalent to SPCL query \( Q_{34} \)
Corollary 4.1.4 The query equivalence problems for DACQs and SPCL expressions are polynomial-time reducible to each other.

4.2 A Canonical Form for DACQs

In order to standardize the input to the query equivalence problem which we consider in the next section, in this section we define a canonical form for DACQs. The canonical form is defined in terms of several declarative properties. In Section 4.2.1 we introduce a new type of dependency that we call a nullability dependency. In Section 4.2.2 we use nullability dependencies to characterize whether a query adheres to the properties of the canonical form, and we give an algorithm for converting an arbitrary DACQ into an equivalent canonical form.

Definition 4.2.1 (DACQ Canonical Form) Given any satisfiable DACQ $Q$, we say that $Q$ is in canonical form if it satisfies all of the following properties.

- **[P1]** For each vertex $u \in \text{body}_Q$, $\text{CQ body}_{\text{body}_u^+}$ is satisfiable.
- **[P2]** For each non-root leaf vertex $u \in \text{body}_Q$, the head tuple $\bar{W}$ of $Q$ contains a component $W_i$ that is labelled with vertex $u$.
- **[P3]** For each vertex $u \in \text{body}_Q$ and each parameter variable $X \in \mathcal{P}_u$:
  
  (a) if $\text{pred}_{\text{anc}}^+ u$ implies that $X$ is equal to a constant, then $X$ does not occur within $\text{atoms}_{\text{body}_u}$; and
  
  (b) if $\text{pred}_{\text{anc}}^+ u$ implies that $X$ is equal to some other parameter $Y \in \mathcal{P}_u$, then $\text{atoms}_{\text{body}_u}$ does not contain both $X$ and $Y$ (it may contain one).
  
  (This presumes that predicates in $\text{pred}_{\text{body}}^u$ do not reference variables from $\mathcal{L}_u$, which is the (non-restrictive) syntactic convention we previously adopted for HCQs—see Section 2.3.1.)

- **[P4]** The body of $Q$ does not contain any redundant atoms, predicates, or edges. An atom within $\text{atoms}_{\text{body}_u}$ is redundant if $\text{pred}_{\text{can}}^u u$ implies that it is identical to an atom within $\text{atoms}_{\text{body}_u}^{\text{can}}$. A predicate within $\text{pred}_{\text{body}}^u$ is redundant if it is implied by $\text{pred}_{\text{can}}^u u$. An edge $(u, v)$ is redundant if there remains a path from $u$ to $v$ after deletion of edge $(u, v)$.

- **[P5]** For each pair of distinct vertices $u, v \in \text{body}_Q$, if $v$ is not an ancestor of $u$, then there exists some terminal embedding of $Q$ into some database $D$ that embeds $\text{body}_u$ into $D$ but not $\text{body}_v$ into $D$.

The purpose of properties [P1] and [P2] is to avoid having vertices in $\text{body}_Q$ that do not affect the query output. The purpose of properties [P3] and [P4] is to standardize the syntax of the CQ bodies within the vertices by removing redundancy and minimizing the number of parameter variables. The purpose of [P5] is to standardize the shape of the DAG formed by the vertices of $\text{body}_Q$—if property [P5] is violated, then an edge can be added from vertex $v$ to vertex $u$ without affecting the output of $Q$ over any database instance.
4.2.1 Functionally-Determined Nullability

In this section we define nullability dependencies, which identify the variables in a DACQ body that determine whether an embedding extends to some node $v$.

**Definition 4.2.2 (Nullability Dependency (ND))** Given a DACQ $Q$, a non-root node $v \in \text{body}_Q$, and a set of variables $S \subseteq \mathcal{B}^{\text{anc}}_v$, we say that $S$ null-determines $v$ within the embeddings of $Q$—denoted $Q \models S \models v$—if for every database instance $D$ and every pair of embeddings $\gamma, \phi : \text{body}^{\text{anc}}_v \rightarrow D$ that coincide over $S$, embedding $\gamma$ extends to an embedding of $v$ iff embedding $\phi$ extends to an embedding of $v$. (Because all embeddings can be assumed to be extended with identity over constants in $\mathcal{C}$, it is a trivial extension to allow $S$ to contain constants.)

**Example 37** Continuing Example 36, terminal embeddings $\gamma_1$ and $\gamma_2$ in Figure 4.6c prove that $Q_{37} \not\models \{A, B\} \models v$. However, $Q_{37} \models \{E\} \models v$ holds (by the next lemma).

**Lemma 4.2.3** For any DACQ $Q$ and non-root node $v \in \text{body}_Q$ containing parameter variables $\mathcal{P}_v$, $Q \models \mathcal{P}_v \models v$. 

98
For any embedding $\gamma : \text{body}^\text{anc}_v \rightarrow D$, whether or not $\gamma$ extends to $v$ is determined only by the tuple of parameter values $\gamma(\overline{P}_v)$. \hfill \Box

We now present a homomorphic condition that characterizes the existence of a nullability dependency. For simplicity, we start by considering the very restricted case where the query body contains only two nodes, does not contain any equality predicates, and obeys properties [P1]–[P4] of Definition 4.2.1.

**Theorem 4.2.4** Given

- $\text{DACQ} Q$ obeying properties [P1]–[P4] and composed of a single edge from vertex $u$ to vertex $v$ in which $\text{pred}_u = \text{pred}_v = \emptyset$, and
- a set $S \subseteq \mathcal{B}_u$;

$Q \models S \models v$ if there exists a mapping $h : \mathcal{B}^+_v \rightarrow \mathcal{B}^+_v \cup \mathcal{C}$, extended with identity over constants $\mathcal{C}$, such that

1. $h$ is the identity mapping over $\text{atoms}_u$, and
2. for any atom $R(X) \in \text{atoms}_v$, if $h(R(X)) \in \text{atoms}_v$ then tuple $h(X)$ does not contain any variables from $\mathcal{B}_u \setminus S$.

**Proof.** Suppose that $h$ exists. Choose any database instance $D$, and any two embeddings $\gamma, \phi : \text{body}_u \rightarrow D$ that coincide over $S$ and for which there exists an embedding $\phi' : \text{body}^+_v \rightarrow D$ that extends $\phi$. Define the variable mapping $\gamma' : \mathcal{B}^+_v \rightarrow D$ as follows.

$$
\gamma'(X) := \begin{cases} 
\gamma \circ h(X) & \text{if } h(X) \in \mathcal{B}_u \\
\phi' \circ h(X) & \text{otherwise}
\end{cases}
$$

Because $h$ is the identity mapping over $\text{atoms}_u$, $\gamma'$ is consistent with $\gamma$ over $\text{body}_u$. For each atom $R(X) \in \text{atoms}_v$, if $h(R(X)) \in \text{atoms}_v$ then $\gamma'(R(X)) = \gamma \circ h(R(X)) \in D$. If $h(R(X)) \in \text{atoms}_v$, then because $h(X)$ does not contain any variables from $\mathcal{B}_u \setminus S$ and because $\gamma, \phi$ are consistent over $S$, $\gamma'(R(X)) = \phi'(R(X)) \in D$. Therefore $\gamma'$ is an embedding of $\text{body}^+_v$ into $D$ that extends $\gamma$, which implies $Q \models S \models v$.

Now suppose $Q \models S \models v$. Let $f_1$ be a function that isomorphically freezes $\mathcal{B}_u$ to constants in $\mathcal{C} \setminus \mathcal{C}_Q$, extended with identity over $\mathcal{C}_Q$. Let $f_2$ be a function that isomorphically freezes $\mathcal{B}^+_v$ to constants in $\mathcal{C} \setminus \mathcal{C}_Q$, extended with identity over $\mathcal{C}_Q$. Choose $f_1, f_2$ so that they are consistent over $S$ and satisfy $f_1(\mathcal{B}_u \setminus S) \cap f_2(\mathcal{B}^+_v \setminus S) = \emptyset$. Define database instances $D_1 := f_1(\text{atoms}_u)$, $D_2 := f_2(\text{atoms}^+_v)$, and $D := D_1 \cup D_2$. Because $f_1^{-1}$ and $f_2^{-1}$ are consistent over their common domain, we can define a single inverse mapping $f^{-1} = f_1^{-1} \cup f_2^{-1}$ that satisfies $f^{-1}(D) = \text{atoms}^+_v$.

Choose embedding $\gamma : \text{body}_u \rightarrow D$ as $\gamma := f_1$. Choose embedding $\phi : \text{body}^+_v \rightarrow D$ as $\phi := f_2$. Because $Q \models S \models v$ and $\gamma, \phi$ are consistent over $S$, there must exist some embedding $\gamma' : \text{body}^+_v \rightarrow D$ that extends $\gamma$.

Define $h_1 := f^{-1} \circ \gamma'$, and for each $i > 1$ define $h_{i+1} := h_1 \circ h_i$. Observe that each $h_i$ is a homomorphism from $\text{body}^+_v$ to $\text{body}^+_v$ that is the identity over $\mathcal{B}_u$ (and therefore over $\text{body}_u$). For $h_1$ and each variable $X \in \mathcal{L}_v$, either
1. \( \gamma'(X) \in \mathcal{C}_Q \) and so \( h_1(X) \in \mathcal{C}_Q \);
2. \( \gamma'(X) \in f_1(S) = f_2(S) \) and so \( h_1(X) \in S \);
3. \( \gamma'(X) \in f_1(B_u \setminus S) \) and so \( h_1(X) \in B_u \setminus S \); or
4. \( \gamma'(X) \in f_2(B_v^+ \setminus S) \) and so \( h_1(X) \in B_v^+ \setminus S \).

Consider the infinite sequence \( h_1(X), h_2(X), \ldots \); after \( k = |\mathcal{L}_v| \) steps, either
1. \( \exists i \in [1,k] \) such that \( h_i(X) \in \mathcal{C}_Q \cup B_u \), which implies that \( \forall j \geq i, [h_j(X) = h_i(X)] \), and so \( \gamma' \circ h_k(X) \in f_1(\mathcal{C}_Q \cup B_u) \); or
2. \( \forall i \in [1,k], h_i(X) \in \mathcal{L}_v \), which implies that the mapping composition is periodic with period at most \( k \), and so \( \forall j, [h_j(X) \in \mathcal{L}_v] \). In this case, \( \gamma' \circ h_k(X) \in f_2(\mathcal{L}_v) \).

Consider any atom \( R(^X) \in \text{atoms}_u \) such that \( h_{k+1}(R(^X)) \in \text{atoms}_u \). By property [P4], \( h_{k+1}(R(^X)) \notin \text{atoms}_u \) and therefore \( \gamma' \circ h_k(R(^X)) \notin D_1 \). Suppose that there exists some variable \( Y \in X \) such that \( h_{k+1}(Y) \in B_u \setminus S \); then, \( h_{k+1}(Y) \) must belong to the first case above, implying that \( h_k(Y) = h_{k+1}(Y) \) and \( \gamma' \circ h_k(Y) = f_1(h_k(Y)) \notin \text{adom}(D_2) \), which is a contradiction. Therefore, variable \( Y \) cannot exist, and so tuple \( h_{k+1}(X) \) does not contain variables from \( B_u \setminus S \).

**Corollary 4.2.5** There exists a unique minimal set \( \mathcal{S}_v \subseteq B_u \) satisfying \( Q \models \mathcal{S}_v \models v \), and \( \mathcal{S}_v \) can be computed in NP time.

**Proof.** It suffices to show that for any pair of sets \( \mathcal{S}_1, \mathcal{S}_2 \subseteq B_u \), if \( Q \models \mathcal{S}_1 \models v \) and \( Q \models \mathcal{S}_2 \models v \) then \( Q \models (\mathcal{S}_1 \cap \mathcal{S}_2) \models v \). If this is true, then

\[
\mathcal{S}_v = \mathcal{P}_v \setminus \{X \in \mathcal{P}_v \mid Q \models (\mathcal{P}_v \setminus \{X\}) \models v\}
\]

and so computing \( \mathcal{S}_v \) requires testing \( |\mathcal{P}_v| \) nullability-dependencies (each of which can be tested in NP time à la Theorem 4.2.4).

Suppose that \( Q \models \mathcal{S}_1 \models v \), and let \( h_1 \) be the corresponding homomorphism that satisfies Theorem 4.2.4. Suppose that \( Q \models \mathcal{S}_2 \models v \), with \( h_2 \) the corresponding homomorphism. Consider the composite function \( h_2 \circ h_1 \). For any atom \( R(^X) \in \text{atoms}_v \), if \( h_1(R(^X)) \in \text{atoms}_u \) then \( h_2 \circ h_1(R(^X)) \in \text{atoms}_u \); while if \( h_1(R(^X)) \in \text{atoms}_v \) and \( h_1(^X) \) contains a variable from \( B_u \setminus S \), then \( h_2 \circ h_1(R(^X)) \in \text{atoms}_u \) as well (because \( h_2 \) is the identity over \( B_u \) and any atom that \( h_2 \) maps into \( \text{atoms}_u \) can not contain \( B_u \setminus S \)). The symmetric argument can then be applied to the role of \( h_2 \) within the composite function \( h_1 \circ h_2 \circ h_1 \), and then to the second \( h_1 \) within \( h_2 \circ h_1 \circ h_2 \circ h_1 \), etc. Let \( h \) be a mapping formed by composing at least \( |\text{atoms}_v| \) alternations of \( h_1 \) and \( h_2 \); \( h \) is the identity over \( \text{atoms}_u \), and for any atom \( R(^X) \in \text{atoms}_v \) that \( h \) maps into \( \text{atoms}_u \), tuple \( h(^X) \) does not contain any variable from \( B_u \setminus (\mathcal{S}_1 \cap \mathcal{S}_2) \). Therefore, \( h \) proves \( Q \models (\mathcal{S}_1 \cap \mathcal{S}_2) \models v \). \( \square \)

**Example 38** Consider the following two DACQs.

\[
Q_{38}(A[@]) \iff \downarrow R(A, B), R(X, Y) \quad \iff \uparrow R(A, A), R(X, Z)
\]

\[
Q_{39}(A[@]) \iff \downarrow R(A, B), R(X, Y) \quad \iff \uparrow R(A, Z), R(X, Z)
\]
The homomorphism $h_b = \{ A/A, B/B, X/X, Y/Y, Z/Y \}$ proves $Q_{38} \models \{ A \} \Rightarrow b$, and $S_b = \{ A \}$ is minimal because no homomorphism from $body\B{}$ to itself can map atom $R(A, A)$ into $body\alpha\B{}$. For $Q_{39}$, the only homomorphism from $body\B{}{\delta}$ to itself that is the identity over $body\B{}{\epsilon}$ is $h_d = \{ A/A, B/B, X/X, Y/Y, Z/Z \};$ therefore, $S_d = \{ A, X \} = \mathcal{P}_d$ is the minimal set satisfying $Q_{39} \models S_d \Rightarrow d$.

We now extend Theorem 4.2.4 to the case where the query body contains equality predicates.

**Theorem 4.2.6** Given

- DACQ $Q$ obeying properties [P1]–[P4] and composed of a single edge from vertex $u$ to vertex $v$, and
- a set $S \subseteq \mathcal{B}_u$;

assume a total ordering over $\mathcal{B} \cup \mathcal{C}$ that places all constants before any variables. Let $\kappa : \mathcal{B} \rightarrow \mathcal{B} \cup \mathcal{C}$ be the mapping of each variable to the minimal element $Y \in \mathcal{B} \cup \mathcal{C}$ such that $\text{pred}_\gamma \B{} Y \equiv Y$. Then, $Q \models S \equiv v$ iff

1. for each variable $X$ in $\text{pred}_\gamma$, there exists a variable $Y \in S$ such that $\text{pred}_\gamma \B{} X \equiv Y$; and
2. $\kappa(Q) \equiv \kappa(S) \equiv v$.

**Proof.** Suppose that $Q \models S \equiv v$.

1. Suppose that $\text{pred}_\gamma$ contains some predicate $X = Z$ such that there does not exist a variable $Y \in S$ satisfying $\text{pred}_\gamma \B{} X = Y$. Let $\gamma$ be any embedding of $body\B{u}{\gamma}$ into any database $D$. Let $c$ denote some fresh constant not in $\text{adom}(D)$. Initialize variable mapping $\phi$ as identical to $\gamma$; then, for each variable $Y$ such that $\text{pred}_\gamma \B{} X = Y$, set $\phi(Y) := c$. Define database $D' := D \cup \phi(\text{atoms}_u)$. Both $\gamma$ and $\phi$ embed $body\B{u}{\gamma}$ into $D'$ and they coincide on $S$, yet $\gamma$ extends to $v$ while $\phi$ does not because $\phi(X) = c \neq \phi(Z)$ (property [P4] requires that $\text{pred}_\gamma \B{} X = Z$). Hence, $Q \not\models S \equiv v$, which is a contradiction, and so $\text{pred}_\gamma$ cannot contain predicate $X = Z$.

2. By construction, mapping $\kappa$ enacts the equality predicates within $\text{pred}_\gamma$; therefore, each embedding of $\kappa(\text{body}_u)$ is also an embedding of $\text{body}_u$, and each embedding of $\kappa(\text{body}_u)$ is also an embedding of $\text{body}_u$. Therefore, any two embeddings of $\kappa(\text{body}_u)$ proving $\kappa(Q) \equiv \kappa(S) \equiv v$ would also prove $Q \not\models S \equiv v$, which is a contradiction. Therefore, $\kappa(Q) \equiv \kappa(S) \equiv v$.

Now suppose that for each variable $X$ in $\text{pred}_\gamma$, there exists a variable $Y \in S$ satisfying $\text{pred}_\gamma \B{} X = Y$, and suppose that $\kappa(Q) \equiv \kappa(S) \equiv v$. Given any database $D$ and any two embeddings $\gamma, \phi : \text{body}_u \rightarrow D$ that coincide over $S$, embeddings $\gamma, \phi$ are also embeddings of $\kappa(\text{body}_u) \rightarrow D$ that coincide over $\kappa(S)$. Furthermore, $\gamma$ and $\phi$ coincide over $\mathcal{P}_u$, and so $\gamma$ satisfies $\text{pred}_\gamma$, iff $\phi$ satisfies $\text{pred}_\gamma$. Suppose that $\gamma'$ extends $\gamma$ to an embedding of $\text{body}_v$; then $\gamma'$ is also an embedding of $\kappa(\text{body}_v)$. By $\kappa(Q) \equiv \kappa(S) \equiv v$, there exists an embedding $\phi'$ that extends $\phi$ to $\kappa(\text{body}_v)$, and $\phi'$ is also an embedding of $\text{body}_v$. Therefore, $Q \models S \equiv v$. □
Each predicate in $\kappa(\text{pred}_Q)$ is a tautology and can be deleted. Hence, $\kappa(Q)$ does not contain equality predicates and so $\kappa(Q) \models \kappa(S) \equiv v$ can be tested via Theorem 4.2.4.

Theorem 4.2.6 can be used to test whether the embeddings of an arbitrary DACQ (satisfying properties [P1]–[P4]) satisfy a given nullability dependency. Given DACQ $Q$ (obeying properties [P1]–[P4]), vertex $v \in \text{body}_Q$, and some set $S \subseteq \mathcal{B}^\text{anc}_v$, create query $Q'$ from $Q$ by first deleting all nodes not in $\{v\} \cup \text{ancestors}(v)$ and then merging together all of the nodes in $\text{ancestors}(v)$. It follows from Definition 4.2.2 that $Q \models S \equiv v$ if and only if $Q' \models S \equiv v$. This further yields the following two corollaries of Theorem 4.2.6.

**Corollary 4.2.7** Given any DACQ $Q$, there exists a unique minimal set $\mathcal{S}_Q \subseteq \mathcal{B}$ such that for every non-root node $v \in \text{body}_Q$ and every set $S \subseteq \mathcal{B}^\text{anc}_v$, if $Q \models S \equiv v$ then $Q \models (\mathcal{S}_Q \cap S) \equiv v$. Furthermore, $\mathcal{S}_Q$ contains all variables occurring within $\text{pred}_Q$.

**Corollary 4.2.8** Set $\mathcal{S}_Q$ can be computed in NP time.

We call $\mathcal{S}_Q$ the *nullability set* of $Q$. We will not actually use set $\mathcal{S}_Q$ within this chapter; however, it plays a prominent role within the next chapter when we define a normal form for HCEQs.

### 4.2.2 Converting DACQs to Canonical Form

We now show how property [P5] can be characterized in terms of nullability dependencies. Using this test, we can then define an algorithm for converting an arbitrary DACQ to an equivalent canonical form.

**Theorem 4.2.9** Given

1. any DACQ $Q$ conforming to properties [P1]–[P4] of Definition 4.2.1, and
2. any two distinct vertices $u, v \in \text{body}_Q$ such that $v$ is not an ancestor of $u$;

create the query $Q_u$ from $Q$ by merging $u$ with all of its ancestors into the root vertex of the query body. Let $\vartheta$ be a mapping from the vertex labels of $Q$ to the vertex labels of $Q_u$. Then, the following property holds

$$\forall D. \forall G \in \text{dags}(Q). [(u \in G \land v \notin G) \implies (\{\gamma : G \sim D\} = \emptyset)] \quad (4.3)$$

if and only if

1. there exists a homomorphism $h : \text{body}_v^+ \rightarrow \text{body}_u^+$ such that $h$ is the identity over $\mathcal{B}_u^+ \cap \mathcal{B}_v^+$, and
2. for all $t \in \text{ancestors}(v) \cup \{v\}, Q_u \models \mathcal{B}_u^+ \cap \mathcal{B}_v^+ \equiv \vartheta(t)$.

**Proof.** Suppose that property 4.3 holds.
1. Property 4.3 implies that any embedding \( \gamma : \text{body}^+_u \to D \) can be extended into an embedding \( \gamma' : \text{body}^+_u \cup \text{body}^+_v \to D \) (note that \( \text{body}^+_u \cup \text{body}^+_v \) is a CQ body since \( \text{body}^+_u \) and \( \text{body}^+_v \) are each sets of atoms and predicates). This implies that the following two CQs \( Q' \) and \( Q'' \) satisfy \( Q' \subseteq Q'' \)

\[
Q'(\overline{B}^+_u) := \text{body}^+_u \\
Q''(\overline{B}^+_u) := \text{body}^+_u \cup \text{body}^+_v
\]

which, by the Homomorphism Theorem for CQ containment \([6]\), implies that there exists a homomorphism \( h' : Q'' \to Q' \) satisfying \( h'(\overline{B}^+_u) = \overline{B}^+_u \). This mapping \( h' \) therefore satisfies the criteria for mapping \( h \) given above.

2. For any \( D \), let \( \gamma, \phi \) be any two embeddings of \( Q_u \) that are consistent over \( B^+_u \cap B^+_v \). For any \( t \in \text{ancestors}(v) \cup \{v\} \), property 4.3 guarantees that both \( \gamma \) and \( \phi \) can be extended into embeddings of \( \vartheta(t) \), and so \( Q_u \models B^+_u \cap B^+_v \equiv \vartheta(t) \) follows by Definition 4.2.2.

We prove the opposite direction by contradiction. Suppose that \( h \) exists and that \( Q_u \) satisfies the nullability dependency for each \( t \in \text{ancestors}(v) \cup \{v\} \). Suppose further than property 4.3 is violated; that is, there exists some database \( D \) and \( G \in \text{dags}(Q) \) with \( u \in G \) and \( v \notin G \) that allows some terminal embedding \( \gamma : G \to D \). Choose any vertex \( t \in \text{ancestors}(v) \cup \{v\} \) that does not appear in \( G \) but whose ancestors all appear in \( G \) (there must exist a choice for \( t \), since \( v \notin G \) and the root node is both an ancestor of \( v \) and in \( G \)). Then,

1. \( \gamma \) is an embedding of \( \text{body}_{\vartheta(t)}^{\text{anc}} \) into \( D \);
2. \( \gamma \circ h \) is an embedding of \( \text{body}_{\vartheta(t)}^{\text{anc}} \) into \( D \);
3. \( \gamma \) and \( \gamma \circ h \) are consistent over \( B^+_u \cap B^+_v \) (by definition of \( h \) and \( B^+_u \cap B^+_v \subseteq B^+_t \));
4. \( \gamma \circ h \) extends to \( \vartheta(v) \) and therefore also to \( \vartheta(t) \); and
5. \( Q_u \models B^+_u \cap B^+_v \equiv \vartheta(t) \) implies that \( \gamma \) must extend to \( t \) within \( \text{body}_{Q_u} \), which further implies that \( \gamma \) extends to \( t \) within \( \text{body}_{Q} \).

The above reasoning contradicts the choice of \( \gamma \) as a terminal embedding of \( G \). Therefore, property 4.3 must hold. \( \square \)

**Corollary 4.2.10** For a given \( Q \) and pair of vertices \( u, v \), testing property 4.3 is NP-complete.

**Corollary 4.2.11** Testing whether \( Q \) satisfies property \([P5]\) is co-NP-complete.

**Proof.** To prove that \( Q \) does not satisfy \([P5]\), it suffices to guess the pair of vertices \( u, v \in \text{body}_{Q} \) that cause the violation (i.e., that satisfy property 4.3), along with the homomorphism \( h \) from Theorem 4.2.4 and, for each \( t \in \text{ancestors}(v) \cup \{v\} \) the homomorphisms needed to verify \( Q_u \models B^+_u \cap B^+_v \equiv \vartheta(t) \) (cf. Theorems 4.2.4 and 4.2.6). \( \square \)

**Corollary 4.2.12** For a given \( Q \) and pair of vertices \( u, v \), property 4.3 holds if

1. there exists a homomorphism \( h : \text{body}^+_u \to \text{body}^+_u \) such that \( h \) is the identity over \( B^+_u \cap B^+_v \), and
Algorithm 4 Converting DACQs into canonical form

\textbf{Canonical}(Q)

\begin{itemize}
\item \textbf{Input:} satisfiable DACQ \( Q(\mathcal{W}) \)
\item \textbf{Output:} an equivalent DACQ in canonical form
\end{itemize}

1. while \( \exists u \in \text{body}_Q \) such that \( \text{body}_u^+ \) is unsatisfiable do
2. delete vertex \( u \) and all its descendants from \( \text{body}_Q \)
3. substitute \( \square \) for each component of \( \mathcal{W} \) labelled with a deleted vertex
4. while \( \exists u \in \text{body}_Q \) such that \( u \) is a leaf, \( u \) is not the root, and \( u \) does not occur as a label in \( \mathcal{W} \) do
5. delete vertex \( u \) from \( \text{body}_Q \)
6. foreach \( u \in \text{body}_Q, X \in \mathcal{P}_u \) do
7. if \( \text{pred}_u^+ \) equates \( X \) to \( Y \in \mathcal{P}_u \cup \mathcal{C} \) then
8. replace any occurrence of \( X \) with \( Y \) within \( \text{atoms}_u \)
9. replace any occurrence of \( X\square \) with \( Y\square \) within \( \mathcal{W} \)
10. delete redundant atoms, predicates, and edges from \( \text{body}_Q \)
11. foreach pair \( u, v \in \text{body}_Q \) such that \( u \neq v \) and \( v \notin \text{ancestors}(u) \) do
12. if \( \text{TestProperty}4.3(u, v) = \text{true} \) then
13. while \( \text{body}_Q \) contains a cycle do
14. merge all participating vertices
15. delete redundant atoms and predicates from \( \text{body}_Q \)
16. return \( Q \)
\end{itemize}

2. for all \( t \in \text{ancestors}(v) \cup \{v\}, \mathcal{P}_t \subseteq (B_u^+ \cap B_v^+) \).

\textbf{Proof.} By Lemma 4.2.3, \( \mathcal{P}_t \subseteq (B_u^+ \cap B_v^+) \) implies that \( Q \models B_u^+ \cap B_v^+ \models t \). By construction of \( Q_u \), this further implies that \( Q_u \models B_u^+ \cap B_v^+ \models \vartheta(t) \). \( \square \)

We now consider how to transform a query into an equivalent canonical form. We start with an example.

Example 39 Reconsider query \( Q_{36} \) from Figure 4.5. It is straightforward to verify that \( Q_{36} \) satisfies properties [P1]–[P3]. To verify property [P5], it suffices to consider each of the following pairs as bindings for vertices \( (u, v) \).

\[ \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 2)\} \]

For each pair, the reader can verify that no homomorphism exists from \( \text{body}_v^+ \) to \( \text{body}_u^+ \); therefore, \( Q_{36} \) satisfies [P5]. Hence, to convert \( Q_{36} \) to canonical form we simply need to delete the redundant atoms so that it satisfies [P4]. The resulting query is variable-isomorphic to query \( Q_{35} \).

The function \textbf{Canonical}(Q) shown in Algorithm 4 transforms an arbitrary satisfiable DACQ into an equivalent query in canonical form. It relies upon invoking
the function \textsc{testProperty4.3}(u, v), which is not shown but directly implements the homomorphic conditions of Theorem 4.2.9. We now briefly argue the correctness of this algorithm.

- **Lines 1–3:** Because body\textsubscript{u} is unsatisfiable, neither \(u\) nor any of its descendants can participate in any embedding. Deleting \(u\) and its descendants from body\textsubscript{Q} does not change the set of terminal embeddings, and the partial application of any terminal embedding to any component of \(\overline{\mathcal{W}}\) labelled with \(u\) or any of its descendants always returns \(\mathbb{E}\). Hence, when the loop terminates the modified query remains equivalent and obeys property \([P1]\).

- **Lines 4–5:** Let \(G\) be any DAG in \textsc{dags(body\textsubscript{Q})} such that \(u \in G\), and let \(G' \in \textsc{dags(body\textsubscript{Q})}\) be the DAG formed by deleting \(u\) from \(G\). Let \(\gamma : G \rightarrow \mathcal{D}\) be any terminal embedding of \(G\) into some database \(\mathcal{D}\), and let \(\gamma' : G' \rightarrow \mathcal{D}\) be the restriction of \(\gamma\) to \(G'\). Deleting \(u\) from body\textsubscript{Q} deletes terminal embedding \(\gamma\) but causes \(\gamma'\) to become a terminal embedding; furthermore, the partial applications of \(\gamma\) and \(\gamma'\) to \(\overline{\mathcal{W}}\) are identical. Hence, when the loop terminates the modified query remains equivalent and obeys properties \([P1]\) and \([P2]\).

- **Lines 6–9:** For any \(G \in \textsc{dags(body\textsubscript{Q})}\) containing vertex \(u\), any embedding \(\gamma\) of \(G\) already satisfies \(\gamma(X) = \gamma Y\). Hence, when the loop terminates the modified query remains equivalent and obeys properties \([P1]\)–\([P3]\).

- **Line 10:** For any vertex \(u\) and any \(G \in \textsc{dags(body\textsubscript{Q})}\) containing \(u\), \(G\) necessarily contains all of \(u\)'s ancestors and so deleting redundant atoms and predicate from \(u\) does not change the embeddings for \(G\) nor the set of terminal embeddings. Deleting a redundant edge does not change the set \textsc{dags(body\textsubscript{Q})}, and so does not affect the set of terminal embeddings. Hence, after this step the modified query remains equivalent and obeys properties \([P1]\)–\([P4]\).

- **Lines 11–16:** If \textsc{testProperty4.3}(u, v) evaluates to \texttt{true}, then any terminal embedding that includes vertex \(u\) must also include vertex \(v\). Therefore, adding edge \((v, u)\) does not change the set of terminal embeddings, although it can both introduce cycles into body\textsubscript{Q} and cause some atoms or predicates to become redundant. Any terminal embedding must include either all or none of the vertices in a cycle, and so merging cycles does not affect the set of terminal embeddings. Hence, after all cycles have been merged and redundant atoms and predicates deleted, the modified query is again a proper DACQ that remains equivalent and obeys properties \([P1]\)–\([P4]\). Once this process has been repeated for all pairs of vertices in body\textsubscript{Q}, the query also satisfies property \([P5]\).

**Proposition 4.2.13** Given any satisfiable DACQ \(Q\), \textsc{Canonical}(\(Q\)) is a DACQ in canonical form satisfying \(Q \equiv \textsc{Canonical}(Q)\).

**Example 40** Consider query \(Q_{40}\) shown in Figure 4.7. The equivalent canonical form \(Q_{41} = \textsc{Canonical}(Q_{40})\)—shown in Figure 4.8—is calculated as follows.

1. body\textsubscript{5} is unsatisfiable, so delete nodes \(\Box\) and \(\Box\).

\[
\text{body}^+_5 := R(A, B, C), R(A, D, 2), S(G, H), A = 2, A = 3
\]
\[ Q_{40}(B, C, D, A, B, F, H, J, L, M) :\]
\[ \bigcup \quad R(A, B, C) \]
\[ \bigcup \quad R(A, D, 2), A = 3 \]
\[ \bigcup \quad R(A, 2, 2), A = 3 \]
\[ \bigcup \quad R(E, B, F) \]
\[ \bigcup \quad S(G, H), A = 2 \]
\[ \bigcup \quad S(A, I), S(I, J) \]
\[ \bigcup \quad S(E, K) \]
\[ \bigcup \quad R(L, L, L) \]
\[ \bigcup \quad S(J, J) \]
\[ \bigcup \quad S(K, M) \]

Figure 4.7: HCQ \( Q_{40} \) prior to conversion to canonical form

\[ Q_{41}(B, C, D, A, B, F, J, M) :\]
\[ \bigcup \quad R(A, B, C), R(E, B, F) \]
\[ \bigcup \quad R(3, D, 2), A = 3 \]
\[ \bigcup \quad R(3, 2, 2) \]
\[ \bigcup \quad S(3, I), S(I, J) \]
\[ \bigcup \quad S(E, K) \]
\[ \bigcup \quad S(K, M) \]
\[ \bigcup \quad S(K, M) \]

Figure 4.8: Query \( Q_{41} = \text{Canonical}(Q_{40}) \)
2. Delete non-root leaf node \[ \Box \] because label \[ \Box \] does not occur in the query head.
3. Replace variable \( A \) with the constant 3 within the atoms of nodes \( \Box, \Box, \) and \( \Box \), and replace \( A\Box \) with \( 3\Box \) within the query head.
4. Add an edge from \( 2 \) to \( 3 \) because \text{testProperty4.3}(\Box, \Box) \) returns \text{true}, justified by Corollary 4.2.12 as follows. Sets \( \mathcal{B}_3^+ = \{ A, B, C \} \) and \( \mathcal{B}_2^+ = \{ A, B, C, D \} \), and so homomorphism \( h : \text{body}_2 \to \text{body}_3^+ \) is the identity over \( \mathcal{B}_3^+ \cap \mathcal{B}_2^+ \).

\[
\begin{align*}
\text{body}_3^+ &= R(A, B, C), R(A, 2, 2), A = 3 \\
\text{body}_2^+ &= R(A, B, C), R(A, D, 2), A = 3 \\
h &= \{ A/A, B/B, C/C, D/D \}
\end{align*}
\]
Next, \( \text{ancestors}(\Box) \cup \{ \Box \} = \{ \Box, \Box \} \), and so \( \mathcal{P}_1 = \emptyset \) and \( \mathcal{P}_2 = \{ A \} \) are both subsets of \( \mathcal{B}_3^+ \cap \mathcal{B}_2^+ \). Adding this edge makes the predicate \( A = 3 \) within node \( \Box \) redundant, and makes the edge \( 1 \) redundant, so delete them.

5. Add an edge from \( 4 \) to \( 1 \) because \text{testProperty4.3}(\Box, \Box) \) returns \text{true}, justified by Corollary 4.2.12 as follows. Sets \( \mathcal{B}_1^+ = \{ A, B, C \} \) and \( \mathcal{B}_4^+ = \{ A, B, C, E, F \} \), and so homomorphism \( h : \text{body}_4^+ \to \text{body}_1^+ \) is the identity over \( \mathcal{B}_1^+ \cap \mathcal{B}_4^+ \).

\[
\begin{align*}
\text{body}_1^+ &= R(A, B, C) \\
\text{body}_4^+ &= R(A, B, C), R(E, B, F) \\
h &= \{ A/A, B/B, C/C, E/A, F/C \}
\end{align*}
\]
Next, \( \text{ancestors}(\Box) \cup \{ \Box \} = \{ \Box, \Box \} \), and so \( \mathcal{P}_1 = \emptyset \) and \( \mathcal{P}_4 = \{ B \} \) are both subsets of \( \mathcal{B}_1^+ \cap \mathcal{B}_4^+ \). Adding this edge causes a cycle, so merge nodes \( \Box \) and \( \Box \) into a single node \( \Box \).

### 4.3 Equivalence of Constant-Headed Queries

Our goal in this chapter is to characterize the query equivalence problem for HCQs and, more generally, for DACQs. In this section we study the equivalence problem for a subclass of DACQs that we call constant-headed, which generalize boolean CQs. We say that a DACQ is constant-headed if its head tuple does not contain any variables, and we use CH-DACQ to denote the class of constant-headed DACQs.

We describe a method for testing CH-DACQ equivalence via reduction to a set of logical implication problems. Unfortunately, although this algorithm is sound, it is not complete for concluding equivalence because it is possible that some of the generated implication problems are only semi-decidable. As a result, our results in this section are inconclusive in that we have neither proved nor disproved whether CH-DACQ equivalence is decidable.

#### 4.3.1 Head Graphs and Canonical Tuples

We will address CH-DACQ equivalence by considering separate equivalence problems for each canonical tuple. Defining a query’s canonical tuples requires understanding a query’s tuple shapes, which are characterized by the query head graph.
Figure 4.9: A constant-headed DACQ with its head graph and canonical tuples

(Although our focus here is on CH-DACQ equivalence, we will make certain definitions and theorems general to arbitrary DACQs so that we can reuse them in Section 4.4 without modification.)

**Definition 4.3.1 (Head Graph)** Given DACQ $Q$ with head label set $L$, the head graph of $Q$ is a rooted DAG $G_{\text{head}}^Q = (U, E)$ defined as follows.

\[
U := L \cup \{ u \mid u \text{ is the root node of } \text{body}_Q \} \\
E := \{ (v_i, v_k) \mid \text{body}_Q \text{ contains path } v_i \rightarrow v_k, \text{ and} \exists v_j \in L. [ \text{body}_Q \text{ contains path } v_i \rightarrow v_j \rightarrow v_k] \}
\]

**Definition 4.3.2 (Tuple Shapes)** Given DACQ $Q$ with head label set $L$, we define a tuple shape $\mathfrak{t}_s$ of $Q$ to be a subset of $L$ corresponding to the vertices of some DAG within $\text{dags}(G_{\text{head}}^Q)$. We use $\mathcal{T}S_Q$ to denote the set of all tuple shapes of $Q$.

\[
\mathcal{T}S_Q := \{ \mathfrak{t}_s \mid \exists (U, E) \in \text{dags}(G_{\text{head}}^Q) \land \mathfrak{t}_s = L \cap U \}
\]

We call shape $\mathfrak{t}_s = L$ the terminal shape for $Q$.

**Example 41** Figures 4.9a and 4.9b show constant-headed query $Q_{42}$ and its head graph $G_{Q_{42}}^\text{head}$, respectively. The tuple shapes of $Q_{42}$ are as follows,

\[
\mathcal{T}S_{42} = \{ \mathfrak{t}_2, \mathfrak{t}_2, \mathfrak{t}_4, \mathfrak{t}_5, \mathfrak{t}_{24}, \mathfrak{t}_{25}, \mathfrak{t}_{46}, \mathfrak{t}_{245}, \mathfrak{t}_{246}, \mathfrak{t}_{2456} \}
\]

where the subscripts denote the set of labels in each shape. Observe that the relation in Figure 4.9c (which we will describe further in Example 43) contains one tuple corresponding to each tuple shape in $\mathcal{T}S_{42}$ except for the terminal shape $\mathfrak{t}_{4256}$.

Next we show that one necessary condition for equivalence between queries in canonical form is that they have isomorphic head graphs.
Definition 4.3.3 (Head Graph Isomorphism) Given DACQs $Q$ and $Q'$ with head graphs $G_Q^\text{head} = (U, E)$ and $G_Q'^\text{head} = (U', E')$, a head graph isomorphism $\theta$ is a bijective mapping $\theta : U \rightarrow U'$ satisfying the two properties

1. $\theta(L) = L'$, and
2. $\theta(G_Q^\text{head}) = G_Q'^\text{head}$.

We say that $Q$ and $Q'$ are head graph isomorphic if $\theta$ exists.

The existence of a head graph isomorphism can be tested in linear time because the property $\theta(L) = L'$ dictates the precise bijection between the vertices of the head graphs. Observe that if $\theta$ is a head graph isomorphism between $Q$ and $Q'$, then $\theta(TS_Q) = TS_{Q'}$ follows from Definitions 4.3.2 and 4.3.3.

Theorem 4.3.4 Let $Q$ and $Q'$ be DACQs in canonical form. If $Q \equiv Q'$ then $Q$ and $Q'$ are head graph isomorphic.

Proof (sketch). We use properties [P1] and [P5] of Definition 4.2.1 and an analysis of cases under which $\theta$ does not exist to establish that if $\theta$ does not exist, then there exists a tuple that only one of the queries is able to output. \qed

The shapes in $TS_Q$ can be used to partition the set $\text{dags(body}_Q)$ so that associated with each shape $s \in TS_Q$ is a subset of $\text{dags(body}_Q)$ comprised of the DAGs whose terminal embeddings yield the tuples corresponding to shape $s$.

$$\text{dags(body}_Q, s) := \{ G \mid G = (U, E) \in \text{dags(body}_Q) \land U \text{ contains all of the labels in } s \land U \text{ contains none of the labels in } L \setminus s \}$$

$$\text{dags(body}_Q) = \bigcup_{s \in TS_Q} \text{dags(body}_Q, s)$$

Observe that equations 4.4 and 4.5 together imply that set $\text{dags(body}_Q, s)$ contains a unique minimal DAG—which we denote as $G_{s}^\text{min}$—such that all other DAGs in $\text{dags(body}_Q, s)$ are extensions of $G_{s}^\text{min}$.

$$G_{s}^\text{min} := G \in \text{dags(body}_Q, s). \text{ all leaves of } G \text{ are in } s$$

We use $G_{s}^\text{min}$ to define a particular CQ we call the shape-$s$ query.

Definition 4.3.5 (Shape Queries) Given DACQ $Q(\overline{W})$, for each shape $s \in TS_Q$ let $\overline{W}_s$ denote the head tuple formed by changing to any component of $\overline{W}$ whose label is not in $s$. We define the shape-$s$ query $\hat{Q}_s$ as the CQ with head tuple $\overline{W}_s$ and body $\text{body}_{G_{s}^\text{min}}^+$, and we define shape query $\hat{Q}$ as the union of all shape-$s$ queries.

$$\hat{Q}_s(\overline{W}_s) := \text{body}_{G_{s}^\text{min}}^+$$

$$\hat{Q} := \bigcup_{s \in TS_Q} \hat{Q}_s$$

109
Example 42 Continuing Example 41, consider shape $t_{s_2} \in TS_{Q_{42}}$, which corresponds to the following subset of $\text{dags}(\text{body}_{Q_{42}})$ (subscripts denote labels within each DAG).

$$\text{dags}(\text{body}_{Q_{42}}, t_{s_2}) = \{G_1, G_{13}\}$$

Shape $t_{s_2}$ has minimal dag $G^\text{min}_{t_{s_2}} = G_1$, yielding shape-$t_{s_2}$ query $\hat{Q}_{42}|_{t_{s_2}}$ as follows.

$$\hat{Q}_{42}|_{t_{s_2}} (\overline{A}, \overline{B}, \overline{C}, \overline{D}) := R(A, B), R(C, D)$$

**Theorem 4.3.6** Given DACQ $Q$ and any database $D$, the output instance of shape query $\hat{Q}$ corresponds precisely to the partial applications of all (not just terminal) embeddings of $\text{body}_Q$ into $D$.

$$\mathbb{T}((\hat{Q})^D) = \{\hat{\gamma}(\overline{W}) \mid \exists G \in \text{dags}(\text{body}_Q), \gamma : G \rightarrow D\} \quad (4.10)$$

**Proof.** By equations 4.5 and 4.6, every $G \in \text{dags}(\text{body}_Q)$ corresponds to precisely one shape $s \in TS_Q$, and so $G$ is an extension of the minimal DAG $G_{s_2}^\text{min}$. Any embedding $\gamma : G \rightarrow D$ is therefore also an embedding of $G_{s_2}^\text{min}$ into $D$, and head tuple $\overline{W}|_s$ (Definition 4.3.5) is designed so that $\hat{\gamma}(\overline{W}) = \gamma(\overline{W}|_s)$ (see equation 2.22).

**Corollary 4.3.7** $Q \equiv \hat{Q}$

**Corollary 4.3.8** Given DACQs $Q$ and $Q'$ in canonical form, if $Q \equiv Q'$ then

$$\forall s \in TS_Q. [\hat{Q}|_s \equiv \hat{Q}'|_{\theta(s)}]$$

where $\theta$ is the head graph isomorphism guaranteed by Theorem 4.3.4.

Corollary 4.3.8 is important because it provides another necessary condition for DACQ equivalence; it can be tested in NP-time per shape $s \in TS_Q$. Roughly speaking, the condition is analogous to explicitly verifying equivalence between each pair of terms in the JDNF of two join queries (see Section 4.6.1). In our context this is only a necessary condition for equivalence, whereas for join queries it is necessary and sufficient for equivalence (due to lack of a projection operator). The complicating effect of projection is that it may project away variables used in outer-join predicates (i.e., any variable used as a parameter variable in a child CQ$^\star$). We can obtain a sufficient condition for query equivalence by dictating a particular agreement between the attributes used in the outer-join predicates. To do this, first define for each shape $s \in TS_Q$ the set of parameter variables $\mathcal{P}_s \subseteq \mathcal{P}_{Q^\star}$, as follows.

$$\mathcal{P}_s := \bigcup_{v \in G_{s}^\text{min}} \mathcal{P}_v$$

**Theorem 4.3.9** Given DACQs $Q$ and $Q'$ in canonical form, if

- there exists a head graph isomorphism $\theta$ between $Q$ and $Q'$,
- there exists a bijection $\nu : \mathcal{P}_Q^\star \rightarrow \mathcal{P}_Q'^\star$ satisfying $\forall s \in TS_Q. [\nu(\mathcal{P}_s) = \mathcal{P}_{\theta(s)}]$, and
for each shape $\mathfrak{t} \in \mathcal{T}_Q$, the CQ equivalence $\hat{Q}|_{\mathcal{t}} \equiv \hat{Q}|_{\theta(\mathfrak{s})}$ can be proved by homomorphisms $h_{\mathfrak{t}} : \hat{Q}|_{\theta(\mathfrak{s})} \to \hat{Q}|_{\mathfrak{t}}$ and $h'_{\mathfrak{t}} : \hat{Q}|_{\mathfrak{t}} \to \hat{Q}|_{\theta(\mathfrak{s})}$ that are consistent with $\nu$.

then $Q \equiv Q'$.

**Proof.** Suppose that the conditions of the theorem are met. For any database $D$, and any tuple shape $\mathfrak{t} \in \mathcal{T}_Q$, let $\gamma$ be any terminal embedding of $Q$ into $D$ yielding a tuple $t$ with shape $\mathfrak{t}$. Then, $\gamma$ is also an embedding of $\hat{Q}|_{\mathfrak{t}}$ into $D$, and so $\gamma \circ h_{\mathfrak{t}}$ is an embedding of $\hat{Q}|_{\theta(\mathfrak{s})}$ into $D$, and therefore $\gamma \circ h_{\mathfrak{t}}$ is an embedding of $Q'$ into $D$ whose partial application yields $t$.

If $\gamma \circ h_{\mathfrak{t}}$ is a terminal embedding then we are done; hence, suppose that there exists some embedding $\phi : Q' \to D$ that extends $\gamma \circ h_{\mathfrak{t}}$ and generates a tuple $t'$ with shape $\mathfrak{t}'$ that subsumes shape $\mathfrak{t}$. In this case, $\phi \circ h'_{\mathfrak{t}}$ is an embedding of $Q$ into $D$ yielding $t'$. Although $\phi \circ h'_{\mathfrak{t}}$ is not necessarily an extension of $\gamma$, it is consistent with $\gamma$ over any variable in $\mathcal{P}_Q$ over which $\gamma$ is defined. It follows that $\gamma$ must have an extension that generates tuple $t'$, which contradicts the choice of $\gamma$ as a terminal embedding.

The sufficient condition in Theorem 4.3.9 can actually be tested in NP-time, because agreement of all of the homomorphisms on the parameter variables means that there must exist a single homomorphism in each direction that can prove equivalence for each pair of shape queries. We defer an example of Theorem 4.3.9 until the end of Example 47.

Having defined both head graphs and shape queries, we can now formally define the canonical tuples for a given CQ-DACQ.

**Definition 4.3.10 (Canonical Tuples)** Given CH-DACQ $Q$ in canonical form, the set of canonical tuples for $Q$—denoted $\mathcal{C}T_Q$—corresponds to the head tuples of satisfiable shape queries.

$$\mathcal{C}T_Q := \{ \mathcal{W}|_{\mathfrak{t}} \mid \mathfrak{t} \in \mathcal{T}_Q \land \hat{Q}|_{\mathfrak{t}} \text{ is satisfiable} \} \tag{4.11}$$

For a given canonical tuple $t \in \mathcal{C}T_Q$, we use $\mathfrak{t}_t^Q$ to denote the corresponding tuple shape from $\mathcal{T}_Q$.

**Lemma 4.3.11** Given CH-DACQs $Q$ and $Q'$ in canonical form, if $Q \equiv Q'$ then $\mathcal{C}T_Q = \mathcal{C}T_{Q'}$.

**Proof.** Follows from Definition 4.3.10 because $\theta(\mathcal{T}_Q) = \mathcal{T}_{Q'}$ (Theorem 4.3.4) and $\hat{Q}|_{\mathfrak{t}} \equiv \hat{Q}'|_{\theta(\mathfrak{s})}$ (Corollary 4.3.8).

111
Even though the set of canonical tuples \( \mathbb{C}_Q \) is defined in terms of satisfiable shape queries, \( \mathbb{C}_Q \) can still contain spurious tuples that \( Q \) cannot actually output. We illustrate this in the following example.

Example 43 Continuing Examples 41 and 42, Figure 4.9c shows the canonical tuples of \( Q_{42} \) (depicted as a relation). Consider the following two shape queries \( \hat{Q}_{42} \mid_{s_{25}} \) and \( \hat{Q}_{42} \mid_{s_{2456}} \), whose bodies correspond to DAGs \( G_{125} \), \( G_{12456} \in \text{dags}(\text{body}_{Q_{42}}) \).

\[
\hat{Q}_{42} \mid_{s_{25}} (0, 0, 0, 0) \leftarrow R(A, B), R(C, D), A = a, B = b, C = c, C = D \\
\hat{Q}_{42} \mid_{s_{2456}} (0, 0, 0, 0) \leftarrow R(A, B), R(C, D), A = a, B = b, C = c, C = D, D = d
\]

The shaded row in Figure 4.9c is canonical tuple \( t_{25} = \text{head}_{\hat{Q}_{42} \mid_{s_{25}}} \). Shape query \( \hat{Q}_{42} \mid_{s_{25}} \) is satisfiable, yet \( Q_{42} \) will never output \( t_{25} \) because any embedding \( \gamma \) of \( \hat{Q}_{42} \mid_{s_{25}} \) is an embedding of \( G_{125} \) that can be extended to an embedding of \( G_{1245} \), and therefore \( \gamma \) is not terminal. The tuple \( t_{2456} = \{ 0, 0, 0, 0 \} = \text{head}_{\hat{Q}_{42} \mid_{s_{2456}}} \) is not a canonical tuple—even though \( s_{2456} \in TS_{Q_{42}} \) because shape query \( \hat{Q}_{42} \mid_{s_{2456}} \) is unsatisfiable.

### 4.3.2 Tuple-Specific Equivalence

Having defined a query’s canonical tuples, we now reduce CH-DACQ equivalence to a set of tuple-specific equivalence tests, one per canonical tuple.

**Definition 4.3.13 (Tuple-Specific Equivalence)** Given any tuple \( t \) and any two DACQs \( Q \) and \( Q' \), we write \( Q \equiv Q' \) to mean \( \forall D, [t \in (Q)^D] \iff [t \in (Q')^D] \).

**Theorem 4.3.14** Given CH-DACQs \( Q \) and \( Q' \) in canonical form, \( Q \equiv Q' \) iff

1. \( \mathbb{C}_Q = \mathbb{C}_{Q'} \), and
2. \( \forall t \in \mathbb{C}_Q, Q \equiv Q' \).

**Proof.** “If” follows from Lemma 4.3.11 and Definition 4.3.13. “Only if” follows from Lemma 4.3.12 and Definition 4.3.13.

In the remainder of this section we consider how to decide the tuple-specific equivalence condition that arises from Theorem 4.3.14. Our presentation of the problem is organized by the complexity of the head graph.

**Tuple-Specific Equivalence: Single-Node Head Graphs**

Let \( Q \) denote a CH-DACQ in canonical form whose head graph contains only one node [□]. This implies \( \text{dags}(G^\text{head}_Q) = \{ G_u \} \), and so \( \mathbb{C}_Q = \{ t \} \) for some (possibly empty) tuple \( t \). By Definition 4.3.1, vertex \( u \) must be the root vertex of \( \text{body}_Q \). Because \( Q \) is in canonical form, property [P2] of Definition 4.2.1 implies that \( \text{body}_Q \) is also only a single vertex—hence, \( Q \) is simply a boolean CQ. Given query \( Q' \) with
\( \mathcal{CT}_{Q'} = \mathcal{CT}_Q \), testing tuple-specific equivalence \( Q \models Q' \) therefore corresponds to boolean CQ equivalence, which corresponds to the NP-complete problem of finding homomorphisms in each direction between \( Q \) and \( Q' \).

Before moving on to queries with more complex head graphs, we first present an alternative characterization of equivalence in terms of finite implication of dependencies. While this may seem like overkilling the problem for this case, in later sections we will extend this alternative characterization to handle more complex head graphs.

An embedded dependency \( \xi \) can be expressed as a rule of the form

\[ P^\xi \rightarrow C^\xi \]

where \( P^\xi \)—called the **premise**—and \( C^\xi \)—called the **conclusion**—are both CQ\(^\ast\) bodies \([6, 30]\). Any variable occurring in both the premise and the conclusion is called **free**, and is implicitly universally quantified; all other variables are implicitly existentially quantified. Dependency \( \xi \) is satisfied by database instance \( D \)—denoted \( D \models \xi \)—if each embedding of the premise into \( D \) can be extended to an embedding of the conclusion. Without loss of generality we can assume that \( P^\xi \subseteq C^\xi \), because any atom or predicate occurring on both sides of the dependency is trivially implied and can therefore be deleted from \( C^\xi \). Given a set of dependencies \( \Sigma \) and a database instance \( D \), we write \( D \models \Sigma \) to mean \( \forall \xi \in \Sigma, D \models \xi \). Given two dependency sets \( \Sigma_1 \) and \( \Sigma_2 \), we write \( \Sigma_1 \models_{\text{fin}} \Sigma_2 \) and \( \Sigma_1 \models_{\text{e}} \Sigma_2 \) to mean finite implication \( \forall D, (D \models \Sigma_1 \implies D \models \Sigma_2) \) and finite equivalence \( \forall D, (D \models \Sigma_1 \iff D \models \Sigma_2) \), respectively (\( \models_{\text{unr}} \) and \( \models_{\text{e unr}} \) are analogous for when the quantification over \( D \) allows infinite database instances). Deciding \( \models_{\text{fin}} \) and \( \models_{\text{e}} \) are mutually-reducible problems,

\[
\begin{align*}
\Sigma_1 \models_{\text{fin}} \Sigma_2 & \iff \Sigma_1 \models_{\text{e}} (\Sigma_1 \cup \Sigma_2) \quad (4.12) \\
\Sigma_1 \models_{\text{e}} \Sigma_2 & \iff (\Sigma_1 \models \Sigma_2) \wedge (\Sigma_2 \models \Sigma_1) \quad (4.13)
\end{align*}
\]

and so although we use finite equivalence for conceptual simplicity, it corresponds to the widely-studied problem of finite implication.

Corresponding to node \( u \) of query \( Q \) and canonical tuple \( t \), we define the following dependency \( \xi^{(u,t)} \) and (singleton) dependency set \( \Sigma^{(Q,t)} \).

\[
\begin{align*}
\xi^{(u,t)} := \text{body}^+_u \rightarrow \bot \quad (4.14) \\
\Sigma^{(Q,t)} := \{ \xi^{(u,t)} \} \quad (4.15)
\end{align*}
\]

**Lemma 4.3.15** For any database \( D \), \( t \in (Q)^D \) iff \( D \not\models \Sigma^{(Q,t)} \).

**Proof.** By equation 4.14, \( D \models \xi^{(u,t)} \) iff \( \text{body}^+_u \) does not embed into \( D \).

**Corollary 4.3.16** \( Q \models Q' \) iff \( \Sigma^{(Q,t)} \models_{\text{fin}} \Sigma^{(Q',t)} \).

Deciding finite equivalence between sets \( \Sigma^{(Q,t)} \) and \( \Sigma^{(Q',t)} \) (as defined by equation 4.15) is NP-complete due to the constrained forms of the dependencies.
Continuing Example 44, for canonical tuple $T_S T_S T_R$. For any database $z$. This implies $S_T S_R 4 T_S T_S T_R$). Because $v_S T_S$ to child $v$.

Figure 4.10 shows four queries that each have a dual-node head graph.

**Tuple-Specific Equivalence: Dual-Node Head Graphs**

Let $Q$ denote a CH-DACQ in canonical form whose head graph contains a single edge from parent $\square$ to child $\square$. This implies $\text{dags}(G^\text{head}_Q) = \{G_{uv}, G_u\}$, and so $CT_Q = \{t_1, t_2\}$ for some tuple $t_1$ whose corresponding shape $t_S Q \in TS_Q$ is terminal, and a second tuple $t_2$ that is subsumed by $t_1$ (and whose corresponding shape $t_S t_2 \in TS_Q$ does not contain $\square$). Because $Q$ is in canonical form, property [P2] of Definition 4.2.1 implies that $\text{body}_Q$ is a rooted DAG with root $v$ and the single leaf $v$, but with an arbitrary number of intermediate vertices. Given a second query $Q'$ with $CT_{Q'} = CT_Q$, we can characterize the relationships $\frac{Q}{Q}$ and $\frac{Q}{Q}$ by reducing them to deciding finite equivalence between sets of dependencies.

**Example 44** Figure 4.10 shows four queries that each have a dual-node head graph and canonical tuples $\{\{0\}, \{\square\}\}$. All four queries are in DACQ canonical form.

Testing equivalence for tuple $t_1$ is analogous to the case of single-node head graphs. Corresponding to node $v$ of query $Q$ and tuple $t_1$ we define the dependency $\xi(v, t_1)$ and the dependency set $\Sigma(Q, t_1)$ as follows.

$$\xi(v, t_1) := \text{body}^+_v \rightarrow \bot$$
$$\Sigma(Q, t_1) := \{\xi(v, t_1)\}$$

As before, deciding finite equivalence between $\Sigma(Q, t_1)$ and $\Sigma(Q', t_1)$ is NP-complete.

**Lemma 4.3.17** For any database $D$, $t_1 \in (Q)^D$ iff $D \not\equiv \Sigma(Q, t_1)$.

**Corollary 4.3.18** $Q \frac{Q}{Q'} \; \text{iff} \; \Sigma(Q, t_1) \equiv_{\text{fin}} \Sigma(Q', t_1)$.

**Example 45** Continuing Example 44, for canonical tuple $\{0\}$ all four queries in Figure 4.10 yield the same $\{0\}$-dependency set

$$\Sigma(Q_{43}, \{0\}) = \Sigma(Q_{44}, \{0\}) = \Sigma(Q_{45}, \{0\}) = \Sigma(Q_{46}, \{0\})$$
$$= \{[R(A, B), S(B, C), T(B, D), S(C, E), T(D, F)] \rightarrow 1\}$$

114
and so $Q_{43} \equiv Q_{44} \equiv Q_{45} \equiv Q_{46}$ follows trivially.

Deciding equivalence for tuple $t_2$ is more challenging, because the definition of DACQ evaluation semantics in terms of terminal embeddings makes query $Q$ non-monotonic with respect to tuple $t_2$. As a result, the dependencies we derive no longer have the constrained form of equations 4.14 and 4.16, but instead have conclusions with non-empty CQ bodies. In particular, it is the existentially-quantified variables within the conclusions that are problematic. Implication of arbitrary sets of embedded dependencies is a widely-studied problem, and in Section 4.6.4 we survey some of the literature pertaining to it. For now, the salient features are that although it is only semi-decidable in general, there are useful sub-classes of the problem which allow decidability—in particular, classes for which the well-studied chase algorithm is guaranteed to terminate [6, 30].

To decide $\Sigma \models_{\text{fin}} \xi$ using the chase we construct a new dependency $\hat{\xi}$ by isomorphically “freezing” both the free variables and the non-free premise variables of $\xi$ into fresh constants. Because premise $P^\xi$ contains only constants, it can be treated as a database instance to be chased. Let database $D := \text{chase}_\Sigma(P^\xi)$ denote the (possibly infinite) database resulting from any sequence of chase steps applying dependencies from $\Sigma$ to the initial database $P^\xi$. If $D$ is finite then $D \models \Sigma$; if $D$ is infinite then the truth of $D \models \Sigma$ is independent of the order of chase steps$^3$ [6, Ch. 10]. In either case, the truth of $D \models \xi$ is independent of the order of chase steps.

The chase procedure can return the conclusion $\Sigma \models_{\text{fin}} \xi$ in finite time by demonstrating any finite database $\hat{D} \subseteq D$ corresponding to a prefix of the chase sequence yielding $D$, such that either

1. there exists an embedding $\gamma : C^\xi \rightarrow \hat{D}$, or
2. $\hat{D}$ contains a contradiction.

This proves $\hat{D} \models \xi$ and hence $D \models \hat{\xi}$ (because premise $P^\xi$ does not contain variables) which—by the isomorphic freezing—implies $\Sigma \models_{\text{unr}} \xi$ and hence $\Sigma \models_{\text{fin}} \xi$.

The chase procedure only returns the conclusion $\Sigma \models_{\text{fin}} \xi$ when $D$ is finite and $D \not\models \xi$ (i.e., $D$ does not contain a contradiction and there does not exist an embedding of $C^\xi$ into $D$). Hence, $\Sigma \models_{\text{fin}} \xi$ can only be decided by the chase algorithm in cases where there exists a finite chase sequence for $\text{chase}_\Sigma(P^\xi)$—a condition which is itself undecidable to identify [30]. In Section 4.5 we discuss how to deal with cases where chase termination cannot be guaranteed.

Recall from equation 4.5 that $\text{dags(body}_Q, t_2^Q)$ contains all of the DAGs in $\text{dags(body}_Q)$ containing node $u$ but not node $v$. For each $G \in \text{dags(body}_Q, t_2^Q)$ we define the following embedded dependency $\xi(G, t_2)$, and we collect all of these dependencies into the dependency set $\Sigma(Q, t_2)$.

$$\xi(G, t_2) := \text{body}_G^v \rightarrow \text{body}_v^+$$  \hspace{1cm} (4.18)

$$\Sigma(Q, t_2) := \{\xi(G, t_2) \mid G \in \text{dags(body}_Q, t_2)\}$$  \hspace{1cm} (4.19)

$^3$As long as the infinite chase sequence obeys certain basic properties—primarily, that no dependency is “starved.”
Continuing Examples 44 and 45, for canonical tuple \( \{a,b\} \) the queries in Figure 4.10 yield the following \( \{a,b\}\)-dependency sets.

\[
\Sigma(Q_{44},\{a,b\}) = \begin{cases} 
(1) : [R(a,b) \rightarrow [S(b,c),T(b,d),S(c,e),T(d,f)]], \\
(2) : [R(a,b),S(b,c),T(b,d),S(c,e),T(d,f)], \\
(3) : [R(a,b),S(b,c),T(b,d),S(c,e),T(d,f)], \\
(4) : [R(a,b),S(b,c),T(b,d),S(c,e),T(d,f)]. 
\end{cases}
\]

Lemma 4.3.19 For any database \( D \), \( t_2 \in (Q)^D \) iff \( D \notin \Sigma(Q,t_2) \).

Proof. Suppose that \( t_2 \in (Q)^D \). Then, there exists some terminal embedding \( \gamma : G \rightarrow D \) of some DAG \( G \in \text{dags}(\text{body}_Q,\xi^{(G,t_2)}) \) which doesn’t contain \( v \); however, \( \gamma \) is also an embedding of \( P^{(G,t_2)} \) that does not extend to \( C^{(G,t_2)} \) and so \( D \notin \xi^{(G,t_2)} \).

Conversely, let \( G \) be a maximal DAG in \( \text{dags}(\text{body}_Q,\xi^{(G,t_2)}) \) such that \( D \notin \xi^{(G,t_2)} \). Then, there exists embedding \( \gamma : P^{(G,t_2)} \rightarrow D \) that does not extend to \( C^{(G,t_2)} \); however, \( \gamma \) is a terminal embedding of \( G \) (by choice of \( G \)) and so generates tuple \( t_2 \in (Q)^D \).

Corollary 4.3.20 \( Q \equiv Q' \) iff \( \Sigma(Q,t_2) \equiv \Sigma(Q',t_2) \).

Example 47 Continuing Examples 44–46, we now prove that \( Q_{43} \equiv Q_{44} \), but for all other pairs of queries, equivalence does not hold.

\[
Q_{43} \equiv Q_{44} : \\
\begin{align*}
\Sigma(Q_{43},\{a,b\}) \equiv_{\text{fin}} \xi^{(1,\{a,b\}))} : & \text{Trivial} \\
\Sigma(Q_{43},\{a\}) \equiv_{\text{fin}} \xi^{(12,\{a\}))} : & D := \text{chase}_{\Sigma(Q_{43},\{a\}))}([R(a,b),S(b,c)]) \\
& = \{R(a,b),S(b,c),S(b,c),T(b,d_1),S(c_1,e_1),T(d_1,f_1),S(c_2)\} \\
& \equiv [T(b,d),S(c,e),T(D,F)] \\
\Sigma(Q_{43},\{b\}) \equiv_{\text{fin}} \xi^{(13,\{b\}))} : & D := \text{chase}_{\Sigma(Q_{43},\{b\}))}([R(a,b),T(b,d)]) \\
& = \{R(a,b),T(b,d),S(b,c),T(b,d_1),S(c_1,e_1),T(d_1,f_1),T(d_2,f_2)\} \\
& \equiv [S(b,c),T(C,E),T(D,F)] \\
\Sigma(Q_{43},\{a,b\}) \equiv_{\text{fin}} \xi^{(123,\{a,b\}))} : & \text{Trivial}
\end{align*}
\]
\[ \Sigma(Q_{44}, \{\emptyset\}) \models_{\text{fin}} \zeta^{a,\{\emptyset\}}:\text{Trivial} \]
\[ \Sigma(Q_{44}, \{\emptyset\}) \models_{\text{fin}} \zeta^{abc,\{\emptyset\}}:\text{Trivial} \]
\[ \Sigma(Q_{44}, \{\emptyset\}) \models_{\text{fin}} \zeta^{abc,\{\emptyset\}}:\text{Trivial} \]
\[ D := \text{chase}_{\Sigma(Q_{44}, \{\emptyset\})}([R(a, b), S(b, c), T(b, d), S(c, e)]) \]
\[ = \{ R(a, b), S(b, c), T(b, d), S(c, e), T(d, f_1) \} \]
\[ \models [T(d, F)] \]

\[ Q_{43} \not\equiv Q_{45}: \Sigma(Q_{45}, \{\emptyset\}) \not\models_{\text{fin}} \zeta^{abc,\{\emptyset\}}:\text{Trivial} \]
\[ D := \text{chase}_{\Sigma(Q_{45}, \{\emptyset\})}([R(a, b), S(b, c), T(b, d)]) \]
\[ = \{ R(a, b), S(b, c), T(b, d), S(c, e_1), T(b, d_1), T(d_1, f_1) \} \]
\[ \not\models [S(c, E), T(d, F)] \]

\[ Q_{43} \not\equiv Q_{46}: \Sigma(Q_{46}, \{\emptyset\}) \not\models_{\text{fin}} \zeta^{lm,\{\emptyset\}}:\text{Trivial} \]
\[ D := \text{chase}_{\Sigma(Q_{46}, \{\emptyset\})}([R(a, b), S(b, c)]) \]
\[ = \{ R(a, b), S(b, c), T(b, d), T(d_1, f_1), S(b, c_1)S(c_1, e_1) \} \]
\[ \not\models [S(c, E), T(d, F)] \]

The equivalence \( Q_{43} \equiv Q_{44} \) also follows from Theorem 4.3.9. Queries \( Q_{43} \) and \( Q_{44} \) have parameter sets \( \mathcal{P}_{Q_{43}} = \{B, C, D\} \) with bijection \( \nu \) being the obvious identity mapping. The obvious identity mapping also forms the homomorphism between the shape queries for shapes \( \mathcal{B}_{Q_{44}} = \emptyset = \mathcal{B}_{Q_{44}} \)

\[ \hat{Q}_{43}|_{\emptyset}(\emptyset) := R(A, B) \]
\[ \hat{Q}_{44}|_{\emptyset}(\emptyset) := R(A, B) \]

and shape queries for shapes \( \mathcal{B}_{Q_{43}}(0) = \{\square\} \in \mathcal{TS}_{Q_{43}} \) and \( \mathcal{B}_{Q_{44}}(0) = \{\square\} \in \mathcal{TS}_{Q_{46}} \)

\[ \hat{Q}_{43}(0) := R(A, B), S(B, C), T(B, D), S(C, E), T(D, F) \]
\[ \hat{Q}_{44}(0) := R(A, B), S(B, C), T(B, D), S(C, E), T(D, F) \]

In contrast, queries \( Q_{45} \) and \( Q_{46} \) have parameter sets \( \mathcal{P}_{Q_{45}} = \{B, C\} \) and \( \mathcal{P}_{Q_{46}} = \{B, D\} \), respectively, and so Theorem 4.3.9 does not apply to any other pairings of the four queries. (Even though \( \mathcal{P}_{Q_{45}} \) and \( \mathcal{P}_{Q_{46}} \) are bijective, no homomorphisms exist between \( \hat{Q}_{45}|_{\emptyset} \) and \( \hat{Q}_{46}|_{\emptyset} \) that are consistent with either possible bijection.)

Tuple-Specific Equivalence: Arbitrary Head Graphs

We now consider the problem of deciding tuple-specific equivalence for constant-headed DACQs with arbitrary head graphs. Our approach is a straightforward
extension of the technique we used for queries with single-edge head graphs; the additional complication is that we require reasoning about implication of disjunctive embedded dependencies. A disjunctive embedded dependency $\xi$ can be expressed as a rule of the form

$$P^\xi \rightarrow C^\xi_1 \mid \ldots \mid C^\xi_n$$  \hspace{1cm} (4.20)

where the premise $P^\xi$ and each conclusion $C^\xi_i$ are CQ$^\xi$ bodies; $\xi$ is satisfied by database instance $D$ if each embedding of the premise into $D$ can be extended to an embedding of at least one of the conclusions. Deciding implication between sets of disjunctive embedded dependencies is discussed in Section 4.6.4.

Given any constant-headed DACQ $Q$ and any canonical tuple $t \in \mathcal{C}T_Q$ corresponding to shape $\mathbf{t}_\mathcal{T} \in \mathcal{TS}_Q$, let $\{v_1, \ldots, v_n\} \subseteq L$ be the set of vertices in head graph $G^\mathcal{head}$ that do not occur in shape $\mathbf{t}_\mathcal{T}$ but are children of vertices in $\mathbf{t}_\mathcal{T}$. For each $G \in \text{dags}(\text{body}_Q, \mathbf{t}_\mathcal{T})$ we define a single embedded disjunctive dependency $\xi(G,t)$

$$\xi(G,t) := \begin{cases} \text{body}_{G}^\xi \rightarrow \bot & \text{if } n = 0 \\ \text{body}_{G}^\xi \rightarrow \text{body}_{v_1}^+ \mid \ldots \mid \text{body}_{v_n}^+ & \text{otherwise} \end{cases}$$  \hspace{1cm} (4.21)

and we define dependency set $\Sigma^{(Q,t)}$ as follows.

$$\Sigma^{(Q,t)} := \{ \xi^{(G,t)} | G \in \text{dags}(\text{body}_Q, \mathbf{t}_\mathcal{T}) \}$$  \hspace{1cm} (4.22)

Note that $n = 0$ if and only if $\mathbf{t}_\mathcal{T}$ is terminal if and only if $\text{dags}(\text{body}_Q, \mathbf{t}_\mathcal{T}) = \{ \text{body}_Q \}$ (by property [P2] of Definition 4.2.1).

**Lemma 4.3.21** For any database $D$ and canonical tuple $t \in (Q)^D$ iff $D \not\equiv \Sigma^{(Q,t)}$.

**Proof.** Suppose that $t \in (Q)^D$. Then, there exists some terminal embedding $\gamma : G \sim D$ of some DAG $G \in \text{dags}(\text{body}_Q, \mathbf{t}_\mathcal{T})$. However, $\gamma$ is also an embedding of premise query $P^{\xi(G,t)}$. If $\mathbf{t}_\mathcal{T}$ is not the terminal shape then $\gamma$ does not extend to any conclusion query $C^{\xi(G,t)}$, whereas if $\mathbf{t}_\mathcal{T}$ is terminal then the conclusion of $\xi^{(G,t)}$ is a contradiction to which no embedding can extend; in either case, $D \not\equiv \xi^{(G,t)}$.

Conversely, let $G$ be a maximal DAG in $\text{dags}(\text{body}_Q, \mathbf{t}_\mathcal{T})$ such that $D \not\equiv \xi^{(G,t)}$. There exists embedding $\gamma : P^{\xi(G,t)} \rightarrow D$ that does not extend to any conclusion of $\xi^{(G,t)}$; however, $\gamma$ is a terminal embedding of $G$ that generates tuple $t \in (Q)^D$.

**Corollary 4.3.22** For any canonical tuple $t \in \mathcal{C}T_Q$, $Q \equiv Q'$ iff $\Sigma^{(Q,t)} \equiv_{\text{fin}} \Sigma^{(Q',t)}$.

### 4.4 Queries with Variable Heads

In this section consider the equivalence problem for arbitrary DACQs. Given a DACQ $Q$ whose head tuple $\mathbf{W}$ may contain variables, we start by observing that
the definitions of head graphs, tuple shapes, and shape queries given in Section 4.3.1 are not specific to CH-DACQs. As a result, both necessary conditions for query equivalence that we derived in that section—isomorphism between head graphs (Theorem 4.3.4) and equivalence of shape queries (Corollary 4.3.8)—apply to arbitrary DACQs.

The definition of canonical tuples given in Section 4.3.1 is specific to CH-DACQs. In Section 4.4.1 we extend the definition of canonical tuples to handle queries with variables in the head tuple, using a standard approach of “freezing” the head variables into constants [6, 34]. This allows us to reduce DACQ equivalence to tuple-specific equivalence between CH-DACQs, analogous to the similar reduction for CH-DACQs (Theorem 4.3.14). In the worst case our decision algorithm enumerates an exponential number of canonical freezings per tuple shape; however, in Section 4.4.2 we discuss an optimization that allows us to avoid considering certain freezings. For DACQs that can be expressed without explicit equality—including all CQs and most practical queries expressible in SPCL—this reduces the problem to considering a single canonical freezing per tuple shape.

### 4.4.1 Canonical Freezings and Canonical Tuples

The canonical freezings of a query are defined in terms of a set of symbolic constants \( C_S \). Choose \( C_S \) as any strict subset of \( C \) such that both \( C_S \) and \( C \setminus C_S \) are infinite. Without loss of generality, we will assume that \( C_S \) is chosen so that for any given set of queries, all constants appearing in the queries are from the set \( C \setminus C_S \). Because the constants in \( C_S \) do not appear in query definitions, their precise identity is unimportant (hence the name “symbolic”).

**Proposition 4.4.1** Given any two DACQs \( Q \) and \( Q' \) (satisfying \( (C_Q \cup C_{Q'}) \cap C_S = \emptyset \)), if \( Q \not\equiv Q' \) then there exists a database instance \( D \) with \( \text{adom}(D) \subseteq C_S \cup C_Q \cup C_{Q'} \) such that \( (Q)^D \not\equiv (Q')^D \).

Given any two tuples \( t \) and \( t' \) over domain \( C \cup \{\equiv\} \), we say that \( t \) is \( C_S \)-homomorphic to \( t' \)—denoted \( t \simeq t' \)—if there exists a mapping \( h : C_S \to C \), extended with identity over \( (C \setminus C_S) \cup \{\equiv\} \), such that \( h(t) = t' \). We say that \( t \) is \( C_S \)-isomorphic to \( t' \)—denoted \( t \cong t' \)—if \( t \simeq t' \) and \( t' \simeq t \). We say that \( t \) is more general than \( t' \)—denoted \( t \succ t' \)—if \( t \simeq t' \) and \( t' \not\cong t \).

Given two sets of tuples \( S_1, S_2 \) over \( C \), we say that \( S_1 \) is \( C_S \)-contained in \( S_2 \)—denoted \( S_1 \subseteq S_2 \)—if \( \forall t_1 \in S_1 \exists t_2 \in S_2 , (t_1 \cong t_2) \). We say that \( S_1 \) is \( C_S \)-equivalent to \( S_2 \)—denoted \( S_1 \equiv S_2 \)—if \( S_1 \subseteq S_2 \) and \( S_2 \subseteq S_1 \).

In order to define the canonical tuples for a DACQ, we first define the canonical freezings of a satisfiable CQ*1. Any tuple that can be output by the CQ*1 can be obtained by an isomorphic substitution for the symbolic constants within one of the canonical freezings.
Definition 4.4.2 (Canonical CQ Freezings) Given any satisfiable CQ $Q(W)$ with $C_B \cap C_S = \emptyset$, let $\mathcal{H}_Q$ denote the set of all homomorphisms $h : W \cap B \to C_S \cup C_Q$ that have been extended with identity over all symbols not in $W \cap B$. The (infinite) set of valid freezings of $Q$—denoted $\mathcal{F}_Q^+$—contains the homomorphic images of head tuple $W$ under mappings from $\mathcal{H}_Q$ that are consistent with the equality predicates in the query body.

$$\mathcal{F}_Q^+ := \{ h(W) \mid h \in \mathcal{H}_Q \land h(\text{pred}_Q) \neq 1 \}$$

(4.23)

The set of canonical freezings of $Q$—denoted $\mathcal{F}_Q$—is a minimal subset of $\mathcal{F}_Q^+$ satisfying $\mathcal{F}_Q \equiv \mathcal{F}_Q^+$ (if many such sets exist, we arbitrarily choose one).

Set $\mathcal{F}_Q$ is finite, with size bounded by a function exponential in both $|C_Q|$ and $|W \cap B|$.

Lemma 4.4.3 Given CQ $Q$ and any database instance $D$ over $C_S \cup C_Q$, $T((Q)^D) \subseteq \mathcal{F}_Q$.

Proof. Any embedding $\gamma : Q \to D$ yields $\gamma(W) \in \mathcal{F}_Q^+$ (by Definition 4.4.2), and so $T((Q)^D) \subseteq \mathcal{F}_Q^+ \subseteq \mathcal{F}_Q$.

Definition 4.4.4 (Canonical Tuples) Given DACQ $Q$ in canonical form, the set $\mathcal{C}_T_Q$ of canonical tuples for $Q$ corresponds to the canonical freezings of all satisfiable shape queries.

$$\mathcal{C}_T_Q := \{ t \mid \exists t_s \in T_S_Q \land \hat{Q}|_{t_s} \text{ is satisfiable } \land t \in \mathcal{F}_Q^+_{t_s} \}$$

As defined previously for CH-DACQs, we use $t_s^Q$ to denote the tuple shape from $T_S_Q$ corresponding to a given canonical tuple $t \in \mathcal{C}_T_Q$.

Example 48 Consider the three DACQs $Q_{47}$, $Q_{48}$, and $Q_{49}$ shown in Figure 4.11a. Observe that all three queries have the following set of tuple shapes,

$$T_S_{Q_{47}} = T_S_{Q_{48}} = T_S_{Q_{49}} = \{ t_1, t_{12}, t_{13}, t_{123} \}$$

where the subscripts denote the set of labels in each shape. Figures 4.11b–4.11f show the canonical tuples for $Q_{47}$, $Q_{48}$, and $Q_{49}$, partitioned into sets of canonical freezings for the different tuples shapes. For each tuple shape we have organized the canonical freezings into a lattice in which ancestors are more general than their descendants. Observe that $\mathcal{C}_T_{Q_{47}} = \mathcal{C}_T_{Q_{48}}$ even though the head tuples of $Q_{47}$ and $Q_{48}$ are not isomorphic. In contrast, $\mathcal{C}_T_{Q_{47}} \neq \mathcal{C}_T_{Q_{49}}$ (due to shape $t_{13}$) even though the head tuples of $Q_{47}$ and $Q_{49}$ are isomorphic.

Lemma 4.4.5 Given DACQ $Q$ in canonical form and any database instance $D$ over $C_S \cup C_Q$, $T((Q)^D) \subseteq \mathcal{C}_T_Q$. 

120
Figure 4.11: Three DACQs with their canonical tuples
PROOF. Follows from Definition 4.4.4 and Lemma 4.4.3. □

Lemma 4.4.6 Given two DACQs \( Q \) and \( Q' \) in canonical form, if \( Q \equiv Q' \) then \( \text{CT}_Q \cong \text{CT}_{Q'} \).

PROOF. Follows from Definition 4.4.4 because \( \theta(\text{TS}_Q) = \text{TS}_{Q'} \) (Theorem 4.3.4), and \( \hat{Q}|_{s_t} \equiv \hat{Q}'|_{\theta(s_t)} \) (Corollary 4.3.8) implies \( \text{CF}_{\hat{Q}|_{s_t}} \cong \text{CF}_{\hat{Q}'|_{\theta(s_t)}} \) (by Definition 4.4.2). □

Corollary 4.4.7 If \( Q \equiv Q' \) then \( \text{CT}_Q \) and \( \text{CT}_{Q'} \) can be chosen so that \( \text{CT}_Q = \text{CT}_{Q'} \).

Definition 4.4.8 (Frozen DACQ) Given DACQ \( Q(\overline{W}) \) in canonical form and any canonical tuple \( t \in \text{CT}_Q \) with corresponding shape \( \overline{t}_Q \in \text{TS}_Q \).

- recall that \( t \) is a canonical freezing of shape query \( \hat{Q}|_{\overline{w}_t} \) (cf. Definition 4.4.4),
- recall that \( \hat{Q}|_{\overline{w}_t} \) is a \( CQ^c \) with head tuple \( \overline{W}|_{\overline{w}_t} \) (cf. Definition 4.3.5), and
- recall that there exists some homomorphism \( h_t : \overline{W}|_{\overline{w}_t} \cap \mathcal{B} \rightarrow \mathcal{C}_S \cup \mathcal{C}_Q \) extended with identity over all symbols not in \( \overline{W}|_{\overline{w}_t} \cap \mathcal{B} \) such that \( t = h_t(\overline{W}|_{\overline{w}_t}) \) (cf. Definition 4.4.2).

We define the freezing of \( Q \) with \( t \)—denoted \( Q|_t \)—as the following CH-DACQ.

- head\(_{Q|t}\) := \( \overline{W}|_t \), where \( \overline{W}|_t \) is constructed from \( \overline{W} \) by locating each component \( W_i \) in which \( v \) is a variable, and if \( v \in \overline{t}_Q \) replacing \( W_i \) with \( h_t(W_i) \), whereas if \( v \notin \overline{t}_Q \) replacing \( W_i \) with an arbitrary constant from \( \mathcal{C} \setminus \mathcal{C}_S \) (for simplicity, we always use the constant 0).
- body\(_{Q|t}\) := \( h_t(\text{body}_Q) \)

Example 49 Continuing Example 48, Figure 4.11g shows four frozen DACQs obtained from \( Q_{47} \) by freezing with the most general canonical freezing for each of the four tuple shapes. Note that all of the DACQs in Figure 4.11g are constant-headed, but none of them are in canonical form.

Lemma 4.4.9 Given any DACQ \( Q \) in canonical form, \( \forall t \in \text{CT}_Q. [Q^t \equiv Q|_t] \).

PROOF. Consider any database \( \mathcal{D} \). Suppose that \( t \in (Q)^{\mathcal{D}} \), then there exists some terminal embedding \( \gamma : Q \rightarrow \mathcal{D} \) satisfying \( \hat{\gamma}(\overline{W}) = t \). This implies that there exists some \( G \in \text{dags}(\text{body}_Q, \overline{t}_Q) \) such that \( \gamma \) is a terminal embedding of \( G \) (see equation 4.5). By Theorem 4.3.6, \( \gamma \) is also an embedding of \( \hat{Q}|_{\overline{w}_t} \) into \( \mathcal{D} \) satisfying \( \hat{\gamma}(\overline{W}|_{\overline{w}_t}) = t \), and so, by definition of \( \overline{W}|_t \), \( \hat{\gamma}(\overline{W}|_t) = \hat{\gamma}(\overline{W}|_{\overline{w}_t}) = t = h_t(\overline{W}|_{\overline{w}_t}) \). Because mappings \( \gamma \) and \( h_t \) agree over \( \overline{W}|_{\overline{w}_t} \) and \( h_t \) is the identity mapping over all other symbols, we can decompose embedding \( \gamma \) as \( \gamma = \phi \circ h_t \) for some mapping \( \phi : \mathcal{B} \rightarrow \text{adm}_t(\mathcal{D}) \) extended with identity over constants \( \mathcal{C} \). Because \( Q^t \) has body \( h_t(\text{body}_Q) \), mapping \( \phi \) is an embedding of \( h_t(G) \) into \( \mathcal{D} \) satisfying \( \phi(\overline{W}|_t) = t \). This implies \( t \in (Q|_t)^{\mathcal{D}} \).
Now suppose that \( t \in (Q|t)^D \); then there exists some \( G' \in \text{dags}(h_t(\text{body}_Q)) \) and some terminal embedding \( \phi : G' \sim D \) satisfying \( \hat{\phi}([\overrightarrow{W}|t]) = t \), implying\(^4\) \( G' \in \text{dags}(h_t(\text{body}_Q), \mathfrak{t}_Q^Q) \). Then \( \gamma := \phi \circ h_t \) is a terminal embedding of the DAG \( G \in \text{dags}(\text{body}_Q) \) satisfying \( G' = h_t(G) \), and \( \hat{\gamma}([\overrightarrow{W}) = t \) because \( G \in \text{dags}(\text{body}_Q, \mathfrak{t}_Q^Q) \). □

The following theorem is the main result of this section. It reduces DACQ equivalence to a set of tuple-specific equivalence tests between CH-DACQs, which was the topic of Section 4.3.2.

**Theorem 4.4.10** Given two DACQs \( Q \) and \( Q' \) in canonical form, \( Q \equiv Q' \) iff

1. \( \text{CT}_Q \equiv \text{CT}_{Q'} \), and
2. \( \forall t \in \text{CT}_Q, Q|t' = Q'|t' \).

**Proof.** "If" follows from Proposition 4.4.1, Lemma 4.4.5, and Corollary 4.4.7. "Only if" follows from Lemmas 4.4.6 and 4.4.9. □

### 4.4.2 Avoiding Tuple-Specific Equivalence Tests

In Section 4.3 we reduced CH-DACQ equivalence to testing one tuple-specific CH-DACQ equivalence problem per canonical tuple (Theorem 4.3.14). While Theorem 4.4.10 proves a similar result for arbitrary DACQs, the difference is that the number of canonical tuples per tuple shape depends upon the number of canonical freezings, which can be exponential in the number of variables in the head tuple. It is well-known that to test CQ equivalence it suffices to test boolean CQ equivalence for only the most general freezing (a consequence of the Homomorphism Theorem), and so the algorithm yielded by Theorem 4.4.10 is clearly not optimal for CQs (i.e., DACQs with one vertex in the body). However, the following example illustrates that for arbitrary DACQs it is not sufficient to only consider the most general freezings.

**Example 50** Continuing Example 49, consider the two queries obtained by freezing \( Q_{48} \) with canonical tuples \( abc \) and \( aace \).

\[
\begin{align*}
Q_{48}^{abc}((a, b, c, 0, 0), (a, b, c, 0, 0)) :&= \begin{array}{c}
\text{R}(a, b, c), \\
\text{R}(a, b, D)
\end{array} \\
\text{S}(c, E), a = b, c = D
\end{align*}
\]

\[
\begin{align*}
Q_{48}^{aace}((a, a, c, 0, 0), (a, a, c, 0, 0)) :&= \begin{array}{c}
\text{R}(a, a, c), \\
\text{R}(a, a, D)
\end{array} \\
\text{S}(c, c), a = a, c = D
\end{align*}
\]

\(^4\)We needed to insert arbitrary constants into head \( [\overrightarrow{W}|t] \) of query \( Q|t \) so that \( [\overrightarrow{W}|t] \) retains the same labels and tuple shapes as \( [\overrightarrow{W}] \). If we instead defined \( [\overrightarrow{W}|t] := t \) —which effectively removes any output labels not in \( \mathfrak{t}_Q^Q \)—then \( Q|t \) would output the tuple \( t \) too often (in particular, \( Q|t \) would incorrectly output \( t \) for any terminal embedding \( \phi' : G' \sim D \) of any \( G' \in \text{dags}(h_t(\text{body}_Q), \mathfrak{t}') \) for which \( \mathfrak{t}_Q^Q \in \mathfrak{t}' \)).
From the queries in Figure 4.11g it is straightforward to verify the tuple-specific equivalences. $Q_{47}^{abc \equiv \{abc\}}$, $Q_{48}^{abc \equiv \{abc\}}$, and $Q_{47}^{aac \equiv \{aac\}}$, $Q_{48}^{aac \equiv \{aac\}}$. In fact, for any canonical tuple $t$ with shape $\mathfrak{t}_{12}$, $\mathfrak{t}_{13}$, or $\mathfrak{t}_{123}$ we can show that the $t$-specific equivalence $Q_{47}^{\mathfrak{t} \equiv \mathfrak{t}}$ holds (even though the generic equivalence $Q_{47}^{\mathfrak{t} \equiv \mathfrak{t}}$ may not hold). In contrast, for shape $\mathfrak{t}_1$, tuple-specific equivalence holds for the most general freezing $abc \equiv \{abc\}$, but the database $D_7 = \{R(a,a,c), R(a,a,d), S(c,e)\}$ proves $Q_{47}^{aac \equiv \{aac\}} \not\equiv Q_{48}^{aac \equiv \{aac\}}$ and therefore $Q_{47} \not\equiv Q_{48}$, as demonstrated by the following relations $(Q_{47})^{D_7}$ and $(Q_{48})^{D_7}$.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & $A$ & $B$ & $C$ & $E$ & $B$ \\
\hline
$a$ & a & a & c & e & e \\
$a$ & a & a & d & e & e \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
 & $A$ & $B$ & $C$ & $E$ \\
\hline
$a$ & a & a & c & e \\
a & a & c & e & e \\
a & a & d & e & e \\
\hline
\end{tabular}
\end{table}

The “problem” illustrated by Example 50 is that the tuple-specific equivalence $Q_{47}^{abc \equiv \{abc\}}$, $Q_{48}^{abc \equiv \{abc\}}$ does not imply $Q_{47}^{aac \equiv \{aac\}}$, $Q_{48}^{aac \equiv \{aac\}}$, even though canonical tuple $\{abc\}$ is more general than $\{aac\}$. In order to do better than the naive algorithm that tests tuple-specific equivalence for every canonical tuple, we require an admissible heuristic for identifying when these “implication failures” are guaranteed not to occur. The next theorem proposes a sufficient condition for identifying when the reasoning that $Q[t_1] \equiv Q[t_1]$ also implies $Q[t_2] \equiv Q[t_2]$, for two canonical tuples satisfying $t_1 > t_2$. First, however, we need to revisit how the chase procedure is used to decide implication of disjunctive embedded dependencies. (The reader is referred to page 115 for discussion of how the chase procedure is used to decide implication of embedded dependencies without disjunction.)

Given disjunctive embedded dependency $\xi$ (see equation 4.20) and a set of disjunctive embedded dependencies $\Sigma$, the chase $\text{chase}_{\Sigma}(P^\xi)$ is a (possibly infinite) chase tree in which each chase step has one branch for each conclusion in the chased dependency. Each path $p$ in the tree corresponds to a unique chase sequence of (non-disjunctive) embedded dependencies that defines some (possibly infinite) database $D_p$. The chase procedure returns the conclusion $\Sigma \vdash_{\text{fin}} \xi$ if and only if for each path $p$ it can prove $D_p \vdash \xi$, which it does by exhibiting a finite database $\hat{D}_p \subseteq D_p$—corresponding to some finite prefix of path $p$—that satisfies $\hat{D}_p \vdash \xi$. Given any mapping $h : C \rightarrow C$, and a specific chase invocation that yielded the conclusion $\Sigma \vdash_{\text{fin}} \xi$, we say that the conclusion is sound w.r.t. $h$ if for each path $p$, whenever the database $\hat{D}_p$ contains a contradiction, so does the database $h(\hat{D}_p)$.

**Theorem 4.4.11** Given DACQs $Q$ and $Q'$ in canonical form satisfying $CT_Q = CT_{Q'}$, and given two canonical tuples $t_1, t_2 \in CT_Q$ satisfying $t_1 > t_2$, let $h$ be the $C_S$-homomorphism (extended with identity) satisfying $h(t_1) = t_2$. If $Q[t_1] \equiv Q'[t_1]$ (i.e., $\Sigma(Q[t_1], t_1) \equiv_{\text{fin}} \Sigma(Q'[t_1], t_1)$) can be proved using only chase invocations that are sound w.r.t. $h$, then $Q[t_2] \equiv Q'[t_2]$ (i.e., $\Sigma(Q[t_2], t_2) \equiv_{\text{fin}} \Sigma(Q'[t_2], t_2)$).
Proof. Suppose that $Q^{[t]} \equiv Q^{[s]}$. Due to Definition 4.4.8 and equations 4.21 and 4.22, we observe that $\Sigma(Q^{[s],t_2}) = h(\Sigma(Q^{[t],t_1}))$ and $\Sigma(Q^{[s],t_2}) = h(\Sigma(Q^{[t],t_1}))$.

Consider any chase invocation that proves some implication $\Sigma \vdash_{\text{fin}} \xi$ as part of the proof that $\Sigma(Q^{[t],t_1}) \equiv_{\text{fin}} \Sigma(Q^{[t],t_1})$. Let $T$ denote a (possibly infinite) chase tree for $\text{chase}_T(P^\xi)$, with $D_p$ denoting the (possibly infinite) database defined by path $p$. Let $\hat{T}$ denote the finite prefix of $T$ that was actually explored by the chase invocation, with $\hat{D}_p \subseteq D_p$ the finite database that terminated the search of the chase sequence along path $p$.

Define $D_1 := P^\xi$, and let $D_1 \xrightarrow{(\xi_i, \gamma)} \{D_{1,1}, \ldots, D_{1,k}\}$ be the first chase step in chase tree $T$ enacted by some variable mapping $\gamma : X \rightarrow \text{adom}(D_1)$, where $X$ is the set of premise and free variables of some dependency $\xi_i \in \Sigma$ having $k$ conclusions, and $\gamma$ embeds premise $P^\xi_i$ into $D_1$. Let $D_{1,j}$ denote the database obtained from $D_1$ by chasing with the $j$th conclusion of $\xi_i$.

Now consider the dependency set $h(\Sigma)$ and the dependency $h(\xi)$ which has premise $P^h(\xi) = h(P^\xi) = h(D_1)$. Because $\xi_i$ and $h(\xi_i)$ have the same variables ($h$ only remaps certain constants), it follows that $h \circ \gamma : X \rightarrow \text{adom}(h(D_1))$ is an embedding of $h(P^\xi_i) = P^h(\xi_i)$ into $h(D_1)$. Furthermore, $h(D_{1,j})$ is the database obtained from $h(D_1)$ by chasing with the the $j$th conclusion of $h(\xi_i)$. It follows that $h(D_1) \xrightarrow{(h(\xi_i), h\gamma)} \{h(D_{1,1}), \ldots, h(D_{1,k})\}$ is a valid—although possibly spurious—first step for some chase invocation proving $h(\Sigma) \vdash_{\text{fin}} h(\xi)$. By induction, it follows that $h(\hat{T})$ is the prefix of a valid chase tree—possibly containing spurious chase steps—for $\text{chase}_{h(\Sigma)}(P^h(\xi))$.

Now consider any database $\hat{D}_p$ at the leaves of $\hat{T}$. We know that $\hat{D}_p \models \xi$, which implies that either there exists an embedding $\gamma : C_j^\xi \rightarrow \hat{D}_p$ of one of the conclusions of $\xi$, or $\hat{D}_p$ contains a contradiction (see page 115). If $\gamma$ exists, then, by the argument above, $h \circ \gamma$ is an embedding of $h(C_j^\xi) = C_j^h(\xi)$ into $h(\hat{D}_p)$. Alternatively, the final chase step leading into $\hat{D}_p$ may have derived a contradiction, either explicitly (i.e., a dependency of the form $P^\xi_i \rightarrow \perp$) or implicitly by equating two distinct constants $a, b \in C$; as long as the chase invocation is sound w.r.t. $h$, $h(\hat{D}_p)$ will also contain a contradiction. In either case, $h(\hat{D}_p) \models h(\xi)$. Hence, the chase tree $h(\hat{T})$ proves $h(\Sigma) \equiv_{\text{fin}} h(\xi)$. \hfill \Box

Corollary 4.4.12 For any shape $\mathfrak{t} \in T_{\text{shape}}$ and corresponding shape $\mathfrak{t}' \in T_{\text{shape}}^\prime$, if

- all of the equality predicates in $\text{body}_Q$ occur within $G_{\mathfrak{t}}^{\text{min}} \in \text{dags(\text{body}_Q)}$, and
- all of the equality predicates in $\text{body}_{Q'}$ occur within $G_{\mathfrak{t}'}^{\text{min}} \in \text{dags(\text{body}_{Q'})}$,

then only the most general canonical tuple of shape $\mathfrak{t}$ needs to be considered when deciding $Q \equiv Q'$.\footnote{The chase step $D_1 \xrightarrow{(\xi_i, \gamma)} \{D_{1,1}, \ldots, D_{1,k}\}$ is spurious if $D_1 \not\models \gamma(\xi_i)$. The introduction of a spurious chase step can cause an otherwise finite chase sequence to become infinite, but it does not change the soundness of the conclusion $\Sigma \vdash_{\text{fin}} \xi$.}
Corollary 4.4.12 can yield an exponential reduction in the number of canonical
tuples that need to be considered. In particular, for queries expressible without
explicit equality predicates, Corollary 4.4.12 implies that the number of tuple-
specific equality tests required is independent of the number of variables in the query
head. Explicit equality is only required to equate ancestor variables to constants
or to each other, which corresponds to a class of LOJ predicates that occurs only
rarely in real world workloads. The vast majority of LOJ predicates in real-world
SPCL workloads equate one attribute from each side of the LOJ, which can be
expressed as a DACQ without explicit equality by using shared variables.

Example 51 Consider the following queries $Q_{50}$ and $Q_{51}$ which are written over the
sample company database schema from Chapter 1 (cf. Figure 1.1).

$$Q_{50}(M \Box, O \Box, P \Box, Y \Box) := \text{EC}(C, M, ‘C’) \rightarrow 0(O, C, D) \rightarrow \text{LI}(O, L, P, Y)$$

$$Q_{51}(M \Box, O \Box, P \Box, Y \Box) := \text{EC}(C, M, T) \rightarrow 0(O, C, D), T = ‘C’ \rightarrow \text{LI}(O, L, P, Y)$$

Query $Q_{50}$ returns the names of all Corporate customers, along with the order ids of any
associated orders (if they exist), and the price and quantity of any associated lineitems
(if they exist); this corresponds to a typical usage of the LOJ operator, and does not
require explicit equality when expressed as a DACQ. Query $Q_{51}$ is more esoteric in
that it returns the names of all customers, but only returns the associated orders and
lineitems for the Corporate customers; this type of LOJ predicate necessitates explicit
equality when expressed as a DACQ.

4.5 Soundness and Completeness

We have reduced DACQ query equivalence to deciding tuple-specific equivalence
between CH-DACQs (Theorems 4.3.14 and 4.4.10), and tuple-specific CH-DACQ
equivalence to finite equivalence of sets of disjunctive embedded dependencies (Corol-
lary 4.3.22). Certainly it is not ideal to reduce a problem we only know to be
NP-hard to one that is undecidable in general. Given a DACQ equivalence prob-
lem, our algorithm might construct an instance of an implication problem for which
we cannot guarantee chase termination (which does not prove that DACQ equiv-
ance is undecidable). In this section we briefly discuss two strategies for dealing
with difficult implication problems, each of which preserves soundness (but not
completeness) of the equivalence test.

4.5.1 Early Termination

As described in Section 4.3.2, even when database $D := \text{chase}_P(P \hat{\gamma})$ is infinite, the
chase algorithm may be able to conclude $\Sigma \models^\text{fin} \xi$ based upon a finite sequence of
chase steps yielding some $\hat{D} \subset D$. Early termination is the obvious pragmatic solution: run the chase algorithm for a bounded amount of time, and return “I don’t know” if by the end of the time limit neither $\Sigma, x \in \text{fin} \, \xi$ nor $\Sigma, x \not\in \text{fin} \, \xi$ has been derived. In the simplest form, the number of chase steps can simply be fixed; unfortunately, this yields a non-deterministic result depending upon the order of chase steps. Alternatively, bounding either the dependency depth—the maximum length of any chain of order-dependencies between chase steps—or the Skolem depth—the nesting depth of the Skolem functions (constants) introduced by chase steps—yields a deterministic result with specific properties that can be used to characterize subclasses of database instances over which the implication does hold. Recently, more sophisticated methods have been proposed that dynamically monitor the chase execution and terminate based upon violation of some boundary condition [81].

4.5.2 Constraint Relaxation

The alternative to early termination is to invoke the chase only when it is guaranteed to terminate. Given a dependency set $\Sigma$ that does not guarantee a terminating chase procedure, the main idea behind constraint relaxation is to replace $\Sigma$ with a set $\hat{\Sigma}$ that both satisfies $\Sigma, x \in \text{fin} \, \hat{\Sigma}$ and guarantees a terminating chase procedure.

**Proposition 4.5.1** Given dependency $\xi$ and set $\Sigma$, let $\hat{\Sigma}$ be any dependency set satisfying $\Sigma, x \in \text{fin} \, \hat{\Sigma}$. If $\hat{\Sigma}, x \in \text{fin} \, \xi$ then $\Sigma, x \in \text{fin} \, \xi$.

The trick is in constructing a useful set $\hat{\Sigma}$ without requiring a chase procedure to explicitly prove $\Sigma, x \in \text{fin} \, \hat{\Sigma}$. One technique is to construct $\hat{\Sigma}$ from $\Sigma$ by identifying dependencies that cause violations of the termination guarantee, and for each such dependency either delete it from $\Sigma$ or relax it by adding atoms/predicates to the premise and/or deleting atoms/predicates from the conclusion.

**Example 52** Consider the following dependency $\xi$ and dependency set $\Sigma = \{\xi_1, \xi_2\}$:

- $\xi : [R(X,Y)] \rightarrow [S(Y,Z)]$
- $\xi_1 : [R(A, B), R(A, C)] \rightarrow [S(B, D), S(C, D)]$
- $\xi_2 : [R(A, B), R(A, C)] \rightarrow [R(B, E)]$

Set $\Sigma$ does not guarantee chase termination, and indeed the database chase$_{\Sigma}([R(x, y)])$ is not finite. If we choose $\hat{\Sigma} = \{\xi_1\}$ then $\Sigma, x \in \text{fin} \, \hat{\Sigma}$ is trivial, and $\hat{\Sigma}$ guarantees chase termination (due to weak acyclicity—see Section 4.6.4). Then, chase$_{\hat{\Sigma}}([R(x, y)]) = \{R(x, y), S(y, d_1)\}$ proves $\hat{\Sigma}, x \in \text{fin} \, \xi$ and therefore $\Sigma, x \in \text{fin} \, \xi$.

4.6 Relevant Literature

To the best of our knowledge, there is no previous literature that directly addresses the problem of query equivalence for queries containing outer joins. There is, however, a wide variety of literature relevant to various aspects of the problem and/or
our solution to it, and so in this section we briefly survey literature surrounding five problems that seem the most pertinent—algebraic optimization of outer joins, containment for query languages with negation, query containment in the presence of nulls, finite implication of dependencies (equivalently, query containment under a set of dependencies), and implication of dependencies over relations containing nulls.

4.6.1 Algebraic Optimization of Outer Joins

Research into algebraic optimization of queries containing outer joins was pioneered by Galindo-Legaria in his doctoral research with his supervisor, Rosenthal [96, 40, 36]. They propose and study the problem of how to generate all equivalent join re-orderings of a given “outer-join tree,” which is an algebraic expression composed of inner, left (right) outer, and full outer joins, together with selection. While this problem is straightforward for inner joins because both commutativity and associativity hold, outer joins are not associative in general. The culmination of this work is a set of transformation rules that is complete for enumerating equivalent outer-join trees [41]. While a complete set of algebraic transformations could serve as the basis for a decision procedure for query equivalence, it is important to note the lack of a projection operator—which means that relation names can be assumed to be unique (an assumption that simplifies the equivalence problem for CQs from NP-complete to linear time). Instead, the main contribution of this work is the study of the conditions under which outer joins are associative.

Subsequent work by Galindo-Legaria proposes join-distinct normal form (JDNF) as a canonical representation for outer-join trees [37]. JDNF is formed by rewriting the outer joins in terms of inner join and minimal union—see equation 4.2—and then commuting all inner joins below all minimal unions. Given a DACQ $Q$ containing unique relation names and whose head tuple does not project away any attributes, JDNF corresponds to replacing union with minimal union within “shape query” $\hat{Q}$ (see Definition 4.3.5); hence, the JDNF is exponential in the size of $Q$. JDNF has been used as the basis for a view rewriting algorithm that handles outer joins [71], but arguably its main contribution is conceptual—it explicitly exposes the interactions between the predicates of the inner and outer-join operators in the join tree.

Later authors have proposed alternative canonical representations for outer-join trees. Bhargava, Goel, and Iyer define a canonical representation that encodes the join predicates within a hypergraph—offering a visual representation of the interactions between join predicates that is much more concise than JDNF [12]. They then show how hypergraph connectivity can be used to guide the algebraic manipulations in order to enumerate all equivalent outer-join trees. Rao, Pirahesh, and Zuzarte propose a canonical representation for outer-join trees that generalizes the standard normal form for SPC queries [6, Ch. 4], by replacing Cartesian product with outer Cartesian product and selection with nullification [94]. Whereas the techniques of
both Galindo-Legaria and Bhargava et al. are geared towards transformation-based enumeration of equivalent join trees, the primary contribution of Rao et al.’s canonical form is that it facilitates integration into bottom-up dynamic-programming join enumeration.

Recently, Hill and Ross have proposed an algorithm for enumerating equivalent join trees that converts outer-join trees into inner join trees over “derived relations” [60]. Their focus is on encapsulating all of the logic pertaining to the non-associativity of outer joins into a pre-computation step, allowing the join enumeration phase to assume associativity. By removing the outer-join-specific logic from the join enumeration phase, they are able to extend existing methods for expanding an optimizer’s search space—in particular, semijoin reduction, which is an important technique within distributed query processing environments—to queries containing outer joins.

4.6.2 Queries with Negation

The LOJ operator can be defined in terms of set difference (see equation 4.1), which accounts for its non-monotonicity with respect to null-padded tuples. It is well known that adding unrestricted set difference to the conjunctive algebra yields the full relational algebra, for which query containment is undecidable [6]. However, there are interesting restrictions on the negation operator that yield decidable query containment.

The most fundamental result is due to Sagiv and Yannakakis, who show that the containment problem is only NP-complete for unions of “elementary differences” (an elementary difference is a difference operations between two CQs) [99]. Their complexity result hinges upon the fact that within this class of expressions, projection never occurs after a union or difference operator. As a consequence, query containment is \( \Pi^p_2 \)-complete for arbitrary relational algebra expressions as long as the projection operator never occurs above difference—which implies that SPCL equivalence is decidable as long as projection never occurs above LOJ. Note the close correspondence between this class of SPCL expressions and the class of outer-join trees studied by Galindo-Legaria, described above. The analogous syntactic restriction for DACQs is that all local variables of all non-leaf nodes appear in the query head, labelled by the node to which they are local. For this special case, the necessary condition in Corollary 4.3.8 and the sufficient condition in Theorem 4.3.9 converge to form a complete characterization of DACQ equivalence.

Other restricted forms of negation have also been explored:

- Ullman shows that extending CQs with safe atomic negation (i.e., negation of individual atoms whose variables all occur within positive atoms) causes containment to be \( \Pi^p_2 \)-complete [106]. Whereas Ullman’s algorithm naively generates canonical databases, for this same class Wei and Lausen propose a recursive decision procedure that essentially performs a chase algorithm over
sets of disjunctive full dependencies. They then extend their algorithm—without any increase in complexity—to decide containment for unions of CQs with safe atomic negation [111].

- Klug shows that extending CQs with the predicates $\neq$, $\leq$, and $<$ keeps the CQ containment (and equivalence) problem in $\Pi^p_2$ [70], assuming an abstract totally-ordered underlying domain. The matching lower bound is due to van der Meyden, who proves that just $\neq$ is enough to make the CQ containment problem $\Pi^p_2$-hard [108]. Ibarra and Su generalize this result further to consider CQs with linear arithmetic constraints, and show the containment and equivalence problems to be decidable in double exponential time but not non-deterministic single exponential time when queries are evaluated over the integers, and decidable in double exponential time but not non-deterministic polynomial time when constant-free queries are evaluated over the real numbers [61].

Unfortunately, there does not appear to be an obvious way to reduce equivalence of arbitrary DACQs either to or from the equivalence or containment problems for any of these extensions to CQs.

### 4.6.3 Query Containment with Null Values

Only recently has the interaction between null values and query containment been considered. Farré, Nutt, Teniente, and Urpi study the containment problem for CQs over database instances containing null values, under the assumption that equality and join predicates are evaluated using 3-valued logic [34]. They show that the containment problem remains NP-complete for boolean queries, and corresponds to the existence of homomorphisms that map join variables onto other join variables. Interestingly, the problem becomes $\Pi^p_2$-complete as soon as the language allows either output variables or an explicit $isNull$ predicate (or both).

### 4.6.4 Query Containment under Dependencies

Because embedded dependencies can be expressed as a pair of CQs—the premise and the conclusion—finite implication of embedded dependencies corresponds precisely to containment of CQs under a set of dependencies. It is standard practise to transform sets of embedded dependencies into an equivalent form in which every dependency is either tuple-generating or equation-generating [6, Ch. 10], and most of the literature that we mention in this section assume this distinction. We did not make this distinction when we introduced embedded dependencies in Section 4.3 because it is orthogonal to our reduction of query containment/equivalence to dependency implication; furthermore, sets of disjunctive embedded dependencies can not be decomposed in this way.

The CQ containment problem is decidable for any class of dependencies that allows a terminating chase procedure. Well-known examples include the class of
full dependencies—i.e., dependencies that do not contain existentially-quantified variables—and the class FDs + JDs + acyclic INDs [6, Ch. 9–10]. Johnson and Klug show that containment of CQs under FDs and INDs is decidable—even though the chase does not terminate—as long as the INDs are key-based; however, their proof allows for infinite counter-example databases [67]. Rosati extends Johnson and Klug’s work and shows that CQ containment under INDs or key-based FDs + INDs is finitely controllable, i.e., that the finite-model and infinite-model problems coincide [95]. To do this, he uses a complex process for generating chase constants that is periodic (thereby bounding the total number of constants) but whose period is provably long enough so that any infinite chase sequence that serves as a counter-example to containment can be mapped to a “periodic” chase sequence that is also a counter-example to containment.

For arbitrary embedded dependencies the problem is known to be undecidable; however, there has been significant research into trying to identify when CQ containment is decidable for a particular set of embedded dependencies. Calì, Gottlob, and Kifer generalize Johnson and Klug’s work by considering CQ containment under arbitrary tuple-generating and equality-generating dependencies [15]. They define classes of guarded dependencies for which CQ containment is EXPTIME-complete with bounded schema size, or 2-EXPTIME-complete in general. With guarded dependencies the chase is still unbounded, but the chase sequence can be modelled by what they call a squid decomposition, which restricts the possible interactions between constants generated by different chase paths. Like Johnson and Klug, their decidability results assume a data model allowing infinite databases; it is not known whether Rosati’s technique can be extended to show finite controllability for this more general problem.

Other authors have focused on identifying when the chase is guaranteed to terminate. Deutsch and Popa propose a termination condition called weak acyclicity [31, 33]; if set Σ is weakly acyclic, then any chase over set Σ is guaranteed to terminate. Weak acyclicity is directly generalized to a termination condition called stratification by Deutsch, Nash, and Remmel [30]. Those authors also show that it is undecidable to identify when a given Σ allows a terminating chase procedure. Meier, Schmidt, and Lausen generalize the stratification condition further, defining an infinite hierarchy of sufficient termination conditions, where each level k in the hierarchy corresponds to a proof that any chase over set Σ will not generate fresh constants with Skolem depth greater than k.

Most of the research on query containment under dependencies focuses on embedded dependencies for which the chase is a linear search of a sequence of chase steps. Beeri and Vardi propose extending the chase procedure to a tree search to handle dependencies containing disjunction in the conclusion [10]. Although not widely used, this extension does add expressive power. Consider a set Σ containing disjunctive dependencies. Let Š be the corresponding set of “flattened”

\footnote{The condition was derived collaboratively by Deutsch and Popa, but published independently (with other authors) under the names “weak acyclicity” [33] and “stratified-witness” [31].}
dependencies, formed by decomposing each $\xi \in \Sigma$ into separate (non-disjunctive) dependencies for each conclusion. Any path within a chase tree over $\Sigma$ corresponds to a valid prefix of a chase sequence over $\hat{\Sigma}$. This implies that decidability under $\hat{\Sigma}$ is a sufficient (although not necessary) condition for decidability under $\Sigma$, making all of the research on decidability for non-disjunctive dependencies directly applicable to disjunctive dependencies.

4.6.5 Dependencies over Relations with Nulls

As mentioned above, the problem of conjunctive query containment under embedded dependencies corresponds precisely to (finite) implication of embedded dependencies. Much of the historic work on dependency analysis is motivated by problems related to database schema design rather than query equivalence, and often considers much simpler dependency classes than arbitrary embedded dependencies. The most fundamental result is the axiomatization of inference over sets of functional dependencies proposed by Armstrong [7], which has been extended to inference over sets of FDs and MVDs [6, Sect. 8.2]. A key assumption in this work (and in the work surveyed on query equivalence under dependencies) is that the data model does not include NULL values. Literature that extends dependency analysis to relations containing NULL values is pertinent to Section 4.2—specifically, to our definition and use of nullability dependencies to characterize property [P5] of DACQ canonical form.

Vassiliou pioneered the analysis of dependencies over relations containing NULL values [109, 110]. He points out that a NULL value can be interpreted in two different ways—either to mean “unknown value,” or to mean “no information” [109]. The “unknown value” interpretation assumes that there exists some substitution of domain values for NULLs yielding tuples that occur within some unknown “complete” version of the relation; hence, relations with NULLs under this interpretation are often referred to as “incomplete relations”—a concept further developed by Imielinski and Lipski [62]—and reasoning about them requires considering the set of possible worlds. In contrast, the “no information” interpretation is more primitive, making no presumption that there exists any underlying completed version of the relation.

Vassiliou extends the concept of functional dependencies to relations containing NULLs, assuming the unknown value interpretation [110]. He classifies FDs as either strong or weak. Vassiliou’s definitions of strong and weak FDs are in terms of lists of syntactic conditions, but more intuitive characterizations used by Levene and Loizou [73] are that an FD is strong if under all possible worlds the completed relation obeys the FD, whereas an FD is weak if there exists at least one possible world under which the completed relation obeys the FD. Vassiliou shows that Armstrong’s axioms are sound and complete for inference over strong FDs. Also assuming the unknown value interpretation, Levene and Loizou extend Armstrong’s axioms to obtain a set of axioms that is sound and complete.
for inference over sets of both strong and weak FDs [73]. Motivated by schema design for databases containing nested relations, Levene and Loizou also define an extension to the nested relational model to handle NULL values, and they consider “null extended data dependencies”—specifically, null extended FDs, MVDs, and join dependencies (JDs)—which extend traditional FDs/MVDs/JDs to both nested relations and NULL values (interpreted as unknown values) [72].

Atzeni and Morfuni study implication of FDs under the “no information” interpretation [8]. They define FD $X \rightarrow Y$ to hold over a relation $R$ when for any two tuples $t, t' \in R$, if $t[X] = t'[X]$ and $t[X], t'[X]$ do not contain NULL, then $t[Y] = t'[Y]$. Under this definition of FDs, Armstrong’s axiom of transitivity fails to hold, and so they introduce a concept they call “null transitivity,” which they use to devise a modified set of axioms that are sound and complete for implication of FDs over relations containing NULLs. Our definition of a nullability dependency (Definition 4.2.2) is convenient for our purposes in this chapter, but it can be equivalently expressed in terms of Atzeni and Morfuni’s definition of an FD, as follows.

**Proposition 4.6.1** Given a query $Q$ and a nullability dependency $ABC \Rightarrow_v$ with $v \in \text{body}_Q$ and $\{A, B, C\} \subseteq \mathcal{B}_v^\text{anc}$, let $c$ be any constant, let $\{u_1, \ldots, u_k\}$ be the set of ancestor nodes of $v$, and let $u_A$ denote the ancestor node of $v$ satisfying $A \in \mathcal{L}_{u_A}$, with $u_B$ and $u_C$ defined analogously. Then, $Q \models ABC \Rightarrow_v$ if and only if over every database $D$, the relation $(\text{body}_Q)^D$ (i.e., the result of $Q$ prior to projection onto the head attributes) satisfies the FD $A[u_A]B[u_B]C[u_C, u_A] \cdots [u_k] \rightarrow c[\mathbf{v}]$ (under Atzeni and Morfuni’s definition).

Whereas all of the literature discussed above addresses dependencies in the context of database schema design, Paulley in his doctoral dissertation analyzes how FDs are introduced by and propagated through algebraic queries [92]. Paulley defines a “strict” FD $X \rightarrow Y$ to hold over a relation with NULLs if the relation satisfies the FD under the classical definition, treating NULL as a normal domain value. He defines a “lax” FD $X \leftrightarrow Y$ to hold if the subset of the relation not containing a NULL value in either $X$ or $Y$ satisfies the FD $X \rightarrow Y$. Of particular note, Paulley works out the interaction between strict/lax FDs and the left outer-join operator [92, Ch. 3]. In the introduction to the next chapter we raise but leave open the problem of testing query validity for encoding queries with DACQ query bodies and encoding heads containing arbitrary assignments of node labels to index attributes. This problem requires testing that the query body implies that the query result—which is a relation containing NULL (i.e., $\equiv$) values—always satisfies the strict FD $\overline{F}_{[1,d]} \rightarrow \overline{V}$. We conjecture that this problem could be solved using Paulley’s results to characterize the effect of arbitrary assignments of node labels within the index attributes.
Chapter 5

Encoding-Equivalence of Hierarchical CEQ’s

In Chapter 3 we considered the encoding-equivalence problem for CEQs—that is, encoding queries whose bodies are CQs—and we demonstrated an equivalence procedure that hinges upon using the encoding signature to convert the query head to a normal form. In this chapter we extend that approach to encoding queries whose bodies contain hierarchical edges (HCEQs).

We started Chapter 3 by characterizing CEQ validity, after which we characterized encoding-equivalence between any pair of valid CEQs. CEQ validity corresponds to a simple syntactic condition (see Definition 2.3.4 and Lemma 3.0.1); unfortunately, hierarchical edges interact with functional dependencies, making the validity problem non-trivial for arbitrary HCEQs. Because the HCEQ validity problem is orthogonal to our original problem of deciding $\text{COCQL}^T$ equivalence, when considering HCEQ encoding-equivalence we will restrict our attention to the subclass of HCEQs that can be output by the $\text{Encode}()$ procedure from Section 2.3.2, which are known to be valid (see Theorem 2.3.6).

Our main result in this chapter is a homomorphism-based sufficient condition for HCEQ encoding-equivalence akin to a synthesis of Theorem 3.2.4 (complete characterization of CEQ encoding-equivalence) with Theorem 4.3.9 (sufficient condition for HCQ/DACQ equivalence). In Section 5.1 we enhance the encoding normal form from Chapter 3 to handle HCEQs. In Section 5.2 we prove a homomorphic condition between HCEQs in encoding normal form to be sufficient for implying encoding-equivalence, while in Section 5.3 we give two examples for which the homomorphic condition fails, but for which we can show that encoding-equivalence holds. We are not aware of any literature specific to the equivalence of queries outputting complex objects (or relational encodings of complex objects) containing empty collections, beyond that already discussed in previous chapters, and so this chapter does not include any further survey of literature.
5.1 A Normal Form for HCEQs

In Chapter 3 we defined $\xi$-normal form for CEQs by identifying a set of core indexes. This definition of core indexes (Definition 3.1.11) explicitly uses MVDs, and implicitly relies upon the FD $\xi = \{A \rightarrow 0\}$. However, hierarchical edges (and the subsequent introduction of node labels into the query head) interact with both MVDs and FDs. The following example illustrates this interaction, and relates it to the nullability dependencies that we defined in Chapter 4.

Example 53 Consider the following HCQ $Q_{52}$, along with the two CQs $Q_{53}$ and $Q_{54}$ corresponding to the two elements of $\text{dags}(\text{body}_{Q_{52}})$.

$$Q_{52}(A \square, B \square, 0 \square) := \begin{array}{c} \square R(A, B) \\ \square R(B, C) \end{array}$$
$$Q_{53}(A, B, 0) := R(A, B)$$
$$Q_{54}(A, B, 0) := R(A, B), R(B, C)$$

Over attribute set $\mathcal{W} = \{A, B, 0\}$, the FD $A \rightarrow 0$ and the MVD $A \rightarrow 0$ are both trivial (because 0 is a constant), and so $Q_{53} \models A \rightarrow 0$, $Q_{53} \models A \rightarrow 0$, $Q_{54} \models A \rightarrow 0$, and $Q_{54} \models A \rightarrow 0$. In contrast, over labelled attribute set $\mathcal{W} = \{A \square, B \square, 0 \square\}$, neither the FD $A \square \rightarrow 0 \square$ nor the MVD $A \square \rightarrow 0 \square$ is trivial, because the database instance $D_8 := \{R(a, a), R(a, b)\}$ proves $Q_{52} \not\models A \square \rightarrow 0 \square$ and $Q_{52} \not\models A \square \rightarrow 0 \square$.

The violation of the FD $A \square \rightarrow 0 \square$ within relation $(Q_{52})^{D_8}$ is caused by two terminal embeddings of $Q_{52}$ that agree on variable $A$, but one embedding extends to vertex $y$ while the other does not. Restated in the terms of nullability dependencies (see Definition 4.2.2), the source of this FD violation is the nullability dependency violation $Q_{52} \not\models A \equiv y$. Similarly, the violation of MVD $A \square \rightarrow 0 \square$ is related to the fact that the nullability set $\mathcal{S}_{Q_{52}} = \{B\}$ (see Corollary 4.2.7) is not subsumed by the left-hand-side of the MVD.

We now extend the definition of $\xi$-NF from Chapter 3 to handle HCEQs originating from the $\text{ENCODE}()$ procedure. Our definition of core indexes relies upon testing whether a CQ $\xi$ satisfies a particular MVD. In Section 3.1.1 we considered the comparable problem for CQs, and those results can be extended to CQs by accounting for the equality predicates in a straightforward (albeit tedious) manner; hence, we do not discuss that problem further here.

Our definition of the core indexes also relies on nullability set $\mathcal{S}_Q$ (see Corollary 4.2.7). To facilitate calculating $\mathcal{S}_Q$ via the theorems in Section 4.2.1, we first convert the query body to DACQ canonical form (assuming $\mathcal{W} := \mathcal{I}_{[1,d]} \mathcal{V}$ for the
purposes of Definition 4.2.1). Conversion to DACQ canonical form may cause the tree-shaped query body to become a DAG, and so the resulting query is no longer a hierarchical CEQ. We also wish to restrict our attention to encoding queries originating from the reduction in Chapter 2; for that reason, we define the following class of encoding queries which we call derived DACEQs.

**Definition 5.1.1 (Derived DACEQs)** A directed acyclic conjunctive encoding query (DACEQ) is a generalization of an HCEQ (see Definition 2.3.3) to have a DACQ body (see Definition 4.1.2). We call DACEQ $Q$ derived if there exists some COCQL\{$i_5$,$j_5$,$j_5$} query $Q'$ such that $Q = \text{CANONICAL} (\text{ENCODE}(Q'))$.

**Example 54** Queries $Q_{55}$ and $Q_{56}$ shown in Figure 5.1 are both derived DACEQs, because they are the result of converting HCEQs $Q_{20}$ and $Q_{21}$ (cf. Figures 2.18 and 2.19) to DACQ canonical form. (To improve readability, we have merged the labels for each index tuple.)

The following lemma lists properties of derived DACEQs that follow directly from the definitions of procedures \text{ENCODE}() (see Section 2.3.2) and \text{CANONICAL}() (see Algorithm 4 in Section 4.2.2). Recall from the definition of encoding queries (Definition 2.3.3) that $\mathcal{I}_i$ is a sequence of labelled indexes, and that we use $\mathcal{I}_i$ and $\mathcal{I}_i$ to denote the corresponding sets of labelled and unlabelled indexes, respectively.

**Observation 5.1.2** Given any derived DACEQ $Q(\mathcal{I}_1; \ldots; \mathcal{I}_d; \mathcal{V})$, query $Q$ satisfies the following properties:

[P6] $Q$ is in DACQ canonical form (see Definition 4.2.1).

[P7] For any $i \in [1, d]$, index set $\mathcal{I}_i$ is either

(a) a subset of $\mathcal{I}_i$ (i.e., $\emptyset$, or $\{v_i\}$)

(b) a non-empty set of variables and constants all labelled with the same node $v_i \in \text{body}_Q$. It is possible that $v_i = v_j$ for $i \neq j$. 

137
[P8] For any variable $X \in \mathcal{I}_{[1,d]}$, let $i \in [1,d]$ be the smallest integer such that $X \in \mathcal{I}_i$. Then, $X \in \mathcal{L}_v$.

[P9] For any $i \in [1,d]$ such that $\hat{\mathcal{I}}_i \not\subseteq \{\emptyset\}$, the parameters of body $v_i$ satisfy $\mathcal{P}_v_i \subseteq \mathcal{I}_{[1,i-1]}$, implying $Q \models \mathcal{I}_{[1,i-1]} \models v_i$ by Lemma 4.2.3.

[P10] For every node $v \in \text{body}_Q$, there exists at least one level $i \in [1,d]$ such that $v = v_i$.

Example 55 Continuing Example 54, the reader can verify that queries $Q_{55}$ and $Q_{56}$ each satisfy all of the properties [P6]–[P10]. Notably, property [P7] guarantees that any valuation of index tuple $\mathcal{I}_i$ is either entirely $\emptyset$ values or does not include any $\emptyset$ value. Property [P9] guarantees that for any valuation of index tuples $\mathcal{I}_{[1,i-1]}$, the possible valuations of index tuple $\mathcal{I}_i$ cannot include both the entirely $\emptyset$ tuple and a tuple not containing $\emptyset$. For a concrete example, the reader is invited to refer back to encoding relations $R_8 = (Q_{55})^D$ and $R_9 = (Q_{56})^D$ in Figures 2.20 and 2.21, respectively. (Proposition 4.2.13 guarantees $Q_{55} \equiv Q_{20}$ and $Q_{56} \equiv Q_{21}$.)

Definition 5.1.3 (Core Indexes) Given any derived DACEQ $Q(\mathcal{I}_1; \ldots; \mathcal{I}_d; \mathcal{P})$ with nullability set $\mathcal{S}_Q$, a length-$d$ signature $\hat{\mathcal{S}}$, and an integer $i \in [1,d]$, for each $G \in \text{dags(\text{body}_Q)}$ satisfying $v_i \in G$, define set $\mathcal{W}(G,i) := \mathcal{B}_G^+ \cap (\mathcal{I}_{[1,i]} \cup \mathcal{I}_{[i+1,d]}^\uparrow)$, and let $Q_{(G,i)}$ be the following CQ.

$$Q_{(G,i)}(\mathcal{W}(G,i)) \models \text{body}_G^+ \tag{5.1}$$

Define the set of core indexes at level $i$ relative to $\hat{\mathcal{S}}$—denoted $\hat{\mathcal{I}}_i^{\hat{\mathcal{S}}}$—as a minimal subset of $\hat{\mathcal{I}}_i$ such that $\mathcal{I}_i$ (the unlabelled version the indexes in $\hat{\mathcal{I}}_i^{\hat{\mathcal{S}}}$—see Section 2.3.1) satisfies both $(\mathcal{V} \cup \mathcal{S}_Q) \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{I}_{[i+1,d]}^\uparrow \cup \mathcal{C}$ and the following signature-specific conditions.

<table>
<thead>
<tr>
<th>$\hat{\mathcal{S}}_i$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>$\mathcal{I}<em>i \subseteq \mathcal{I}</em>{[1,i-1]} \cup \mathcal{I}_{[i+1,d]}^\uparrow \cup \mathcal{C}$</td>
</tr>
<tr>
<td>$s$</td>
<td>$\forall G \text{ containing } v_i : Q_{(G,i)}(\mathcal{B}<em>G^+ \cap (\mathcal{I}</em>{[1,i-1]} \cup \mathcal{I}_{[i+1,d]}^\uparrow)) \models \mathcal{B}<em>G^+ \cap \mathcal{I}</em>{[i+1,d]}^\uparrow$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\forall G \text{ containing } v_i : Q_{(G,i)}(\mathcal{B}<em>G^+ \cap \mathcal{I}</em>{[1,i-1]} \models \mathcal{B}<em>G^+ \cap \mathcal{I}</em>{[i+1,d]}^\uparrow)$</td>
</tr>
</tbody>
</table>

We denote by $\hat{\mathcal{I}}_i^{\hat{\mathcal{S}}}$ the tuple formed by deleting from $\hat{\mathcal{I}}_i$ all attributes not in $\hat{\mathcal{I}}_i^{\hat{\mathcal{S}}}$.

Lemma 5.1.4 Definition 5.1.3 determines a unique minimal set $\hat{\mathcal{I}}_i^{\hat{\mathcal{S}}}$.

Proof (sketch). Identical to the proof of Lemma 3.1.15, except that the MVD arguments for cases $\hat{\mathcal{S}}_i = s$ and $\hat{\mathcal{S}}_i = n$ need to be repeated for each $Q_{(G,i)}$ (equation 5.1).

Example 56 Continuing Examples 54 and 55, Query $Q_{55}$ satisfies $\{F\} \models 4$ and $\{D\} \models 3$, and so $\mathcal{S}_{Q_{55}} = \{D, F\}$. The core indexes of $Q_{55}$ with respect to encoding signature $\hat{\mathcal{S}} = \text{snsb}$ are calculated as follows.
$i = 4$: $\hat{\mathcal{I}} = \{b, \hat{\mathcal{I}}_4^\mathcal{F} = \{H, J\}\}$

$c\notin \hat{\mathcal{I}}_4^\mathcal{F}$ because $c \in \mathcal{C}$; $F\notin \hat{\mathcal{I}}_4^\mathcal{F}$ because $F \in \mathcal{I}_3$; $\{H, J\} \subseteq \hat{\mathcal{I}}_4^\mathcal{F}$ in order to satisfy condition $\mathcal{I}_4 = \{c, F, H, J\} \subseteq \hat{\mathcal{I}}_4^\mathcal{F}$.

$i = 3$: $\hat{\mathcal{I}} = \{f, \hat{\mathcal{I}}_3^\mathcal{F} = \{F\}\}$

$B\notin \hat{\mathcal{I}}_3^\mathcal{F}$ because $B \in \mathcal{I}_2$; $F\notin \hat{\mathcal{I}}_3^\mathcal{F}$ because $F \in \mathcal{S}_{Q_{155}} \setminus (\mathcal{I}_{1,2} \cup \hat{\mathcal{I}}_3^\mathcal{F} \cup \mathcal{C})$.

$i = 2$: $\hat{\mathcal{I}}_2 = \{l, n, \hat{\mathcal{I}}_2^\mathcal{F}\}$

$L\notin \hat{\mathcal{I}}_2^\mathcal{F}$ because $L \in \mathcal{V} \setminus (\mathcal{I}_1 \cup \hat{\mathcal{I}}_3^\mathcal{F} \cup \mathcal{C})$; $B\notin \hat{\mathcal{I}}_2^\mathcal{F}$ because $B \in \mathcal{I}_1$; $P\notin \hat{\mathcal{I}}_2^\mathcal{F}$ due to the MVDs $Q_{(123,2)} = aABD \rightarrow LFN$ and $Q_{(123,2)} = aABD \rightarrow LNFHJ$.

while $Q_{(123,2)}$ requires $N\notin \hat{\mathcal{I}}_2^\mathcal{F}$. In order to satisfy the necessary MVD.

\[ Q_{(123,2)}(a, A, B, D, L, N, P, F) = R(A, B), R(B, D), R(B, F), S(D, L), S(L, N), R(B, P) \]
\[ Q_{(123,2)}(a, A, B, D, L, N, P, F, H, J) = R(A, B), R(B, D), R(B, F), S(D, L), S(L, N), R(B, P), S(F, H), S(H, J) \]

\[ i = 1: \hat{\mathcal{I}} = \{b, \hat{\mathcal{I}}_1^\mathcal{F} = \{B, D, L\}\} \]

$a\notin \hat{\mathcal{I}}_1^\mathcal{F}$ because $a \in \mathcal{C}$; $B\notin \hat{\mathcal{I}}_1^\mathcal{F}$ because $B \in \mathcal{V} \setminus (\hat{\mathcal{I}}_1^\mathcal{F} \cup \mathcal{C})$; $D\notin \hat{\mathcal{I}}_1^\mathcal{F}$ because $D \in \mathcal{S}_{Q_{155}} \setminus (\hat{\mathcal{I}}_1^\mathcal{F} \cup \mathcal{C})$; $A\notin \hat{\mathcal{I}}_1^\mathcal{F}$ due to the MVDs $Q_{(12,1)} = BD \rightarrow F$, $Q_{(123,1)} = BD \rightarrow LFN$, $Q_{(124,1)} = BD \rightarrow FHJ$, and $Q_{(123,1)} = BD \rightarrow LNFHJ$.

\[ Q_{(12,1)}(a, A, B, D, F) = R(A, B), R(B, D), R(B, F), S(D, L), S(L, N), R(B, P) \]
\[ Q_{(123,1)}(a, A, B, D, L, N, F) = R(A, B), R(B, D), R(B, F), S(D, L), S(L, N), R(B, P) \]
\[ Q_{(124,1)}(a, A, B, D, F, H, J) = R(A, B), R(B, D), R(B, F), S(F, H), S(H, J) \]
\[ Q_{(1234,1)}(a, A, B, D, L, N, F, H, J) = R(A, B), R(B, D), R(B, F), S(D, L), S(L, N), R(B, P), S(F, H), S(H, J) \]

Given derived DACEQ $Q(\hat{\mathcal{I}}_1; \ldots; \hat{\mathcal{I}}_d; \mathcal{V})$, we convert $Q$ to the $\hat{\mathcal{F}}$-normal form ($\hat{\mathcal{F}}$-NF) by deleting all non-core indexes from the query head. Performing $\hat{\mathcal{F}}$-normalization of derived DACEQs is clearly NP-hard, as Definition 5.1.3 directly generalizes the CEQ case. We can also derive a (possibly loose) upper bound directly from Definition 5.1.3.

**Theorem 5.1.5** Performing $\hat{\mathcal{F}}$-normalization of derived DACEQs is in $\overline{\Sigma}_2^p$.

**Proof.** For each level $i$ and each dag $G \in \text{dages(body}_{Q})$, the MVD associated with query $Q_{(G,i)}$ (equation 5.1) dictates that a certain subset of $\mathcal{I}_i$ needs to be core, and this subset can be identified in NP time (per Theorem 3.1.16). Therefore, verifying each core $I \in \hat{\mathcal{I}}_{1,i}^\mathcal{F}$ is in $\Sigma_2^p$ (by verifying that either $I \in \mathcal{V} \cup \mathcal{S}_Q$ or by guessing the appropriate query $Q_{(G,i)}$ and verifying in NP time that its MVD
Performing $\bar{\Phi}$-normalization of derived DACEQs is in NP.

**Conjecture 5.1.6** Performing $\bar{\Phi}$-normalization of derived DACEQs is in NP.

**Comment.** It would suffice to show

$$\forall v \in \text{body}_Q \left[ Q_{(v,i)} = \mathcal{B}^+ \cap (\mathcal{I}[1,i-1] \sqcup \mathcal{I}^\bar{\phi}) \rightarrow \mathcal{B}^+ \cap \mathcal{I}^\bar{\phi}_{[i+1,d]} \right] \implies$$

$$\forall G \text{ containing } v_i : \left[ Q_{(G,i)} = \mathcal{B}^G \cap (\mathcal{I}[1,i-1] \sqcup \mathcal{I}^\bar{\phi}_G) \rightarrow \mathcal{B}^G \cap \mathcal{I}^\bar{\phi}_{[i+1,d]} \right]$$

and

$$\forall v \in \text{body}_Q \left[ Q_{(v,i)} = \mathcal{B}^+ \cap \mathcal{I}[1,i-1] \rightarrow \mathcal{B}^+ \cap \mathcal{I}^\bar{\phi}_{[i,d]} \right] \implies$$

$$\forall G \text{ containing } v_i : \left[ Q_{(G,i)} = \mathcal{B}^G \cap \mathcal{I}[1,i-1] \rightarrow \mathcal{B}^G \cap \mathcal{I}^\bar{\phi}_{[i,d]} \right]$$

(5.2)

where $Q_{(v,i)}$ with body $\text{body}_Q^+$ is defined analogously to $Q_{(G,i)}$ from equation 5.1. Then, for each level $i$ it is only necessary to examine each query $Q_{(v,i)}$, and for each such query the hypergraph-based algorithm from Theorem 3.1.17 would identify a subset of $\mathcal{T}_i$ that must be core. $\mathcal{T}^\bar{\phi}_i$ would then be the union of these sets.

Unfortunately, due to the intricacy of the various mappings involved, we have been unable to definitively prove (or disprove) equations 5.2 and 5.3.

**Lemma 5.1.7** Given any derived DACEQ $Q(\mathcal{T}_1;\ldots;\mathcal{T}_d;\mathcal{V})$, any index level $i \in [1,d]$, and any node $v \in \text{body}_Q$: if $Q \models \mathcal{I}_i[v] \Rightarrow v$ then $Q \models \mathcal{I}^\bar{\phi}_i[v] \Rightarrow v$.

**Proof.** Definition 5.1.3 implies that $(\mathcal{S}_Q \cap \mathcal{I}_i) \subseteq \mathcal{I}^\bar{\phi}_i$. By Corollary 4.2.7, if $Q \models \mathcal{I}_i[v] \Rightarrow v$ then $Q \models (\mathcal{S}_Q \cap \mathcal{I}_i)[v] \Rightarrow v$.

**Theorem 5.1.8** $\bar{\Phi}$-Normalization preserves $\bar{\Phi}$-equivalence

**Proof.** For every $i \in [0,d]$, let $Q_i$ denote the following HCEQ,

$$Q_i(\mathcal{T}_1;\ldots;\mathcal{T}_i;\mathcal{T}^\bar{\phi}_{i+1};\ldots;\mathcal{T}^\bar{\phi}_d;\mathcal{V}) \Rightarrow \text{body}_Q$$

(5.4)

and assume that $\mathcal{T}_i = \mathcal{T}^\bar{\phi}_i \cdot \mathcal{J}_i$, where $\mathcal{J}_i$ is the set of non-core indexes in $\mathcal{T}_i$. The induction proceeds exactly as in the proof of Theorem 3.1.17. The only complication is that the conditions in Definition 5.1.3 are stated in terms of the query $Q_{(G,i)}$ which is a CQ*, whereas query $Q_i$ has a hierarchical body. (In contrast, in the proof of Theorem 3.1.17, queries $Q_i$ and $Q_i$ had the same body.) We prove here for each case within the inductive step that the conditions on the output of CQ* $Q_{(G,i)}$ in Definition 5.1.3 imply corresponding conditions for the output of HCEQ $Q_i$.

**Case $\bar{\Phi}_i = b$:**

It suffices to show that for any database $\mathcal{D}$, the relation $\Phi := (Q_i)^D$ satisfies the FD $\mathcal{T}^\bar{\phi}_{[1,i-1]} \rightarrow \mathcal{J}_i$; the remainder then follows identically to the proof of Theorem 3.1.17. By Definition 3.1.11, either
1. \( \mathcal{J}_i \in \{\not=\} \), or
2. \( \mathcal{J}_i \in \mathcal{I}_{[1,i-1]} \cup \mathcal{C} \).

In the second case, property \([P7]\) (Observation 5.1.2) implies that all elements of \( \mathcal{J}_i \) are labelled with \( v_i \) satisfying \( Q \models \mathcal{I}_{[1,i-1]} \models v_i \); therefore, in either case the relation \( \Phi \) satisfies the FD \( \hat{\mathcal{I}}_{[1,i-1]} \rightarrow \mathcal{J}_i \).

**Case \( s_i = s \):**

It suffices to show that relation \( \Phi \) satisfies the MVD \( \hat{\mathcal{I}}_{[1,i-1]} \cup \hat{\mathcal{I}}^\top_i \rightarrow \hat{\mathcal{I}}^\top_{[i+1,d]} \cup \hat{\mathcal{V}} \); the remainder then follows identically to the proof of Theorem 3.1.17.

By property \([P7]\), either \( \mathcal{I}_i \not= \{\not=\} \) or all elements of \( \mathcal{I}_i \) are labelled with \( v_i \). If \( \mathcal{I}_i \not= \{\not=\} \) then attributes \( \mathcal{I}_i \) are constant within relation \( \Phi \) and the MVD holds trivially; therefore, assume that some \( v_i \) labels all elements of \( \mathcal{I}_i \).

Let \( \phi_1 : G_1 \rightarrow D \) and \( \phi_2 : G_2 \rightarrow D \) be any two terminal embeddings in \( \Phi \)—of \( G_1, G_2 \in \text{dags(body}_Q\), respectively—that satisfy \( \hat{\phi}_1(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i) = \hat{\phi}_2(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i) \). (Recall from equation 2.22 that \( \hat{\phi} \) denotes the partial application of \( \phi \).) For each \( j \in [1,i] \) such that \( \mathcal{I}_j \not= \{\not=\} \), \( v_j \in G_1 \) iff \( v_j \in G_2 \).

If \( v_i \not\in G_1 \) then \( \hat{\phi}_1(\mathcal{J}_i) = \hat{\phi}_2(\mathcal{J}_i) \) is a constant tuple composed only of the \( \not= \) symbol, and so the MVD holds trivially; therefore, assume that \( v_i \in G_1 \) and \( v_i \in G_2 \).

Define \( G_3 = G_1 \cap G_2 \).\(^1\)

Because

\[
Q_{(G_3,i)} \models B_{G_3}^+ \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^\top) \rightarrow B_{G_3}^+ \cap \mathcal{I}_{[i+1,d]}^\top
\]

and \( \phi_1, \phi_2 \) are embeddings of \( \text{body}_{G_3}^+ \) into \( D \) that agree on \( B_{G_3}^+ \cap (\mathcal{I}_{[1,i-1]} \cap \mathcal{I}_i^\top) \), there exists an embedding \( \phi_3 : G_3 \rightarrow D \) satisfying

- \( \hat{\phi}_3(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i^\top) = \hat{\phi}_2(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i^\top) = \hat{\phi}_1(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i^\top) \),
- \( \hat{\phi}_3(\mathcal{J}_i) = \hat{\phi}_2(\mathcal{J}_i) \) which, by \( \mathcal{I}_{[1,i-1]} \models v_i \), implies \( \hat{\phi}_3(\mathcal{J}_i) = \hat{\phi}_2(\mathcal{J}_i) \), and
- \( \forall j \in [i + 1, d] \), if \( v_j \not\in (G_1 \setminus G_3) \) then \( \hat{\phi}_3(\mathcal{I}_j^\top) = \hat{\phi}_1(\mathcal{I}_j^\top) \).

Now consider the smallest \( j \in [i + 1, d] \) such that \( v_j \in (G_1 \setminus G_3) \), and define \( G_4 := G_3 \cup \{v_j\} \). By property \([P9]\) and Lemma 5.1.7, \( Q \models \mathcal{I}_{[1,i-1]} \cup \mathcal{I}_{[i,j-1]}^\top \models v_j \). By \( \hat{\phi}_3(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i^\top) = \hat{\phi}_1(\mathcal{I}_{[1,i-1]}\hat{\mathcal{I}}_i^\top) \) and the definition of nullability dependencies (Definition 4.2.2), \( \phi_3 \) must extend to \( G_4 \). We now iterate the above argument, appealing to the following MVD

\[
Q_{(G_4,i)} \models B_{G_4}^+ \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^\top) \rightarrow B_{G_4}^+ \cap \mathcal{I}_{[i+1,d]}^\top
\]

and to \( \phi_1, \phi_3 \) being embeddings of \( \text{body}_{G_4}^+ \) into \( D \) agreeing on \( B_{G_4}^+ \cap (\mathcal{I}_{[1,i-1]} \cap \mathcal{I}_i^\top) \) to deduce the existence of some embedding \( \phi_4 : G_4 \rightarrow D \). By induction, we eventually arrive at some terminal embedding \( \phi_k : G_1 \rightarrow D \) satisfying

\(^1\)We are abusing notation by using \( G \) to refer to both a DAG within \( \text{dags(body}_Q\) and the set of node labels occurring within that DAG.
• \( \hat{\phi}_k(\overline{T}_{[1,i-1]}^{\overline{x}_i}) = \hat{\phi}_2(\overline{T}_{[1,i-1]}^{\overline{x}_i}) = \hat{\phi}_1(\overline{T}_{[1,i-1]}^{\overline{x}_i}) \),
• \( \hat{\phi}_k(\overline{J}_i) = \hat{\phi}_2(\overline{J}_i) \), and
• \( \forall j \in [i+1,d], \hat{\phi}_k(\overline{T}_{[i+1,d]}^{\overline{x}_i}) = \hat{\phi}_1(\overline{T}_{[i+1,d]}^{\overline{x}_i}) \).

Embedding \( \hat{\phi}_k \) proves that that relation \( \Phi \) satisfies the desired MVD.

Case \( \hat{s}_i = n \):

It suffices to show that relation \( \Phi \) satisfies the MVD \( \hat{\mathcal{T}}_{[1,i-1]} \rightarrow \hat{\mathcal{T}}_{[1,d]} \cup \hat{\mathcal{Y}} \).

The proof is identical to case \( \hat{s}_i = s \), with appropriate modifications to the variables in the MVD. The remainder then follows identically to the proof of Theorem 3.1.17.

Example 57 Continuing Example 56, query \( Q_{57} \) in Figure 5.2a is the snsb-normal form of \( Q_{55} \) and therefore \( Q_{57} \equiv_{\text{snb}} Q_{55} \equiv_{\text{snb}} Q_{20} \). Query \( Q_{58} \) in Figure 5.2b is the snsb-normal form of \( Q_{56} \) and therefore \( Q_{58} \equiv_{\text{snb}} Q_{56} \equiv_{\text{snb}} Q_{21} \). Figure 5.2c shows the encoding relation \( R_{16} = (Q_{57})^{P_3} \), which can be obtained by projecting away the non-core index attributes from encoding relation \( R_8 = (Q_{20})^{P_3} \) (cf. Figure 2.20). Figure 5.2d shows the encoding relation \( R_{17} = (Q_{58})^{P_3} \), which can be obtained by projecting away the non-core index attributes from encoding relation \( R_9 = (Q_{21})^{P_3} \) (cf. Figure 2.21); observe that \( R_9 \) has the identical instance as \( R_8 \). In the next section (Examples 58 and 59) we will prove that over any database, the pair of encoding relations yielded by queries \( Q_{57} \) and \( Q_{58} \) will have identical instances.

5.2 Encoding-Equivalence of Derived DACEQs

In this section we partially characterize \( \hat{\mathcal{F}} \)-equivalence between derived DACEQs. We start by deriving a necessary condition that relates \( \hat{\mathcal{F}} \)-equivalence to DACQ equivalence, whose corollary is a necessary isomorphic condition between DACEQ query bodies analogous to the head graph isomorphisms of Section 4.3.1. We then extend the definition of index-covering homomorphisms (see Chapter 3) to derived DACEQs, and show that the existence of index-covering homomorphisms between the \( \hat{\mathcal{F}} \)-normal forms constitutes a sufficient condition for \( \hat{\mathcal{F}} \)-equivalence.

Definition 5.2.1 (Structural DACQ) Given derived DACEQ \( Q(\overline{X}_1; \ldots; \overline{X}_d; \overline{V}) \), for each \( i \in [1,d] \) define the attribute \( X_i \) as follows.

\[
X_i := \begin{cases} 
\hat{\mathcal{F}}_i & \text{if } \overline{\mathcal{F}}_i \notin \{\hat{\mathcal{F}}\} \\
0 & \text{otherwise (} v_i \text{ defined as in property [P7] of Observation 5.1.2)}
\end{cases}
\]

We define the structural query \( \text{STRUCTURAL}(Q) \) as the DACQ with the same body as \( Q \) and with the head tuple \( \overline{W} := X_{[1,d]} \overline{V} \).
Figure 5.2: The snsb-normal forms of Q55 and Q56 and their results over D₃
Theorem 5.2.2  Given any two derived DACEQs $Q$ and $Q'$ and any encoding signature $\bar{\gamma}$, if $Q \equiv_{\bar{\gamma}} Q'$ then $\text{STRUCTURAL}(Q) \equiv \text{STRUCTURAL}(Q')$.

Proof. Given any database $D$, let $\Gamma$ denote the set of terminal embeddings of $\text{body}_Q$ into $D$, and let $\Phi$ be the analogous set for $\text{body}_{Q'}$. Suppose that $Q \equiv_{\bar{\gamma}} Q'$; then, there must exist some $\bar{\gamma}$-certificate $C$ proving $(Q)^D \equiv_{\bar{\gamma}} (Q')^D$. Consider any $\gamma \in \Gamma$—then, there must exist some $\phi \in \Phi$ such that certificate $C$ contains a tuple node $n^\epsilon$ that proves $\hat{\gamma}(\bar{\nu}) = \hat{\phi}(\bar{\nu})$. By Definitions 2.2.5–2.2.7, the certificate node at any level $i \in [1,d]$ along the path to $n^\epsilon$ must satisfy $\hat{\gamma}(\bar{t}_i) = \bar{t}_i \iff \hat{\phi}(\bar{t}_i) = \bar{t}_i$. It follows that $\hat{\gamma}(X_{[1,d]}^{\bar{\nu}}) = \hat{\phi}(X_{[1,d]}^{\bar{\nu}})$, which proves $\text{STRUCTURAL}(Q) \equiv \text{STRUCTURAL}(Q')$. Repeating the argument in the other direction proves $\text{STRUCTURAL}(Q') \equiv \text{STRUCTURAL}(Q)$. \hfill $\Box$

Corollary 5.2.3 If $Q \equiv_{\bar{\gamma}} Q'$ then $\text{STRUCTURAL}(Q)$ and $\text{STRUCTURAL}(Q')$ are head graph isomorphic.

Proof. Theorem 4.3.4. \hfill $\Box$

Corollary 5.2.4 Given any derived DACEQ $Q$ and any signature $\bar{\gamma}$, the $\bar{\gamma}$-normal form of $Q$ contains within its head the label of every node in $\text{body}_Q$.

Proof. Property [P10] of Observation 5.1.2 implies that the label of every node in $\text{body}_Q$ appears within the head graph of $\text{STRUCTURAL}(Q)$, and so the claim follows by Theorem 5.1.8 and Corollary 5.2.3. \hfill $\Box$

Our previous definition of index-covering homomorphisms between CEQs (Definition 3.2.1) was a simple generalization of homomorphisms between CQs (Definition 3.1.1). We now extend the definition of index-covering homomorphisms to handle DACEQs. Unlike CEQ bodies, DACEQ bodies may contain equality predicates; therefore, it is helpful to first review how the definition of CQ homomorphisms can be extended to handle CQ’s.

Definition 5.2.5 (CQ* Homomorphism) Given two satisfiable CQ* $Q(\bar{W})$ and $Q'(\bar{W}')$, a homomorphism $h$ from $Q'$ to $Q$ is a mapping $h : \mathcal{B}' \to \mathcal{B} \cup \mathcal{C}$ satisfying

1. $\text{body}_Q \models h(\text{body}_{Q'})$; in other words,
   (a) $\text{pred}_Q \models h(\text{pred}_{Q'})$, and
   (b) $\forall R(\bar{X}') \in \text{atoms}_{Q'} : \exists R(\bar{X}) \in \text{atoms}_Q : [\text{pred}_Q \models h(\bar{X}') = \bar{X}]$;

2. for each attribute $W'_i \in \bar{W}' : \text{pred}_Q \models h(W'_i) = W_i$, where $W_i$ is the corresponding attribute in $\bar{W}$.

The reader is invited to verify that in the restricted case where the queries do not contain equality predicates, Definition 5.2.5 is equivalent to Definition 3.1.1.

Definition 5.2.6 (Index-Covering Homomorphism) Given any two DACEQs $Q(\bar{\mathcal{T}}_1; \ldots ; \bar{\mathcal{T}}_d; \bar{\nu})$ and $Q'(\bar{\mathcal{T}}'_1; \ldots ; \bar{\mathcal{T}}'_d; \bar{\nu}')$ such that $\text{STRUCTURAL}(Q')$ and $\text{STRUCTURAL}(Q)$...
are head graph isomorphic, for each node \( u \in \text{body}_Q \) let \( u' \) denote the corresponding node in \( \text{body}_{Q'} \), and vice-versa. Then, an index-covering homomorphism from \( Q' \) to \( Q \) is a mapping \( h : \mathcal{B}' \to \mathcal{B} \cup \mathcal{C} \) satisfying

1. for each node \( u' \in \text{body}_{Q'} \), \( \text{body}_u^+ \models \text{body}_{u'} \); in other words,
   (a) \( \text{pred}_u^+ \models h(\text{pred}_{u'}) \), and
   (b) \( \forall R(X) \in \text{atoms}_{u'} \exists R(X) \in \text{atoms}_u^+ \left[ \text{pred}_u^+ \models h(X) = X \right] ;
2. for each attribute \( V_i' \in \mathcal{V}' \) in \( \mathcal{V}' \):
   \( \text{pred}_u^+ \models h(V_i') = V_i \), where \( V_i' \) is the corresponding attribute in \( \mathcal{V}' \); and
3. \( \forall i \in [1,d]: \tilde{T}_i \subseteq h(\tilde{T}_i) \).

Again, the reader is invited to verify that in the restricted case where the query bodies contain only a single node (i.e., both \( Q \) and \( Q' \) are CEQs), Definition 5.2.6 is equivalent to Definition 3.2.1.

**Example 58** The following mappings \( h : Q_{57} \to Q_{58} \) and \( h' : Q_{58} \to Q_{57} \) between the queries in Figure 5.2 are both index-covering homomorphisms.

\[
\begin{align*}
h &= \{A/A', B/B', D/D', F/F', L'/L', N/N', P/P', H/H', J/J'\} \\
h' &= \{A'/A, B'/B, D'/D, F'/F, L'/L, N'/N, H'/H, J'/J\}
\end{align*}
\]

**Theorem 5.2.7** Two derived DACEQs are \( \hat{\xi} \)-equivalent if there exists index-covering homomorphisms in both directions between their \( \hat{\xi} \)-normal forms.

**Proof.** Let \( Q(\tilde{T}_1; \ldots ; \tilde{T}_d; \tilde{\mathcal{V}}) \) and \( Q'(\tilde{T}_1; \ldots ; \tilde{T}_d; \tilde{\mathcal{V}}') \) be the \( \hat{\xi} \)-normal forms of the given queries, and let \( h : Q \to Q \) and \( h' : Q \to Q' \) be index-covering homomorphisms between them. By Theorem 5.1.8, the given queries are \( \hat{\xi} \)-equivalent iff \( Q \equiv_{\hat{\xi}} Q' \). The existence of \( h \) and \( h' \) implies by definition that there exists a head graph isomorphism \( \theta \) from \( Q \) to \( Q' \), and by Corollary 5.2.4 \( \theta \) must be a bijection between the nodes in \( \text{body}_Q \) and \( \text{body}_{Q'} \).

By Definition 5.2.6, composite mapping \( h \circ h' \) satisfies \( \forall i \in [1,d]. \left[ h \circ h'(\tilde{T}_i) = \tilde{T}_i \right] \). Therefore, we can assume w.l.o.g. that \( h \circ h' \) satisfies the following property

\[
\forall i \in [1,d]. \left[ h \circ h'(\tilde{T}_i) = \tilde{T}_i \right] \tag{5.5}
\]

(by the finite period of any permutation of tuple \( \tilde{T}_{[1,d]} \)).

Consider any database \( D \) and any terminal embedding \( \gamma : G \to D \) of any \( G \in \text{dags}(\text{body}_Q) \); then, \( \gamma \circ h \) is an embedding of \( \theta(G) \in \text{dags}(\text{body}_{Q'}) \) into \( D \). Definition 5.1.3 implies \( S_Q \subseteq \tilde{T}_{[1,d]} \), and equation 5.5 implies \( \gamma \circ h \circ h'(\tilde{T}_{[1,d]}) = \gamma(\tilde{T}_{[1,d]}) \); therefore, the definition of \( S_Q \) (see Corollary 4.2.7) mandates that \( \gamma \circ h \circ h' \) is a terminal embedding. Hence, embedding \( \gamma \circ h \) must also be be a terminal embedding of \( \theta(G) \).

Repeating this argument in the other direction proves that encoding relations \( (Q)^D \) and \( (Q')^D \) differ only by permutation of attributes within each index level, and so it is trivial to construct a \( \hat{\xi} \)-certificate proving \( (Q)^D \equiv_{\hat{\xi}} (Q')^D \). By our unrestricted choice of \( D \), \( Q \equiv_{\hat{\xi}} Q' \) follows. \( \square \)
Example 59 The homomorphisms in Example 58 prove that over any database, the encoding relations yielded by \( Q_{57} \) and \( Q_{58} \) will have identical\(^2\) instances, and hence \( Q_{57} \equiv_{\text{ssnsb}} Q_{58} \). We conclude that \( Q_{20} \equiv_{\text{ssnsb}} Q_{21} \) \footnote{In general, the existence of index-covering homomorphisms in both directions only implies only that query results are identical \textit{modulo permutation of attributes within each index level}. However, the index-covering homomorphisms \( h \) and \( h' \) from Example 58 have the additional property that for each index level \( i \), \( h \) and \( h' \) form an isomorphism between the pair-wise components of the two index tuples. This justifies the stronger claim that the query results are identical, without any permutation of attributes.} (see Example 57) and therefore \( Q_{18} \equiv Q_{19} \) (see Example 23).

5.3 Food for Thought

In this section we show two examples that provide further insight into the equivalence results of the last section. In particular, the examples illustrate that the condition in Theorem 5.2.7—the existence of index-covering homomorphisms in both directions between the \( \overline{\mathbb{F}} \)-normal forms—is not necessary for \( \overline{\mathbb{F}} \)-equivalence between derived DACEQs.

The first example illustrates a shortcoming in our characterization of core indexes. Ideally, the set of core indexes should be minimal in the sense that no strict subset would preserve \( \overline{\mathbb{F}} \)-equivalence. Our definition of core indexes for CEQs in Chapter 3 satisfies this minimality property, as evidenced by the proof of Theorem 3.2.4 (see Appendix A). This example demonstrates that our definition of core indexes for derived DACEQs (Definition 5.1.3) does not share this property.

Example 60 Consider the three HCEQs shown in Figure 5.3a, all sharing a query body that is already in DACQ canonical form. Consider the \( \text{ssns} \)-normalization of \( Q_{59} \) (using Definition 5.1.3).

\[
i = 3: \quad \frac{s_3 = s, T_3}{\overline{\mathbb{F}}} = \{\}
\]
\[
C \in \not T_3 \text{ because } C \in \mathcal{I}_2.
\]

\[
i = 2: \quad \frac{s_2 = s, T_2}{\overline{\mathbb{F}}} = \{A[t_2], C[t_2]\}
\]
\[
C \in \not T_2 \text{ because } C \in \mathcal{V} \setminus (\mathcal{I}_1 \cup T_3 \cup C); \text{ meanwhile, } A \in \not T_2 \text{ because } Q_{(u, 2)} \models BA \rightarrow C \text{ and } Q_{(uv, 2)} \models BA \rightarrow C \text{ (both trivially), but } Q_{(u, 2)} \equiv B \rightarrow C.
\]

\[
Q_{(u, 2)}(B, A, C) := R(A, B), S(B, C), R(A, D), S(D, C), T(D)
\]
\[
Q_{(uv, 2)}(B, A, C) := R(A, B), S(B, C), R(A, D), S(D, C), T(D), T(B)
\]

\[
i = 1: \quad \frac{s_1 = s, T_1}{\overline{\mathbb{F}}} = \{B[t_1]\}
\]
\[
B \in \not T_1 \text{ because } S_{Q_{59}} = \{B\}.
\]

Therefore, \( Q_{60} \) is the \( \text{ssns} \)-NF of \( Q_{59} \). However, we can show that \( Q_{60} \equiv_{\text{ssns}} Q_{61} \), implying that \( A \) should not need to be a core index of \( Q_{59} \) to preserve \( \text{ss} \)-equivalence.
Suppose that we replace the attribute $C$ with the constant 0 within the head of $Q_{59}$. Within this new query, variable $C$ would not be an output variable, and so the trivial MVDs $Q_{(u,2)} \vdash B \rightarrow \emptyset$ and $Q_{(w,2)} \vdash B \rightarrow \emptyset$ would justify $\hat{\mathcal{F}} = \emptyset$ (i.e., neither $A$ nor $C$ would be core indexes). Therefore, the assessment that both $A$ and $C$ are core within $Q_{59}$ (as per Definition 5.1.3) is directly attributable to head attribute $C$.

Now reconsider CQ $Q_{(w,2)}$ shown above; $Q_{(w,2)}$ is not a minimal CQ because it allows the automorphism $h = \{A/A, B/B, C/C, D/B\}$. After using $h$ to minimize the query body, it becomes clear that $Q_{(w,2)}$ satisfies the MVD $B \rightarrow C$. In contrast, $h$ is not an automorphism for $Q_{(u,2)}$, and $Q_{(u,2)} \not\vdash B \rightarrow C$. As a concrete example, consider the database instance $D_9$ shown in Figure 5.3b and the set of terminal embeddings $\Gamma = \{\gamma : Q_{59} \sim D_9\}$ shown in Figure 5.3c. If we project "relation" $\Gamma$ onto attributes $ABC$, it violates the MVD $B \rightarrow C$; however, if we first restrict $\Gamma$ to only embeddings that extend to $v$, then MVD $B \rightarrow C$ is satisfied.

In summary, index $C$ needs to be core in order to functionally determine the value of the output attribute $C$, but only for embeddings that extend to node $v$. Index $A$ was deemed core because in general it is not separated from core index $C$ by an MVD, but the subset of embeddings that extend to node $v$—i.e., the embeddings that require $C$ to be core—satisfy an MVD separating $A$ from $C$ (or more precisely, $A$ from $C$).

It is likely that the shortcomings of Definition 5.1.3 illustrated by Example 60 could be overcome by a more sophisticated characterization of core indexes. One possibility would be to devise a system of bookkeeping that associates with each core index a set of reasons justifying why that index needs to be core. The reasons associated with known core indexes would then need to factor into later decisions identifying further core indexes.

In the next example we use HCEQ encoding-equivalence to decide equivalence between a pair of HCQs composed of a single edge between two nodes. The two HCQs in this example have been carefully crafted to illustrate why index-covering homomorphisms are not sufficient for characterizing HCEQ encoding-equivalence.

Example 61 Consider the two HCQs $Q_{62}$ and $Q_{63}$ shown in Figure 5.4. Using the results of Section 4.3, we can show $Q_{62} \equiv Q_{63}$ as follows.

Both queries have canonical tuples $\{0, 0\}$. The tuple-specific equivalence $Q_{62} \equiv Q_{63}$ follows as a result of the following two homomorphisms.

$$h = \{A/W, B/X, C/Y, D/Z, E/W, F/X, B_1/X_1, \ldots, B_n/X_n\}$$

$$h' = \{W/A, X/B, Y/A, Z/B, X_1/B_1, \ldots, X_n/B_n, Z_1/B_1, \ldots, Z_n/B_n\}$$

To prove $Q_{62} \equiv Q_{63}$ requires proving $\Sigma(Q_{62}; 0) \equiv_{\text{fin}} \Sigma(Q_{63}; 0)$, where $\Sigma(Q_{62}; 0)$ and
(a) HCEQs $Q_{59}$, $Q_{60}$, and $Q_{61}$

(b) Database $D_9$

(c) Terminal embeddings of $Q_{59}$ into $D_9$

Figure 5.3: Encoding queries $Q_{59}$, $Q_{60}$, and $Q_{61}$ evaluated over database $D_9$.

Figure 5.4: HCQs $Q_{62}$ and $Q_{63}$
\( \Sigma(Q_{62}(\mathbb{M})) \) are defined as follows.

\[
\Sigma(Q_{62}(\mathbb{M})) = \{ \xi(u,\mathbb{M}) : \text{body}_u \rightarrow \text{body}_v \}\n\]
\[
\Sigma(Q_{63}(\mathbb{M})) = \{ \xi(a,\mathbb{M}) : \text{body}_1 \rightarrow \text{body}_2 \}\n\]

The implication \( \Sigma(Q_{62}(\mathbb{M})) \models_{\text{fin}} \Sigma(Q_{63}(\mathbb{M})) \) can be proven by the chase procedure using two chase steps:

1. chase \( \xi(u,\mathbb{M}) \) with binding \( X/b, Z/* \) to generate the chains
   \( S(b, b), S(b_1, b_1), \ldots, S(b_n, b_n) \) and \( R(b, b_1), \ldots, R(b_{n-1}, b_n) \);
2. chase \( \xi(a,\mathbb{M}) \) with binding \( X/f, Z/d \) to generate the atom \( S(d, f) \).

The implication \( \Sigma(Q_{63}(\mathbb{M})) \models_{\text{fin}} \Sigma(Q_{62}(\mathbb{M})) \) can be proven by the chase procedure using \( n + 2 \) chase steps:

1. chase \( \xi(1,\mathbb{M}) \) with binding \( B/x, D/z, F/x \) to generate the atom \( S(z, x) \) and the chains \( S(x, x), S(x_1, x_1), \ldots, S(x_n, x_n) \) and \( R(x, x_1), \ldots, R(x_{n-1}, x_n) \);
2. chase \( \xi(0,\mathbb{M}) \) with binding \( B/z, D/*, F/* \) to generate the chains
   \( S(z, z), S(z_1, z_1), \ldots, S(z_n, z_n) \) and \( R(z, z_1), \ldots, R(z_{n-1}, z_n) \);
3. for each \( i \in [1, n] \), chase \( \xi(i,\mathbb{M}) \) with binding \( B/*, D/z_i, F/x_i \) to generate the atom \( S(z_i, x_i) \).

Now consider the following HCEQ \( Q_{64} \).

\[
Q_{64}(W, X, Y, Z, X_1, \ldots, X_n, Z_1, \ldots, Z_n, 0) :\text{body}_{Q_{62}}
\]

Theorem 4.2.4 implies that \( S_{Q_{64}} = \{ X, Z \} \) is the minimal set satisfying \( Q_{64} \models S_{Q_{64}} \equiv v \), and so ss-normalization of \( Q_{64} \) yields the following query \( Q_{65} \).

\[
Q_{65}(X, Z, \vdots, 0) :\text{body}_{Q_{62}}
\]

It straightforward to verify that over any database \( D \), the ss-decoding of \( (Q_{65})^D \) yields one of four possible values:

1. \( \{ 0 \} \) if and only if \( (Q_{62})^D = \{ 0 \} \),
2. \( \{ 0 \} \) if and only if \( (Q_{62})^D = \{ 0 \} \),
3. \( \{ \mathbb{M} \} \) if and only if \( (Q_{62})^D = \{ \mathbb{M} \} \), or
4. \( \{ \} \) if and only if \( (Q_{62})^D = \{ \} \).

Repeating the above argument for \( Q_{63} \), we define \( Q_{66} \)

\[
Q_{66}(A, B, C, D, E, F, B_1, \ldots, B_n, 0) :\text{body}_{Q_{63}}
\]

which has \( S_{Q_{66}} = \{ B, D, F \} \) and ss-normal form \( Q_{67} \).

\[
Q_{67}(B, D, F, \vdots, 0) :\text{body}_{Q_{63}}
\]
Due to correspondence between the outputs of $Q_{67}$ and $Q_{63}$, we conclude the following.

$$Q_{65} \equiv_{as} Q_{67} \iff Q_{62} \equiv Q_{63}$$

We have already shown that $Q_{62} \equiv Q_{63}$, and therefore $Q_{65} \equiv_{as} Q_{67}$. Clearly index-covering homomorphisms can not exist in both directions between $Q_{65}$ and $Q_{67}$, since they have different numbers of core indexes. Furthermore, this phenomenon is not due to a failure to minimize the core indexes—as in Example 60—because for both $Q_{65}$ and $Q_{67}$ the core indexes are already minimal (in the sense that no strict subset preserves $as$-equality).

The reduction in Example 61 can be repeated for any pair of two-node HCQs. Consequently, fully solving the HCEQ encoding-equivalence problem necessitates first solving the HCQ equivalence problem (at least for the restricted case of two-node HCQs). Unfortunately, our results from Chapter 4 where we considered HCQ/DACQ equivalence do not guarantee a decidable equivalence condition even for the simple class of two-node HCQs.
Chapter 6

Conclusions

In Section 6.1 we summarize the main results of this thesis. In Section 6.2 we revisit the motivating examples introduced in Chapter 1 and use them to illustrate both concrete applications of our results and shortcomings in the theory which motivate some of the avenues of future work we discuss in Section 6.3.

6.1 Summary of Results

To the best of our knowledge, this thesis is the only systematic study of the query equivalence problem for conjunctive queries extended with nested aggregation. Our study yielded the following results:

1. **A novel encoding of complex objects within a single flat relation.**  
   Most previous encodings of complex objects as flat relations utilize multiple relations, with the existence of a collection stored separately from the values of its elements. Our definition of object “linearization” and subsequent use of null values to encode empty sub-collections highlights an interesting connection between scalar aggregation (i.e., aggregation operators capable of aggregating over empty inputs) and the left outer-join operator (see Chapter 2).

2. **A normal form for encoding queries that illuminates the interactions between nested aggregation operations.**  
   In particular, our characterization of the core indexes of an encoding query in terms of query-implied multi-valued dependencies reveals how the joins within the query body interact with nested aggregation (see Chapters 3 and 5).

3. **A complete characterization of equivalence between conjunctive queries that construct objects without empty sub-collections.**  
   For the special case of object-constructing queries that do not construct empty sub-collections, deciding query equivalence reduces to deciding encoding-equivalence...
between standard conjunctive queries. We prove that deciding this encoding-equivalence condition is NP-complete and corresponds to a homomorphic condition between the normal forms of the encoding queries (see Chapter 3 and Appendix A).

4. **Necessary conditions and sufficient conditions for equivalence between conjunctive queries with outer joins.**

To the best of our knowledge, this thesis is the first work addressing the query equivalence problem for queries containing the left outer-join operator. We do not completely solve this problem; however, we do introduce a helpful high-level representation of these queries as DAGs whose nodes are (rule-based) conjunctive queries. This variable-based (rather than algebraic) syntax facilitates converting the query bodies to a canonical form, over which we are able to state several necessary conditions and sufficient conditions for query equivalence in terms of homomorphisms between query bodies (see Chapter 4).

5. **Necessary conditions and sufficient conditions for equivalence between conjunctive queries that construct arbitrary objects.**

These conditions are stated in terms of homomorphisms between normal forms of the associated encoding queries, and could be used to develop more powerful logical rewriting rules for relational query optimizers. We also show that query equivalence in the presence of nested aggregation is at least as hard as query equivalence in the presence of left outer joins (see Chapter 5).

### 6.2 Comprehensive Examples

Recall the motivating examples from Chapter 1. We now show that, without the primary and foreign key constraints shown in Figure 1.1, the logical rewriting we suggest within Example 1 does not preserve equivalence.

**Example 62** Consider queries $Q_1$ and $Q_3$ from Example 1. If we model the outputs of the SQL aggregation functions `SUM` and `AVG` as bags and normalized bags, respectively, then $Q_1$ and $Q_3$ translate into the COCQL\(^{(U, F, F)}\) queries $Q_{68}$ and $Q_{69}$, shown in Figures 6.1 and 6.2. The translation of $Q_1$ makes use of a simple transformation from an aggregation block with $k$ aggregation expressions into a join of $k$ such blocks, each with a single aggregation expression. (For space reasons, we use a hybrid notation in Figures 6.1 and 6.2 that replaces blocks of joins between base tables with an equivalent rule-based notation. We have used frames to indicate the logical expansion of view definitions, with thin frames denoting logical views—which are logically expanded during query evaluation—and thick frames denoting materialized views—which are not logically expanded during query evaluation.)

Figure 6.3 depicts the output sort $\tau_{14}$ of $Q_{68}$ and $Q_{69}$, as well as its transformation to the chain sort $(b \bowtie b, 6)$. Figure 6.4 illustrates the CEQs $Q_{70} := \text{ENCODE}(Q_{68})$ and
$Q_{71} := \text{Encode}(Q_{69})$. (Components of the queries have been labelled for the sake of clarity.) Converting query $Q_{70}$ to $\text{bnbnb}$-NF removes the shaded indexes from $\bar{I}_4$ and $\bar{I}_2$. Query $Q_{71}$ is already in $\text{bnbnb}$-NF. Clearly, index-covering homomorphisms cannot exist in both directions between the normalized versions of $Q_{70}$ and $Q_{71}$, because the sizes of the index sets differ. Therefore, we conclude $Q_{70} \not\equiv_{\text{bnbnb}} Q_{71}$, which implies $Q_{68} \not\equiv Q_{69}$, which further implies that the SQL queries $Q_1$ and $Q_3$ are not equivalent over unrestricted database instances.\footnote{Cohen et al. prove that queries performing top-level application of \textit{sum} are equivalent if and only if the relational inputs to the \textit{sum} function are always equivalent under bag semantics [25]. Grumbach et al. prove a similar result relating \textit{avg} to normalized bags [50].}

Our characterization of CEQ encoding-equivalence in terms of homomorphisms is a direct generalization of the traditional characterization of CQ equivalence. Because of this, it is straightforward to adapt techniques for testing CQ equivalence with respect to a set $\Sigma$ of schema constraints (denoted $Q \equiv^\Sigma Q'$) to CEQ encoding-equivalence. Assuming that $\Sigma$ contains only functional dependencies (FDs) and acyclic inclusion dependencies (INDs)—which generalize primary keys and acyclic foreign keys—we can decide \textit{encoding-equivalence} \textit{w.r.t.} $\Sigma$ as follows:

1. Prior to the conversion to $\text{\S}$-normal form, we pre-process each CEQ by first applying the standard $\Sigma$-chase, as defined over CQs [6, Ch. 8–9]. (Chasing a query with an FD causes two variables to be merged, while chasing with an INDs introduces a new atom into the query body.)
2. After the queries have been chased, we use only the FDs in $\Sigma$ to expand out the index sets in the query head. For each $i \in [1, d]$ starting with $i = 1$, we expand $\bar{I}_i$ with all variables functionally determined by $\bar{I}_{[1,i]}$, deleting variables from inner index sets whenever they are added to outer index sets.
3. The conversion to $\text{\S}$-NF is unchanged, but the testing of the query-implied MVDs when identifying core indexes (Definition 3.1.11) may need to account for the FDs in $\Sigma$. (More specifically, if query-implied MVDs are tested via the hypergraph condition of Theorem 3.1.10, then no changes are necessary; however, if query-implied MVDs are tested via the query equivalence condition of Theorem 3.1.3, then that condition needs to use $\equiv^\Sigma$ instead.)

We now repeat Example 62, showing that the logical rewriting we suggest within Example 1 preserves equivalence with respect to database instances that conform to the schema constraints.

Example 63 Reconsider queries $Q_{70}$ and $Q_{71}$ in Figure 6.4. Let $\Sigma$ denote the set of primary and foreign key constraints shown in Figure 1.1. Chasing the body of $Q_{70}$ with $\Sigma$ does not introduce any new atoms, but it does merge the variables $N, N_2, N_4$. After expanding the index sets in the head of $Q_{70}$, we obtain the new query $Q_{72} := \text{chase}_{\Sigma}(Q_{70})$ shown in Figure 6.4. Shaded attributes again indicate redundant index columns that get removed by $\text{bnbnb}$-normalization. Query $Q_{71}$ is unchanged after chasing with $\Sigma$.\footnote{Cohen et al. prove that queries performing top-level application of \textit{sum} are equivalent if and only if the relational inputs to the \textit{sum} function are always equivalent under bag semantics [25]. Grumbach et al. prove a similar result relating \textit{avg} to normalized bags [50].}
Figure 6.1: O(\text{OCQL}) translation of SQL query

\[
(\emptyset \mathcal{I}_f I^T Z^m I^N_{\text{det}}) \text{VAL} E = \emptyset
\]
Figure 6.2: \( Q_{69} \), the \( \text{COCQL}\{f_b,f_s,f_n\} \) translation of SQL query \( Q_3 \)

Figure 6.3: Output sort \( \tau_{14} \) of \( Q_{68} \) and \( Q_{69} \), and its transformation to a chain
The following two mappings \( h : Q_{72} \to Q_{71} \) and \( h' : Q_{71} \to Q_{72} \) are index-covering homomorphisms between the bnb-b-normal forms of \( Q_{72} \) and \( Q_{71} \).

\[
\begin{align*}
  h := &\{A'\l A', N'\l N', R\l R', \cr &C_1/C_1', M_1/M_1', O_1/O_1', D_1/D_1', L_1/L_1', P_1/P_1', Y_1/Y_1', \cr &C_2/C_2', M_2/M_2', O_2/O_2', D_2/D_2', L_2/L_2', P_2/P_2', Y_2/Y_2', \cr &C_3/C_3', M_3/M_3', O_3/O_3', D_3/D_3', L_3/L_3', P_3/P_3', Y_3/Y_3', \cr &C_4/C_4', M_4/M_4', O_4/O_4', D_4/D_4', L_4/L_4', P_4/P_4', Y_4/Y_4' \} \\
  h' := &\{A'\l A, N'\l N, R\l R, \cr &C_1'/C_1, M_1'/M_1, O_1'/O_1, D_1'/D_1, L_1'/L_1, P_1'/P_1, Y_1'/Y_1, \cr &C_2'/C_4, M_2'/M_4, O_2'/O_4, D_2'/D_4, L_2'/L_4, P_2'/P_2, Y_2/Y_4 \}
\end{align*}
\]

Homomorphisms \( h \) and \( h' \) together prove \( Q_{72} \equiv_{\Sigma}^{\Sigma_{bnb}} Q_{71} \). Because \( Q_{72} := \text{chase}_{\Sigma}(Q_{70}) \) implies \( Q_{72} \equiv_{\Sigma}^{\Sigma_{bnb}} Q_{70} \) for any compatible signature \( \Sigma \), this entails \( Q_{70} \equiv_{\Sigma_{bnb}}^{\Sigma} Q_{71} \) and therefore \( Q_{68} \equiv_{\Sigma}^{\Sigma_{09}} Q_{69} \). It follows that the SQL queries \( Q_1 \) and \( Q_3 \) are equivalent over all database instances conforming to the schema constraints in \( \Sigma \).

We now turn our attention to Example 2 and show how the techniques in our thesis allow for simplification of the nested scalar aggregation in query \( Q_4 \).

Example 64 Consider queries \( Q_4 \) and \( Q_6 \) from Example 2. If we model the output of aggregation function \( \text{MAX} \) as a set, and the output of \( \text{COUNT}(\ast) \) as a bag of zero-ary tuples, then \( Q_4 \) and \( Q_6 \) translate into the \( \text{COCQL}\{h_s,f_s,h_k\} \) queries \( Q_{73} \) and \( Q_{74} \) shown in Figure 6.5 (thin and thick frames denote the expansions of logical and materialized views, respectively). Queries \( Q_{73} \) and \( Q_{74} \) both have output sort \( \{[\{\text{dom}\}, \{\{\text{dom}\}\}]\} \), which we linearize into the chain sort \( \text{bnb}\)-NF.

The encoding query \( Q_{75} := \text{ENCODE}(Q_{73}) \) is shown in Figure 6.6; observe that \( Q_{75} \) is a hierarchical CEQ due to the scalar aggregation operator in \( Q_4 \). As described in Chapter 5, before converting \( Q_{75} \) into \( \text{bsb}\)-normal form we must first convert its body into DACQ canonical form (see Chapter 4). Because there exists a homomorphism \( h_b : \text{body}^+_b \to \text{body}^+_a \) that is the identity over \( B^+_a \),

\[
h_b = \{C/C, R\l R, O_1/O_1, D_1/D_1, M_1/M_1, O_2/O_2, D_2/D_2 \}
\]

the conversion to DACQ canonical form merges node \( \square \) into \( \square \). This yields the CEQ \( Q_{76} \) shown in Figure 6.6; note that because the body does not contain any hierarchical edges, we can drop the node labels from the query head. We have shaded the attributes that are removed from the head when \( Q_{76} \) is converted to \( \text{bsb}\)-NF.

Figure 6.6 also shows encoding query \( Q_{77} := \text{ENCODE}(Q_{74}) \); the absence of scalar aggregation in \( Q_{74} \) means that \( Q_{77} \) does not contain any hierarchical edges, and so the attributes in the head do not require node labels. We have shaded the attributes that are removed from the head when \( Q_{77} \) is converted to \( \text{bsb}\)-NF.
Figure 6.6: Encoding queries derived from \( Q_73 \) and \( Q_74 \), the \( \text{COCQL}\{f_b,f_s,f_h\} \) translations of SQL queries \( Q_4 \) and \( Q_6 \)
The following two mappings $h: Q_{76} \to Q_{77}$ and $h': Q_{77} \to Q_{76}$ are index-covering homomorphisms between the bsb-normal forms of $Q_{76}$ and $Q_{77}$.

\[
h := \{O_1/O', C/C', D_1/D', R/R', M_1/M', O_2/O', D_2/D'\}
\]

\[
h' := \{O'/O_2, C'/C, D'/D_2, R'/R\}
\]

Homomorphisms $h$ and $h'$ together prove $Q_{76} \approx_{\text{bsb}} Q_{77}$. Because $Q_{76}$ is the canonical form of $Q_{75}$ and therefore equivalent to it, this entails $Q_{75} \approx_{\text{bsb}} Q_{77}$ and therefore $Q_{73} \approx Q_{74}$. It follows that the SQL queries $Q_4$ and $Q_6$ are equivalent.

### 6.3 Future Work

There are many directions in which the work in this thesis can be extended. We briefly mention some of them here.

**Schema Dependencies** Throughout Chapters 2–5 we made the assumption that the set of possible database instances was not constrained by any schema dependencies. Clearly, taking schema constraints into consideration is desirable for query optimization (cf. Examples 62 and 63). In Section 6.2 we briefly described how sets of FDs and acyclic INDs can be utilized when testing CEQ encoding-equivalence (if we restrict our attention to sets of primary and foreign keys, then we can drop the acyclicity requirement [67]). This technique can be extended to HCEQ encoding-equivalence in a straightforward manner: prior to the conversion to DACQ canonical form, the nodes in the query body are visited in any order that places ancestors before their descendants, and when each node $v$ is visited the chase is invoked upon the CQ body $^+_v$.

From a practical perspective, the above technique is sufficient to handle most schema constraints permitted by commercial RDBMSs. An interesting question is whether our equivalence results can be extended to other types of schema constraints that guarantee terminating chases sequences. In particular, our characterization of core indexes in terms of MVDs assumes that all MVDs holding over the output must have originated from the query structure. If MVDs can also originate from the database schema, then the definition of core indexes breaks down—specifically, there is no longer a guarantee that there is one unique set of core indexes.

**Distributive Aggregation Functions** The most significant drawback of our modelling of aggregation functions as collection constructors is the implicit assumption that queries yielding objects of different sorts are not equivalent. For example, our techniques would fail to recognize that the following two queries are equivalent, because $Q$ and $Q'$ translate to COCQL $\{f_s.f_t.f_h\}$ queries with output sorts $\{\{\{\text{dom}\}\}\}$ and $\{\{\{\text{dom}\}\}\}$, respectively.
Many common SQL aggregation functions are distributive [48], and the ability to “reaggregate” their output values is crucial for view rewriting within data warehousing environments. There are well-known algebraic transformations for manipulating operators that apply distributive aggregation [114]. A very practical extension to our work would be to define a normalization algorithm that performs the maximal flattening of distributive aggregation functions prior to applying the techniques from our thesis. In the case of query Q' above, a normalization algorithm would merge the two sum aggregations, yielding query Q.

Inequalities and Higher-Order Comparisons Decision support workloads commonly include comparison predicates. While we have assumed an abstract domain of database constants, it would be very useful to extend our results to permit non-equality atomic comparisons over ordered concrete domains (e.g., integers or real numbers). We expect that this should be a relatively straightforward generalization of techniques known for extending CQ equivalence to atomic comparisons [70]. A more significant extension along the same vein would be to allow higher-order comparisons between aggregated values that occur either implicitly (within GROUP BY clauses) or explicitly (within WHERE or HAVING clauses). This would be particularly useful for optimizing decision support workloads, whose complex queries often contain predicates over the result of scalar aggregated subqueries (e.g., see the TPC-H [105] or TPC-DS [104] workloads). Explicit higher-order comparisons are known to make the equivalence problem undecidable with respect to the semantics of specific aggregation functions (both for numeric functions such as COUNT, SUM, etc. and for collection constructors such as the functions \( f_s, f_b, f_w \) we used in this thesis) [14, 52, 97]; however, it is possible that the equivalence problem with higher-order comparisons would remain decidable if we modelled aggregation functions as black boxes known only to satisfy certain arithmetic properties (e.g., see Cohen’s work on queries with non-nested aggregation [24]).

Outer Joins and Null Values In Chapter 4 we considered queries with explicit outer-join operators but no aggregation, whereas in the rest of the thesis we considered queries with aggregation but no explicit outer join. We expect that without too much difficulty our results could be extended to handle queries mixing both aggregation and explicit outer join, likely by using two types of hierarchical edges outputting distinct types of null values (\( \_\_\_ \) vs. NULL). A more significant extension along the same vein would be to consider database instances that contain NULL values within the base relations/tables, along with queries that permit an explicit test for NULL. This extension was only recently studied for the CQ equivalence
problem by Farré et al. [34], and it would be interesting to determine whether their results can be extended to our encoding-equivalence problem.

**View Rewriting and Cost-based Optimization**  As described in Chapter 1, an equivalence test is only one component of a view-rewriting algorithm. Given a query, our results in this thesis help to characterize the (infinite) space of equivalent logical rewritings. From a practical perspective, the most important next step is to use these results to design algorithms that generate logical query rewritings (either for the purposes of view rewriting or for query simplification). The design of any such algorithm and how it would fit into cost-based query optimization would be dependent upon the architecture of the target query optimizer, and thus this research needs to be conducted for each architecture. The results would be a set of cost-based optimizers that can efficiently process arbitrary queries with nested aggregation and outer joins in the presence of materialized views that also include nested aggregation and outer joins.
Appendix A

Proof of Theorem 3.2.4

In this section we formally prove that $\bar{\text{s}}$-equivalence of CEQs implies the existence of index-covering homomorphisms between $\bar{\text{s}}$-normal forms. Throughout this section we will make the following assumptions:

[A1] $Q(\mathcal{I}_1; \ldots ; \mathcal{I}_d; \mathcal{V})$ and $Q'(\mathcal{I}_1'; \ldots ; \mathcal{I}_d'; \mathcal{V}')$ are two CEQs satisfying $Q \equiv_\bar{s} Q'$.

[A2] $Q$ and $Q'$ are already in $\bar{s}$-NF (justified by Theorem 3.1.17).

[A3] The body of $Q$ is minimal relative to treating $Q$ as a CQ with head $Q(\mathcal{I}_1; \ldots ; \mathcal{I}_d; \mathcal{V})$ (justified by $(\mathcal{V} \cap \mathcal{B}) \subseteq \mathcal{I}_{[1,d]}$ and because minimization of CQs preserves relational equivalence). Similarly, the body of $Q'$ is minimal relative to CQ head $Q'(\mathcal{I}_1'; \ldots ; \mathcal{I}_d'; \mathcal{V}')$.

[A4] Within the context of some database $\mathcal{D}$, we use $\Gamma$ to denote both the embeddings of $\text{body}_Q$ into $\mathcal{D}$ and—by abuse of notation—the encoding relation $(Q)^\mathcal{D}$ with schema $Q(\mathcal{I}_1; \ldots ; \mathcal{I}_d; \mathcal{V})$. Then, given some value $\bar{a} \in \text{dom}(\mathcal{I}_{[1,d]})$, we use $\Gamma[\bar{a}]$ to denote the sub-relation indexed by $\bar{a}$, which has schema $Q(\mathcal{I}_1; \ldots ; \mathcal{I}_d; \mathcal{V})$ and instance $\Pi_{\mathcal{I}_{[1,d]}} \mathcal{V}(\sigma_{\mathcal{I}_{[1,d]}}(\Gamma))$ (see Section 2.2.1). We define $\Phi$ analogously for $Q'$.

The traditional proof of the Homomorphism Theorem for equivalence/containment of CQs [6, Theorem 6.2.3] can be broken down into the following four steps:

1. Use $\text{body}_Q$ to construct a “canonical database” $\mathcal{D}_Q$.
2. Choose a particular embedding $\gamma : \text{body}_Q \to \mathcal{D}_Q$ that yields a “canonical tuple” $\gamma(\mathcal{V})$ within $(Q)^{\mathcal{D}_Q}$.
3. Use the definition of query equivalence/containment to argue the existence of an embedding $\phi : \text{body}_{Q'} \to \mathcal{D}_Q$ that yields the same canonical tuple $\phi(\mathcal{V}') = \gamma(\mathcal{V})$.
4. Define a mapping $h$ in terms of $\phi$, and use both the definition of $\mathcal{D}_Q$ and the properties of the chosen canonical tuple to prove that $h$ is a homomorphism from $Q'$ to $Q$.

At a high level, our proof of Theorem 3.2.4 uses the same four steps. Unfortunately, encoding-equality between encoding relations is characterized by the existence of a $\bar{s}$-certificate (see Theorem 2.2.9), which is a significantly more complicated
condition than equality/containment of two flat relations. This extra complexity arises for two distinct reasons—first, the recursive nature of comparing nested (encoded) objects rather than flat tuples; and second, the differing semantics required at each level of the comparison depending upon the encoding signature $\xi$.

The complexity due to nested (encoded) objects forces two modifications. First, we need to generalize the idea of “equal canonical tuples” $\gamma(\overline{V}) = \phi(\overline{V}')$ to “$\xi$-equal canonical sub-relations” $\Gamma[\overline{a}_{[1,i]}] \equiv_{\xi,[i+1,d]} \Phi[\overline{a}'_{[1,i]}]$ (that is, corresponding sub-relations that encode the same “canonical sub-object”). Second, to ensure that $h$ is index-covering requires demonstrating a relationship between corresponding index assignments $\overline{a}_i$ and $\overline{a}'_i$, which requires proving properties about specific certificate nodes. To this end, Section A.1 defines certain degenerate classes of “pseudo-trivial” certificate nodes whose existence allows us to conclude the necessary properties for $h$.

To argue for the existence of a $\xi$-certificate containing the necessary pseudo-trivial nodes, we require a carefully designed canonical database $D_Q$ and corresponding definitions of canonical sub-relations/sub-objects. This is where the complexity due to varying semantics manifests itself—each semantic type places different requirements on the design of $D_Q$. Bag equivalence is based upon counting, and so for bag levels we rely upon certain algebraic properties of polynomials combined with a procedure to inflate cardinalities of database constants in a regular way such that non-pseudo-trivial certificate nodes would imply differing element cardinalities. In Section A.2 we describe this process, including the complete proof for the restricted case where every level requires bag semantics (i.e., $\xi = b^d$). Set equivalence is based upon existence, and so for set levels we inject large amounts of symmetry into the construction of $D_Q$, with the constants corresponding to the index variables in $Q$ forming the “axes” of the symmetry. We then show that non-pseudo-trivial certificate nodes would imply the existence/non-existence of extra/missing canonical sub-objects. This process is described in Section A.3, including the complete proof for the case $\xi = s^d$. Normalized bag equivalence exhibits both count- and existence-based features, and so for these levels our argument synthesizes both the polynomial and the symmetry techniques; Section A.4 details normalized bag equivalence, including the complete proof for the case $\xi = n^d$. Finally, Section A.5 shows how the three arguments can be interleaved to handle arbitrary signatures.

A.1 Pseudo-Trivial Certificate Nodes

Given two encoding relations $\Gamma$ and $\Phi$, consider any $\xi$-certificate between them. A $\xi$-certificate is a tree built out of set, bag, normalized bag, and tuple nodes (cf. Definitions 2.2.4–2.2.8). Choose any node $n$ at any level $i$ within the certificate; then, node $n$ is itself the root of a $\xi_{[i,d]}$ certificate that proves $\xi_{[i,d]}$-equality between two sub-relations $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$, for some $\overline{a} \in \text{adom}(T_{[1,i-1]} \Gamma)$ and $\overline{a}' \in \text{adom}(T_{[1,i-1]} \Phi)$. 

164
A.1.1 Trivial Nodes

Regardless of the node type of $n$ (i.e., set, bag, or normalized bag), the internal mechanisms of $n$ effect one or more mappings from $\text{adom}(\vec{I}, \Gamma[\vec{a}])$ to $\text{adom}(\vec{I}', \Phi[\vec{a}'])$ and from $\text{adom}(\vec{I}, \Phi[\vec{a}'])$ to $\text{adom}(\vec{I}', \Gamma[\vec{a}])$, and these mappings govern which $\xi_{[i+1,d]}$-certificates node $n$ must have as children. We say that $n$ is trivial w.r.t. index value $\vec{x}$ if any child $\xi_{[i+1,d]}$-certificate of $n$ that involves sub-relation $\Gamma[\vec{x}x]$ also involves sub-relation $\Phi[\vec{a}\vec{x}]$, and vice versa (i.e., all of the mappings within $n$ act as the identity for the value $\vec{x}$). We say that $n$ is trivial if it is trivial w.r.t. every value within $\text{adom}(\vec{I}, \Gamma[\vec{a}]) \cup \text{adom}(\vec{I}', \Phi[\vec{a}'])$.

We say that relations $\Gamma[\vec{a}]$ and $\Phi[\vec{a}']$ permit a proof that is trivial w.r.t. index value $\vec{x}$ if

1. $\vec{x} \in \text{adom}(\vec{I}, \Gamma[\vec{a}])$,
2. $\vec{x} \in \text{adom}(\vec{I}', \Phi[\vec{a}'])$, and
3. $\Gamma[\vec{x}x] \equiv_{\xi_{[i+1,d]}} \Phi[\vec{a}'\vec{x}]$.

If $\Gamma[\vec{a}]$ and $\Phi[\vec{a}']$ permit a proof that is trivial w.r.t. index value $\vec{x}$, we call the node $n$ potentially trivial w.r.t. index value $\vec{x}$. It is straightforward to show that if node $n$ is potentially trivial w.r.t. $\vec{x}$ then we can re-arrange the mappings within $n$ until we obtain a node that is trivial w.r.t. value $\vec{x}$ (while still forming a valid $\xi_{[1,d]}$-certificate between $\Gamma[\vec{a}]$ and $\Phi[\vec{a}']$).

We say that $\Gamma[\vec{a}]$ and $\Phi[\vec{a}']$ permit a proof that is trivial if for every value $\vec{x}$ in $\text{adom}(\vec{I}, \Gamma[\vec{a}]) \cup \text{adom}(\vec{I}', \Phi[\vec{a}'])$ they permit a proof that is trivial w.r.t. $\vec{x}$. In this case, we call node $n$ potentially trivial; again, it is straightforward to show for any potentially trivial node that we can re-arrange the mappings to obtain a trivial node.

A.1.2 Pseudo-Trivial Nodes

Observe that $\Gamma[\vec{a}]$ and $\Phi[\vec{a}']$ only permit a trivial proof if $\Gamma[\vec{a}] \equiv_{\xi_{[1,d]}} \Phi[\vec{a}']$; this follows because triviality implies equality (and hence bijectivity) between $\text{adom}(\vec{I}, \Gamma)$ and $\text{adom}(\vec{I}', \Phi)$. If $\xi_i \neq \xi$, proving that $\Gamma[\vec{a}]$ and $\Phi[\vec{a}']$ permit a trivial proof (and hence, that $n$ can be reorganized into a trivial node) is challenging since it requires demonstrating bag-equality, which is an encoding relationship that is potentially stronger than the one enforced by $n$. For this reason, we want to relax the notion of a trivial proof/node.

We say that $n$ is pseudo-trivial w.r.t. value $\vec{x}$ if $n$ contains a child $\xi_{[i+1,d]}$-certificate that proves $\Gamma[\vec{x}x_1] \equiv_{\xi_{[i+1,d]}} \Phi[\vec{a}\vec{x}_2]$, where $x_1 \in \text{adom}(\vec{I}, \Gamma[\vec{a}])$ and $x_2 \in \text{adom}(\vec{I}', \Phi[\vec{a}'])$ are two (not necessarily distinct) permutations of $\vec{x}$. We say that $n$ is pseudo-trivial if it is pseudo-trivial w.r.t. every value within $\text{adom}(\vec{I}, \Gamma[\vec{a}]) \cup \text{adom}(\vec{I}', \Phi[\vec{a}'])$.
We say that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ permit a proof that is pseudo-trivial w.r.t. value $\overline{x}$ if for some index level $n$:

1. $\exists \overline{x}_1 \in \text{adom}(\overline{I}_i, \Gamma[\overline{a}])$ that is a permutation of $\overline{x}$,
2. $\exists \overline{x}_2 \in \text{adom}(\overline{I}_i', \Phi[\overline{a}])$ that is a permutation of $\overline{x}$, and
3. $\Gamma[\overline{a}x_1] =_{\overline{x}_1[i+1,d]} \Phi[\overline{a}'x_2]$.

If $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ permit a proof that is pseudo-trivial w.r.t. value $\overline{x}$, we say that node $n$ is potentially pseudo-trivial w.r.t. $\overline{x}$ because we can re-arrange the mappings within $n$ to obtain a node that is pseudo-trivial w.r.t. $\overline{x}$.

Similarly, we say that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ permit a proof that is pseudo-trivial if for every value $\overline{x}$ in $\text{adom}(\overline{I}_i, \Gamma[\overline{a}]) \cup \text{adom}(\overline{I}_i', \Phi[\overline{a}])$ they permit a proof that is pseudo-trivial w.r.t. $\overline{x}$ in this case, we say that $n$ is potentially pseudo-trivial because we can re-arrange the mappings within $n$ to obtain a pseudo-trivial node.

Observe that pseudo-triviality allows the permutation orderings to vary between different values of $\overline{x}$. Because of this, the existence of a pseudo-trivial proof only implies set-equality between the collections encoded by $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$. Set-equality is the weakest form of encoding-equality, and so proving that $n$ can be re-organized into a pseudo-trivial node does not require demonstrating an encoding relationship stronger than the one already enforced by $n$.

**Lemma A.1.1** Let $I \in (\mathcal{V} \cap \mathcal{B})$ and $I' \in (\mathcal{V}' \cap \mathcal{B}')$ be output variables that occur in corresponding positions within $\overline{V}$ and $\overline{V}'$. Then, $I$ and $I'$ must be indexes at the same level.

**Proof.** By Lemma 3.0.1, we know $\exists i, j \in [1, d]$ such that $I \in \mathcal{I}_i$ and $I' \in \mathcal{I}_j$.

Suppose that $i > j$. Then, it is easy to construct a database $\mathcal{D}$ such that for some $\overline{a} \in \text{adom}(\overline{I}_{[1,j]}, \Gamma)$, sub-relation $\Gamma[\overline{a}]$ contains two tuples with differing values for output column $I$. In contrast, for every $\overline{a}' \in \text{adom}(\overline{I}_{[1,j]}, \Phi)$ the output column $I'$ is constant across all tuples in $\Phi[\overline{a}']$, implying $\Gamma[\overline{a}] \not\equiv_{\overline{a}'[i+1,d]} \Phi[\overline{a}']$ (a contradiction). Hence, conclude $i \not\preceq j$. Conclude $i \not\preceq j$ similarly.

**Corollary A.1.2** If for some index level $i \in [1, d]$ both $\mathcal{I}_i \in \mathcal{V}$ and $\mathcal{I}_i' \in \mathcal{V}'$, then for any database $\mathcal{D}$ and any $\overline{\mathcal{S}}$-certificate between $\Gamma$ and $\Phi$, all of the certificate nodes at level $i$ are pseudo-trivial.

### A.2 Bag Equivalence: A Counting Argument

Although bag nodes are defined in terms of a bijective function between index values, their effect can be equivalently stated in terms of counting—bag nodes effect agreement of element cardinalities. In order to argue that within a given $\overline{\mathcal{S}}$-certificate a particular bag node is trivial, we use a counting argument based upon uniqueness of polynomials. The following example illustrates the core idea.
Example 65 Consider the following three conjunctive encoding queries

\[ Q_{78}(A, C; C) \vdash \text{Student}(A, B, C) \]
\[ Q_{79}(A, B, C; C) \vdash \text{Student}(A, B, C) \]
\[ Q_{80}(B, C, D; C) \vdash \text{Student}(A, B, C), \text{Student}(A, D, C) \]

and the database \( D_{10} \) shown in Figure A.1a. The results of evaluating the queries over \( D_{10} \) are shown in Figure A.2. Clearly, the three encoding relations \( R_{21}, R_{22}, \) and \( R_{23} \) are \( b \)-equal (and hence \( n \)-equal). Now consider database \( D_{11} \) shown in Figure A.1b which was created from \( D_{10} \) by “diversifying” the constant \( CS \) into two distinct constants (which we show as \( CS \) and \( CS \)). Under \( D_{11} \), no two of the resulting encoding relations (shown in Figure A.3) are \( b \)-equal (or \( n \)-equal).

In the above example, diversifying a single constant by a factor of two effectively breaks any nodes that prove the \( b \)-equality of the original encoding relations. (Because normalized bag nodes utilize a layer of bag sub-nodes, in this example the same process also breaks any normalized bag nodes that prove \( n \)-equality.) We formalize this process for bag equality next. Formalizing it for normalized bag equality requires additional mechanisms which we defer until Section A.4.
A.2.1 Colour-Diversification

Suppose that we are given an infinite palette of colours, each indexed by a positive integer:

\[ \text{colour}_1 \, \text{colour}_2 \, \text{colour}_3 \, \text{colour}_4 \, \text{colour}_5 \, \ldots \]

(It is intentional and significant that \( \text{colour}_1 \) appears transparent.)

Let \( \overline{\text{dom}} \in \text{dom} \) be any set of \( n \) constants adhering to some total ordering \( < \).

For each integer \( i \geq 2 \), let \( \overline{\text{dom}}_i \) be a fresh set of constants isomorphic to \( \overline{\text{dom}} \), and let \( \delta_i : \overline{\text{dom}} \rightarrow \overline{\text{dom}}_i \) be a function that “paints” each \( c_j \in \overline{\text{dom}} \) with colour \( \text{colour}_i \) to yield the constant \( c_j \in \overline{\text{dom}}_i \). As implied by the transparency of \( \text{colour}_1 \), we define \( \overline{\text{dom}}_1 = \overline{\text{dom}} \) and the painting function \( \delta_1 : \overline{\text{dom}} \rightarrow \overline{\text{dom}}_1 \) as the identity function. Finally, because the different paintings of \( \overline{\text{dom}} \) are mutually disjoint, we can define a single whitewash\(^1\) function \( \delta^{-1} : \bigcup_{i} \overline{\text{dom}}_i \rightarrow \overline{\text{dom}} \) as follows.

\[
\forall i \forall c_j \in \overline{\text{dom}}_i : \delta^{-1}(c_j) := c_j \quad (A.1)
\]

We now describe a procedure we call “colour-diversification” which we define for individual tuples, but generalize to arbitrary sets of tuples (and hence relations and databases).

**Definition A.2.1 (Diversification Point)** Given totally-ordered domain \( \overline{\text{dom}} \) of \( n \) constants, a diversification point for \( \overline{\text{dom}} \) is any \( n \)-dimensional point \( \overrightarrow{r} \in \mathbb{N}^n \) whose coordinates are positive integers.

**Definition A.2.2 (\( \overrightarrow{r} \)-Colour-Diversification)** Given domain \( \overline{\text{dom}} \), diversification point \( \overrightarrow{r} \) for \( \overline{\text{dom}} \), and any tuple \( t \) over \( \overline{\text{dom}} \),

\[
t = \{ c_{i_1}, c_{i_2}, \ldots, c_{i_m} \}
\]

the colour-diversification of \( t \) w.r.t. \( \overrightarrow{r} \)—denoted \( \Delta^\overrightarrow{r}(t) \)—is the set of all possible “paintings” of \( t \) generated by independently choosing for each tuple component \( c_{i_j} \) one of the first \( r_{i_j} \) colours in the colour palette. We also refer to \( \Delta^\overrightarrow{r}(t) \) the \( \overrightarrow{r} \)-colour-diversification of \( t \) or the \( \overrightarrow{r} \)-diversification of \( t \).

We can colour-diversify an entire database \( D_i \) by colour-diversifying each tuple \( t \in D_i \) with the same point \( \overrightarrow{r} \) (where \( \overrightarrow{r} \) is a diversification point for active domain \( \overline{\text{dom}}(D_i) \)). We then obtain a new database \( D_j \)

\[
D_j = \Delta^\overrightarrow{r}(D_i) = \bigcup_{t \in D_i} \Delta^\overrightarrow{r}(t)
\]

\(^1\)The inverse of painting
which we call the \( \tau \)-\((colour-)diversification of \( D_i \). We also say that \( D_j \) is \( \tau \)-\( colour\-\)diverse \wrt point \( \tau \)—a.k.a. \( \tau \)-\( colour\-diverse \) or \( \tau \)-\( diverse \)—if \( D_j \) is the result of performing \( \tau \)-\( colour\-diversification \) on some other database \( D_i \). Because colour-diversification is defined in terms of painting, the whitewashing function \( \delta^{-1} \) from equation A.1 is also the inverse of \( \tau \)-diversification (when extended to sets of tuples). Hence, given a database \( D_j \) that is \( \tau \)-\( colour\-diverse \), we can always re-obtain the original database \( D_i \) by whitewashing \( D_j \).

\[
D_i = \delta^{-1}(D_j) = \delta^{-1} \circ \Delta^\tau(D_i)
\]

Note that when we say \( D_j \) is \( \tau \)-\( colour\-diverse \), it implies that \( \tau \) is a diversification point for active domain \( \text{adom}(\delta^{-1}(D_j)) = \text{adom}(D_i) \), \textit{not} active domain \( \text{adom}(D_j) \) (which is already colour-diverse).

\textbf{Example 66} Consider database \( D_{10} \) in Figure A.1a which contains four distinct constants in its active domain. Assume the following ordering for \( \text{adom}(D_{10}) \).

\[
\text{adom}(D_{10}) = \{\text{UW, CS, Dave, Art}\}
\]

Then, database \( D_{11} \) in Figure A.1b is the \([1,2,1,1]\)-diversification of \( D_{10} \) because constant \( c_2 = \text{CS} \) is painted with two colours. As another example, the \([2,1,3,2]\)-diversification of \( D_{10} \) is shown in Figure A.4.

The cardinality of a tuple's colour-diversification depends upon both \( \tau \) and the number of occurrences of each constant \( c_i \in \text{dom} \) within the tuple. For a given tuple \( t \), let \( \#(t,c_i) \) denote the number of occurrences of constant \( c_i \) within \( t \). Then, the set \( \Delta^\tau(t) \) has the following cardinality.

\[
|\Delta^\tau(t)| = \prod_{c_i \in \text{dom}} (r_i)^{\#(t,c_i)} \tag{A.2}
\]
Observe that if tuple $t$ is fixed, then each value $\#(t, c_i)$ is also fixed; therefore, equation A.2 defines a monomial of variables $r_1, \ldots, r_n$ with coefficient one and degree equal to the arity of $t$. Given an arbitrary set $S$ of tuples over $\overline{\text{dom}}$, the cardinality of the colour-diversification of $S$ is then given by the following multivariate polynomial $f_S$ over variables $\overline{t}$.

$$f_S(\overline{t}) = |\Delta^\overline{t}(S)| = |\bigcup_{t \in S} \Delta^\overline{t}(t)| = \sum_{t \in S} |\Delta^\overline{t}(t)| = \sum_{t \in S} \prod_{c_i \in \overline{\text{dom}}} (r_i)^{\#(t, c_i)} \quad (\text{A.3})$$

Furthermore, given two arbitrary sets $S_1$ and $S_2$ of tuples over $\overline{\text{dom}}$, equation A.3 implies the following relationships between polynomials $f_{S_1}$ and $f_{S_2}$.

$$f_{S_1}(\overline{t}) + f_{S_2}(\overline{t}) = f_{(S_1 \cup S_2)}(\overline{t}) \quad (\text{A.4})$$

$$f_{S_1}(\overline{t}) \times f_{S_2}(\overline{t}) = f_{(S_1 \times S_2)}(\overline{t}) \quad (\text{A.5})$$

**Example 67** Consider again Example 66 and observe that database $\mathcal{D}_{10}$ contains two tuples $t_1 := \text{Student(UW, CS, Dave)}$ and $t_2 := \text{Student(UW, Art, Art)}$. The colour-diversification of $t_1$ and $t_2$ yields the following monomials

$$f_{t_1}(\overline{r}) = |\Delta^\overline{t}(\text{[UW, CS, Dave]})| = (r_1)^1 (r_2)^1 (r_3)^1 (r_4)^0 = (r_1)(r_2)(r_3)$$

$$f_{t_2}(\overline{r}) = |\Delta^\overline{t}(\text{[UW, Art, Art]})| = (r_1)^1 (r_2)^0 (r_3)^0 (r_4)^2 = (r_1)(r_4)^2$$

and so the $[2,1,3,2]$-diversification of $\mathcal{D}_{10}$ (shown in Figure A.4) has the following cardinality.

$$f_{\mathcal{D}_{10}}([2,1,3,2]) = f_{t_1}([2,1,3,2]) + f_{t_2}([2,1,3,2])$$

$$= 2 \times 1 \times 3 + 2 \times 2^2$$

$$= 14$$

The symmetry that colour-diversification injects into the base relations of a database also propagates to the result of any queries. More specifically, let $\mathcal{D}$ be any database, $\overline{r}$ any diversification point for $\overline{\text{dom}}(\mathcal{D})$, and $Q''$ be any conjunctive query over $\mathcal{D}$ that does not contain constants in the query head (i.e., $Q''$ does not output constant columns). Then, the output of $Q''$ exhibits the following characteristic.

$$(Q'')^{\Delta^\overline{t}(\mathcal{D})} = \Delta^\overline{t}((Q'')^\mathcal{D}) \quad (\text{A.6})$$

The reader can verify that $[1,2,1,1]$-diversifying the query results in Figure A.2 yields the relations in Figure A.3.

Similarly, by whitewashing the result of any query over a colour-diverse database, we obtain the result of that query over the original database.

$$\delta^{-1} \left( (Q'')^{\Delta^\overline{t}(\mathcal{D})} \right) = \delta^{-1} \circ \Delta^\overline{t} ((Q'')^\mathcal{D}) = (Q'')^\mathcal{D} \quad (\text{A.7})$$

The reader can verify that whitewashing the query results in Figure A.3 yields the relations in Figure A.2.
A.2.2  \( k \)-Distinguishing Diversification

Examples 65 and 66 illustrate that by colour-diversifying a database we can differentiate queries that are not equivalent; however, the choice of the diversification point is important. By choosing diversification point \([1, 2, 1, 1]\) for \(D_{10}\) we distinguished between the results of \(Q_{78}, Q_{79}\), and \(Q_{80}\) under \(b\)-equality (and \(n\)-equality). However, the reader can verify that point \([1, 2, 2, 2]\) fails to distinguish between any of the queries under \(b\)-equality (or \(n\)-equality). Ideally, we would like to show that we can always use colour-diversification to distinguish between non-equivalent queries.

**Definition A.2.3 (\(k\)-Distinguishing Diversification Point)**
Given any positive integer \(k\) and any diversification point \(\bar{r}\) for domain \(\overline{\text{dom}}\), we say that \(\bar{r}\) is \(k\)-distinguishing if for any two sets \(S_1, S_2\) composed of tuples over \(\overline{\text{dom}}\) with maximum arity \(k\), either \(f_{S_1} = f_{S_2}\) or \(f_{S_1}(\bar{r}) \neq f_{S_2}(\bar{r})\).

**Definition A.2.4 (\(k\)-Distinguishing Database)**
We say that database \(D\) is \(k\)-distinguishing if there exists some \(k\)-distinguishing diversification point \(\bar{r}\) for domain \(\delta^{-1}(\text{adom}(D))\) such that \(D\) is colour-diverse w.r.t. \(\bar{r}\).

In light of the polynomials within Definition A.2.3, we turn our attention to known results about equivalence of polynomials. The Fundamental Theorem of Algebra (proved by Gauss) states that a univariate polynomial of degree \(d\) has \(d\) roots (some of which might be degenerate). The Schwartz-Zippel Theorem is essentially a generalization of this result to multivariate polynomials. The following formulation of the theorem is due to Rudich [98].

**Theorem A.2.5 (Schwartz-Zippel)**
Let \(f(x_1, \ldots, x_n) \neq 0\) be a degree \(k\) multivariate polynomial over some field \(F\). For any finite set \(S \subset F\), if you pick random (not necessarily distinct) \(r_1, r_2, \ldots, r_n \in S\) then

\[
\Pr[f(r_1, r_2, \ldots, r_n) = 0] \leq \frac{k}{|S|}.
\]

**Corollary A.2.6**
Let \(F = \{f_1, f_2, \ldots, f_m\}\) be any finite set of distinct multivariate polynomials over the real numbers with maximum degree \(k\) and \(m \geq 2\). Let \(S \subset \mathbb{N}\) be any finite set of positive integers. If you pick a random point \(\bar{r} \in S^n\), then

\[
\Pr[\exists i, j \in [1, m]. (i < j \land f_i(\bar{r}) = f_j(\bar{r}))] < \frac{km^2}{2|S|}.
\]

**Corollary A.2.7**
For any finite totally-ordered domain \(\overline{\text{dom}}\) and non-negative integer \(k\), there exists an infinite number of \(k\)-distinguishing diversification points.

**Corollary A.2.8**
Given any database \(D\) and any non-negative integer \(k\), there exists a colour-diversification of \(D\) that is \(k\)-distinguishing.
A.2.3 Guaranteeing Pseudo-Trivial Certificate Nodes

We now show how colour-diversification can be used to guarantee that any \( \overline{s} \)-certificate between the results of queries \( Q \) and \( Q' \) contains pseudo-trivial nodes. Consider any colour-diverse database \( D \) and any \( \overline{s} \)-certificate proving \( \Gamma \vdash \Phi \). For any \( i \in [1, d] \), let \( g \) be any node in level \( i \); then, node \( g \) proves \( \Gamma(\overline{a}) \vdash \Phi(\overline{a}') \) for some \( \overline{a} \in \text{adom}(\overline{I}_{[1,i-1]}, \Gamma) \) and \( \overline{a}' \in \text{adom}(\overline{I}_{[1,i-1]}, \Phi) \). Regardless of the value of semantic indicator \( \overline{s}_i \), node \( g \) must at the very least enforce that \( \Gamma(\overline{a}) \) and \( \Phi(\overline{a}') \) encode equivalent sets of sub-objects; therefore, let \( \text{subobjects}(n) \) denote the set of sub-objects encoded within the sub-relations of \( \Gamma(\overline{a}) \) and \( \Phi(\overline{a}) \).

\[
\text{subobjects}(n) := \text{Decode}(\Gamma(\overline{a}), \overline{s}_{[1,i+1,d]}) = \text{Decode}(\Phi(\overline{a}'), \overline{s}_{[1,i+1,d]}) \quad (A.8)
\]

For each sub-object \( o \in \text{subobjects}(n) \), let \( \Gamma(\overline{a}) \) denote the subset of \( \Gamma(\overline{a}) \) containing only the embeddings that correspond to an encoding of sub-object \( o \).

\[
\Gamma(\overline{a}) := \{ \gamma \mid \exists \overline{b} \in \text{adom}(\overline{I}_i, \Gamma(\overline{a})). (\gamma \in \Gamma(\overline{a}b) \land \text{Decode}(\Gamma(\overline{a}b), \overline{s}_{[1,i+1,d]}) = o) \} \quad (A.9)
\]

When viewed as an encoding relation, \( \Gamma(\overline{a}) \) has the same schema as \( \Gamma(\overline{a}) \). Define relation \( \Phi(\overline{a}') \) analogously.

In order to invoke count-based reasoning, we need to model sub-object cardinalities as polynomials. Consider the relation \( \Pi_{\overline{I}_i}(\Gamma(\overline{a})) \) because \( D \) is colour-diverse w.r.t. some point \( \overline{r} \), it follows from equation A.6 that \( \Pi_{\overline{I}_i}(\Gamma(\overline{a})) \) is \( \overline{r} \)-diverse. Now, for each sub-object \( o \in \text{subobjects}(n) \), consider the relation \( \Pi_{\overline{I}_i}(\Gamma(\overline{a}) \rangle_o \) — unfortunately, this relation is not necessarily \( \overline{r} \)-diverse, because all of the values in attributes \( \overline{I}_i \cap \overline{V} \) will be constant (they are functionally determined by \( o \)). However, if we define \( \overline{I}_i \) as the non-output columns within \( \overline{I}_i \), then the relation \( \Pi_{\overline{I}_i}(\Gamma(\overline{a}) \rangle_o \) is \( \overline{r} \)-diverse. Next, define set \( S_o \) as the whitewashing of the tuples in this relation instance

\[
S_o := \delta^{-1}(\Gamma(\Pi_{\overline{I}_i}(\Gamma(\overline{a}) \rangle_o))) \quad (A.10)
\]

and define polynomial \( f_{S_o} \) as in equation A.3. Finally, define \( \overline{I}_i \), \( S_o \), and \( f_{S_o} \) analogously for \( \Phi(\overline{a}') \). If we let \( \#(\Gamma(\overline{a}), o) \) and \( \#(\Phi(\overline{a}'), o) \) denote the frequency of sub-object \( o \) within the encoding relations \( \Gamma(\overline{a}) \) and \( \Phi(\overline{a}') \), then the following equations hold by construction and the fact that \( D \) is \( \overline{r} \)-diverse.

\[
\#(\Gamma(\overline{a}), o) = |\Pi_{\overline{I}_i}(\Gamma(\overline{a}) \rangle_o)| = |\Pi_{\overline{I}_i}(\Gamma(\overline{a}) \rangle_o)| = |\Delta(\overline{r}(S_o))| = f_{S_o}(\overline{r}) \quad (A.11)
\]

\[
\#(\Phi(\overline{a}'), o) = |\Pi_{\overline{I}_i}(\Phi(\overline{a}') \rangle_o)| = |\Pi_{\overline{I}_i}(\Phi(\overline{a}') \rangle_o)| = |\Delta(\overline{r}(S_o'))| = f_{S_o}(\overline{r}) \quad (A.12)
\]

Using the definitions above, we can now identify situations where certificates contain pseudo-trivial nodes. This is enough to prove Theorem 3.2.4 in the restricted case where every level requires bag semantics (i.e., \( \overline{s} = b^d \)).

Lemma A.2.9 Given

- some database \( D \) such that \( \Gamma \) and \( \Phi \) are non-empty,
Suppose that \( r \) is potentially pseudo-trivial.

**Lemma A.2.10** if \( D \) is \(|\mathcal{I}_{1,d}| + |\mathcal{I}'_{1,d}|\)-distinguishing then every node \( n^b \) in level \( i \) of the certificate is potentially pseudo-trivial.

**Proof.** Suppose that \( D \) is \( k \)-distinguishing for \( k = |\mathcal{I}_{1,d}| + |\mathcal{I}'_{1,d}| \). If \( |\mathcal{I}_i| = |\mathcal{I}'_i| = 0 \) then any node at level \( i \) is trivial; therefore assume \( |\mathcal{I}_i| + |\mathcal{I}'_i| > 0 \). Choose any \( n^b \) in level \( i \) of the certificate. Then, \( n^b \) proves \( \Gamma[\bar{a}] \simeq_{b^i[i+1,d]} \Phi[\bar{a}'] \) for some \( \bar{a} \in \text{adom}(\mathcal{T}[1,i-1]) \) and \( \bar{a}' \in \text{adom}(\mathcal{T}'[1,i-1]) \).

For each \( o \in \text{subobjects}(n^b) \), bag equivalence requires that \( \#(\Gamma[\bar{a}], o) = \#(\Phi[\bar{a}'], o) \).

Because \( D \) is colour-diverse w.r.t. some \( k \)-distinguishing point \( \mathcal{T} \), by equations A.11 and A.12, this implies \( f_{S_o}(\mathcal{T}) = f_{S'_o}(\mathcal{T}) \). Observe that sets \( S_o \) and \( S'_o \) both contain tuples whose arity does not exceed the value \( k \); hence, \( f_{S_o} = f_{S'_o} \) follows by Definitions A.2.3 and A.2.4.

By virtue of equations A.3 and A.2, \( f_{S_o} = f_{S'_o} \) implies that each tuple \( t \in S_o \) can be mapped to a permutation of itself in \( S'_o \), and vice versa. (It additionally implies that there is a bijection between \( S_o \) and \( S'_o \) that satisfies this property, but we don’t require that to prove pseudo-triviality). This further implies that a similar “permuting mapping” exists in each direction between the tuples of relations \( \Pi_{\mathcal{T}_i}(\Gamma[\bar{a}]) \) and \( \Pi_{\mathcal{T}'_i}(\Phi[\bar{a}']) \).

Lemma A.1.1 implies that the same set of constant columns both extends relation \( \Pi_{\mathcal{T}_i}(\Gamma[\bar{a}]) \) into \( \Pi_{\mathcal{T}_i}(\Gamma[\bar{a}]) \) and extends relation \( \Pi_{\mathcal{T}'_i}(\Phi[\bar{a}']) \) into \( \Pi_{\mathcal{T}'_i}(\Phi[\bar{a}']) \). Hence, the “permuting mappings” between \( \Pi_{\mathcal{T}_i}(\Gamma[\bar{a}]) \) and \( \Pi_{\mathcal{T}'_i}(\Phi[\bar{a}']) \) are easily extended into a “permuting mappings” between \( \Pi_{\mathcal{T}_i}(\Gamma[\bar{a}]) \) and \( \Pi_{\mathcal{T}'_i}(\Phi[\bar{a}']) \) — in other words, \( \Gamma[\bar{a}] \) and \( \Phi[\bar{a}'] \) permit pseudo-trivial proofs for all of the index values in \( \Pi_{\mathcal{T}_i}(\Gamma[\bar{a}]) \) and \( \Pi_{\mathcal{T}'_i}(\Phi[\bar{a}']) \). Repeating this argument for each \( o \in \text{subobjects}(n^b) \), we conclude that \( \Gamma[\bar{a}] \) and \( \Phi[\bar{a}'] \) permit a pseudo-trivial proof (which means \( n^b \) is potentially pseudo-trivial). \( \square \)

**Lemma A.2.10** If \( |\mathcal{I}| = b^d \) then there exists index-covering homomorphisms in both directions between \( Q \) and \( Q' \).

**Proof.** To construct an index-covering homomorphism \( h \) from \( Q' \) to \( Q \), we start by defining the canonical database \( D_Q \) as follows

\[
D_Q := \Delta^\mathcal{T}(\text{body}_Q) \quad \text{(A.13)}
\]

where \( \mathcal{T} \) is any \(|\mathcal{I}_{1,d}| + |\mathcal{I}'_{1,d}|\)-distinguishing diversification point for (any total ordering of) the set \( \mathcal{B} \cup \mathcal{C}_Q \) (i.e., the variables and constants occurring in \( \text{body}_Q \)).

Consider any \( b^d \)-certificate proving \( \Gamma \simeq_{b^d} \Phi \). Choose any \( \gamma \in \Gamma \) that satisfies \( \delta^{-1} \circ \gamma(\mathcal{T}_{1,d}) = \mathcal{T}_{1,d} \) (by construction of \( D_Q \), there exist many such \( \gamma \)). Let \( \bar{a}_{[1,d]} \) denote the tuple of index values \( \gamma(\mathcal{T}_{1,d}) \). By Lemma A.2.9, the root node of the certificate is potentially pseudo-trivial w.r.t. \( \bar{a}_1 \); therefore, re-arrange
the mappings within the node until it is pseudo-trivial w.r.t. $\overline{a}_i$. By recursing into any child $b^{d-1}$-certificate involving sub-relation $\Gamma[\overline{a}_1]$ and repeating this procedure, we eventually arrive at a tuple node proving $\Gamma[\overline{a}_{1,d}] \equiv \Phi[\overline{a}'_{1,d}]$ for some $\overline{a}'_{1,d} \in \text{adom}(\overline{I}'_{1,d}; \Phi)$. Choose any $\phi \in \Phi[\overline{a}'_{1,d}]$.

Define $h := \delta^{-1} \circ \phi$. Then, $h$ is a mapping from $B'$ to $\delta^{-1}(\text{adom}(D_Q)) = B \cup C_Q$ that satisfies the following properties.

1. $h(\text{body}_{Q'}) = \delta^{-1} \circ \phi(\text{body}_{Q'}) \in \delta^{-1}(D_Q) = \text{body}_Q$
2. $h(\overline{V}) = \delta^{-1} \circ \phi(\overline{V}) = \delta^{-1} \circ \gamma(\overline{V}) = \overline{V}$
3. $\forall i \in [1,d], h(I'_i) = \delta^{-1} \circ \phi(I'_i) = \delta^{-1} \circ \gamma(I_i) = I_i$

which holds because the node proving $\Gamma[\overline{a}_{1,i-1}] \equiv \xi_{\overline{a}_{1,i-1}} \Phi[\overline{a}'_{1,i-1}]$ was rearranged to be pseudo-trivial w.r.t. $\overline{a}_i$ and so $\phi(I'_i) = \overline{a}'_i$ is a permutation of $\overline{a}_i = \gamma(I_i)$.

Hence, $h$ satisfies Definition 3.2.1. The proof is analogous for $h'$ from $Q$ to $Q'$.

\[ \square \]

### A.3 Set Equivalence: A Symmetry Argument

The proof for bag equivalence in the previous section focuses on object frequencies without paying much attention to the objects themselves. In particular, within the proof of Lemma A.2.10 where we argue that any bag node involving sub-relation $\Gamma[\overline{a}_{1,i-1}]$ must be pseudo-trivial w.r.t. $\overline{a}_i$, the argument complete ignores the actual sub-object encoded by sub-relation $\Gamma[\overline{a}_{1,i}]$. Instead, the focus is entirely upon the effect that index tuple $\overline{a}_i$ has on the sub-object’s frequency.

In contrast, the argument for set equivalence depends upon examining the contents of the encoded sub-objects. To argue that any set node involving some sub-relation $\Gamma[\overline{a}_{1,i-1}]$ must be pseudo-trivial w.r.t. some $\overline{a}_i$, we use a carefully crafted “canonical sub-object” that can be uniquely identified only by the precise collection of constants within index tuple $\overline{a}_{1,i-1}$. We illustrate the basic idea using the following (rather lengthy) example. For the duration of this section we assume that the encoding signature contains only set semantics ($\overline{s} = s^d$).

**Example 68** Consider a relation schema $\text{Wrote}(\text{Name}, \text{Title}, \text{Year})$ that models the fact that an author published a particular title in a particular year. Assume that each publication is uniquely identified by the combination of its title and year, so multiple tuples with the same title and year represent a co-authored publication. Figure A.5 shows a sample database instance $D_{12}$ containing relation $\text{Wrote}$.

Now consider the following CEQ $Q_{81}$. Figure A.5 shows the encoding relation $R_{27}$ that results from evaluating $Q_{81}$ over $D_{12}$, as well as the object $o_{16}$ that results from decoding $R_{27}$ with signature ss.

$$Q_{81}(N,M,Y;T,T) :- \text{Wrote}(N,T,Y), \text{Wrote}(M,T,Y)$$
Intuitively, encoding query $Q_{81}$ groups together publication titles related to pairs of (not necessarily distinct) author names and a particular year; hence, object $o_{16}$ could be called the "set of annual co-authorships," where each "annual co-authorship" is simply a set of related titles. The crucial point is that $o_{16}$ does not actually contain the author names and year that relate the titles within a particular annual co-authorship.

Now consider three more CEQs that all contain fewer index variables than $Q_{81}$.

$$Q_{82}(N,Y;T;T) :\leftarrow \text{Wrote}(N,T,Y)$$
$$Q_{83}(T;:) :\leftarrow \text{Wrote}(N,T,Y)$$
$$Q_{84}(N,M;T;T) :\leftarrow \text{Wrote}(N,T,Y), \text{Wrote}(M,T,Y)$$

Ideally, we would like $o_{16}$ to contain a sub-object that can be uniquely identified only via the author pair and year that index it in $R_{27}$. Such an sub-object could be used to
distinguish $o_{16}$ from the result of any CEQ whose resulting encoding relation does not contain an index tuple with the same combination of authors and year (including the three queries above).

Consider sub-object \{ A1, AB1 \} corresponding to index tuple Amy-Amy-2007 in $R_{27}$. The reader can easily verify that this sub-object also occurs in the result of $Q_{82}$ corresponding to index tuple Amy-2007. The same argument applies to any sub-object corresponding to an index tuple in $R_{27}$ with a repeated author name, including \{ A2, AB2, AB3, AC1, AC2 \}, \{ B1, AB1 \}, \{ B2, AB2, AB3 \}, and \{ C1, AC1, AC2 \}.

Consider sub-object \{ B1 \} corresponding to index tuples Amy-Bob-2007 and Bob-Amy-2007 in $R_{27}$. Clearly, this sub-object also occurs in the result of $Q_{83}$ corresponding to index tuple Amy-2007. The same argument applies to any sub-object corresponding to an index tuple in $R_{27}$ with a repeated author name, including \{ A2, AB2, AB3, AC1, AC2 \}, \{ B1, AB1 \}, \{ B2, AB2, AB3 \}, and \{ C1, AC1, AC2 \}.

Consider sub-object \{ AC1, AC2 \} corresponding to index tuples Amy-Chad-2008 and Chad-Amy-2008 in $R_{27}$. This sub-object also occurs in the result of $Q_{84}$ corresponding to index tuples Amy-Chad and Chad-Amy.

Finally, consider sub-object \{ AB2, AB3 \} corresponding to index tuples Amy-Bob-2008 and Bob-Amy-2008 in $R_{27}$. Because the sub-object is not singleton it does not occur in the result of $Q_{83}$, and it also does not occur in the results of $Q_{82}$ or $Q_{84}$ because each of those encoding relations only contains index tuples with two out of the three necessary values needed to isolate the set \{ AB2, AB3 \}.

Unfortunately, sub-object \{ AB2, AB3 \} is not sufficient to distinguish object $o_{16}$ from the result of any CEQ. Consider the following query $Q_{85}$, which is also in ss-NF.

$$Q_{85}(T', T''; T; T) :- \text{Wrote}(N, T, Y), \text{Wrote}(M, T, Y), \text{Wrote}(N, T', Y), \text{Wrote}(M, T'', Y)$$

The reader can verify that the relation $(Q_{85})^{D_{12}}$ contains an encoding of sub-object \{ AB2, AB3 \} (corresponding to index tuples A2-B2 and B2-A2). Query $Q_{85}$ exploits the fact that within $D_{12}$, occurrences of the values A2 and B2 are correlated with occurrences of other constants. Specifically, any embedding of $Q_{85}$ into $D_{12}$ that maps attribute $T'$ to A2 must also map $N$ to Amy and $Y$ to 2008; similarly, mapping $T''$ to B2 forces mappings of $M$ to Bob and $Y$ to 2008. By “triangulating” index values A2 and B2, $Q_{85}$ effectively isolates the values Amy, Bob, and 2008 without explicitly naming them. Indiscriminately adding a few tuples with titles A2 or B2 to the database instance will not necessarily prevent this, either. For example, adding the tuples \text{Wrote}(Don, A2, 2008) and \text{Wrote}(Chad, A2, 2007) to $D_{12}$ does not change the sub-object corresponding to A2-B2 or B2-A2 in $(Q_{85})^{D_{12}}$, because these tuples do not increase the possible choices for authors who co-authored with Bob in 2008. However, by adding tuple \text{Wrote}(Chad, B2, 2008) to $D_{12}$ we allow variable $M$ to map to either Bob or Chad, which changes the sub-object A2-B2 (and B2-A2) in $(Q_{85})^{D_{12}}$ to \{ AB2, AB3, AC1, AC2 \} without affecting the value of sub-object Amy-Bob-2008/Bob-Amy-2008 in $(Q_{27})^{D_{12}}$.

Example 68 only considers encoding queries/relations of depth two, which is actually the simplest interesting case for nested sets (because a depth one query in
s-NF is essentially just a CQ). Nevertheless, from the example we can still draw several lessons about the necessary features of any canonical sub-object and the implications for the structure of the database instance that yields it; these lessons generalize to considering canonical sub-objects of varying depth. First of all, the canonical object needs to be a strict subset of the objects formed when one or more of the index columns is removed from the query head. Second, the canonical object cannot be a singleton set, or else it could be constructed by grouping on values from inner index columns. Finally, the database needs to contain enough symmetry to guarantee that isolating an index value via data triangulation requires more triangulation points than there are index columns in either $Q$ or $Q'$.

### A.3.1 Labels, Prefixes and Canonical Database $D_Q$

We now describe a method for constructing canonical database $D_Q$ from the body of query $Q$ that contains large amounts of symmetry. The symmetry is designed so that each constant is related to a large number of other constants in a symmetric manner, which forms the basis of the argument against isolating particular index values via triangulation. Whereas in Section A.2 we “painted” with different colours to generate multiple copies of database constants, in this section we will embellish database constants with labels which we denote as superscripts.

Choose $N$ to be any integer satisfying the following constraint.

$$N > \text{max}(|\mathcal{I}_{[1,d]}|, |\mathcal{I}_{[1,d]}|) + 1 \quad (A.14)$$

The symmetry will be specified using mechanisms we will call label-generating components, sequences, and prefixes. For each level $i \in [1,d]$, let $LGC_i$ denote the set of label-generating components at level $i$, defined as follows.

$$LGC_i := \{ (\bar{y}_i, z_i) \mid y_i, z_i \in [1, N]|\mathcal{I}_i| \} \quad (A.15)$$

We say that component $c = (\bar{y}_i, z_i)$ contains a conflict at position $i,j$ if $y_{i,j} = z_{i,j}$, and we use $CF-LGC_i$ to denote the conflict-free subset of $LGC_i$.

$$CF-LGC_i := LGC_i \setminus \{ (\bar{y}_i, z_i) \mid \exists j: (y_{i,j} = z_{i,j}) \} \quad (A.16)$$

Observe that set $LGC_i$ has size $N^{2|\mathcal{I}_i|}$, while set $CF-LGC_i$ has size $N^{|\mathcal{I}_i|}(N-1)^{|\mathcal{I}_i|}$.

**Example 69** Suppose $Q = Q_{S1}$ and $N = 6$. Because every integer in $[1, N]$ is a single digit, we will represent each $\bar{x} \in [1, N]|\mathcal{I}_i|$ simply as a $|\mathcal{I}_i|$-digit number. Then, sets $LGC_2$ and $CF-LGC_2$ are as follows:

$$LGC_2 = \{(1,1), \ldots , (1,6), (2,1), \ldots , (6,6)\}$$

$$CF-LGC_2 = LGC_2 \setminus \{(1,1), (2,2), \ldots , (6,6)\}$$
continuing example 69, consider the following label-generating sequences

\[ \text{LGC}_1 = \{(111, 111), \ldots (111, 666), (112, 111), \ldots (666, 666)\} \]

\[ \text{CF-LGC}_1 = \text{LGC}_1 \setminus \{(111, 111), \ldots , (111, 661), (112, 111), \ldots , (666, 666)\} \]

with sizes \(N[2^{12}] = 36\) and \(N[2^{11}] (N-1)[2^{12}] = 30\), respectively. Similarly, \(\text{LGC}_1\) and \(\text{CF-LGC}_1\) are as follows

\[ \text{LGC}_1 = \{(111, 111), \ldots (111, 666), (112, 111), \ldots (666, 666)\} \]

\[ \text{CF-LGC}_1 = \text{LGC}_1 \setminus \{(111, 111), \ldots , (111, 661), (112, 111), \ldots , (666, 666)\} \]

with sizes \(N[2^{12}] = 46656\) and \(N[2^{11}] (N-1)[2^{12}] = 27000\), respectively.

Let \(\text{LGS}\) denote the set of label-generating sequences, composed out of label-generating components as follows.

\[ \text{LGS} := \{c_1.c_2 \ldots c_d \mid \forall i \in [1,d] : c_i \in \text{LGC}_i\} \]  

(A.17)

We say that a sequence \(s \in \text{LGS}\) is conflict-free if it is composed entirely of conflict-free components, and we use \(\text{CF-LGS}\) to denote the conflict-free subset of \(\text{LGS}\).

For each integer \(m \in [1,d]\), let \(\text{LGP}_m\) denote the set of label-generating prefixes of length \(m\), defined as follows.

\[ \text{LGP}_m := \{c_1.c_2 \ldots c_{(m-1)}.\bar{\gamma}_m \mid \forall i \in [1,m-1]. (c_i \in \text{LGC}_i) \wedge \exists \bar{\gamma}_m. ((\bar{\gamma}_m, \bar{\gamma}_m) \in \text{LGC}_m)\} \]  

(A.18)

Let \(\text{LGP}\) denote the set of all label-generating prefixes. Every sequence \(s \in \text{LGS}\) corresponds to a unique prefix in each of \(\text{LGP}_1, \ldots, \text{LGP}_d\). Conversely, every prefix \(p \in \text{LGP}_m\) with \(m < d\) can be extended in \(N[2^{11}] \cdot N[2^{(m+1)}]\) different ways to yield a prefix in \(\text{LGP}_{m+1}\), while every prefix \(p \in \text{LGP}_d\) can be extended in \(N[2^{12}]\) different ways to yield a complete sequence in \(\text{LGS}\). We say that a prefix \(p \in \text{LGP}\) is conflict-free if it can be iteratively extended into a conflict-free sequence, and we use \(\text{CF-LGP}_m\) (\(\text{CF-LGP}\)) to denote the conflict-free subset of \(\text{LGP}_m\) (\(\text{LGP}\)). Every conflict-free prefix \(p \in \text{CF-LGP}_m\) with \(m < d\) can be extended in \((N-1)[2^{12}] \cdot N[2^{(m+1)}]\) different ways to yield a conflict-free prefix in \(\text{CF-LGP}_{m+1}\), while every conflict-free prefix \(p \in \text{CF-LGP}_d\) can be extended in \((N-1)[2^{12}]\) different ways to yield a conflict-free sequence in \(\text{CF-LGS}\).

**Example 70** Continuing example 69, consider the following label-generating sequences \(s_1, s_2, s_3 \in \text{LGS}\) and prefixes \(p_{12}, p_3, p_{123} \in \text{LGP}\).

\[ s_1 = (354, 133). (6, 5) \quad p_{12} = (354, 133). 6 \]
\[ s_2 = (354, 133). (6, 6) \quad p_3 = (354, 134). 1 \]
\[ s_3 = (354, 134). (1, 2) \quad p_{123} = 354 \]

Sequence \(s_1\) is conflict-free, while \(s_2\) and \(s_3\) contain conflicts at positions 2.1 and 1.3, respectively. Prefix \(p_{12}\) is the length 2 prefix of both \(s_1\) and \(s_2\); \(p_3\) is the length 2 prefix of \(s_3\); and \(p_{123}\) is the length 1 prefix of \(s_1, s_2, \text{ and } s_3\). Finally, \(p_{12}\) and \(p_{123}\) are conflict-free, but \(p_3\) is not.

We now define a complex system of labels in terms of the label-generating components/sequences/prefixes defined above. The easiest way to describe the labels
Because the labelling system is to show how they are applied; hence, for the moment, let \( l \) denote an abstract label. Corresponding to each such label \( l \), let \( \mathcal{B}^l \) be a fresh set of constants isomorphic to \( \mathcal{B} \) (the variables in query \( Q \)), and let \( \alpha' : \mathcal{B} \rightarrow \mathcal{B}^l \) be a “labelling function” from each \( x \in \mathcal{B} \) to the corresponding constant in \( \mathcal{B}^l \) (which we will denote as \( x' \)). Because all such sets \( \mathcal{B}^l \) are mutually disjoint, we use \( \alpha^{-1} \) to denote the obvious “de-labelling function” \( \alpha^{-1} : (\bigcup \mathcal{B}^l) \rightarrow \mathcal{B} \) which we extend with identity over constants in \( \mathcal{C}_Q \) (the constants in query \( Q \)).

For each label-generating sequence \( s \in \text{LGS} \), we now define the following label-generating function \( \beta_s : \mathcal{B} \rightarrow (\bigcup \mathcal{B}^l) \) which performs a particular labelling of the variables of \( Q \).

\[
\beta_{(\bar{y}_1, \bar{z}_1) \ldots (\bar{y}_d, \bar{z}_d)}(x) := \begin{cases} 
\alpha^{y_1, j}(x) & \text{if } \exists j \in [1, |\mathcal{I}_1|] \text{ such that } x = I_{1, j} \\
\alpha^{z_1, \bar{y}_2, j}(x) & \text{if } \exists j \in [1, |\mathcal{I}_2|] \text{ such that } x = I_{2, j} \\
\alpha^{z_1, \bar{z}_2, \bar{y}_3, j}(x) & \text{if } \exists j \in [1, |\mathcal{I}_3|] \text{ such that } x = I_{3, j} \\
\vdots & \vdots \\
\alpha^{z_1 \ldots \bar{z}_{(d-1)}, \bar{y}_d, j}(x) & \text{if } \exists j \in [1, |\mathcal{I}_d|] \text{ such that } x = I_{d, j} \\
\alpha^{z_1 \ldots \bar{z}_d}(x) & \text{otherwise}
\end{cases}
\] (A.19)

We immediately extend function \( \beta_s \) to tuples, sets, and atoms and with identity on constants in \( \mathcal{C} \). We make the following observations about function \( \beta_s \).

1. Because \( \beta_s \) is defined purely in terms of labelling functions, the de-labelling function \( \alpha^{-1} \) acts as an inverse of \( \beta_s \).
2. For any level \( m \in [1, \bar{m}] \), the labels that \( \beta_s \) assigns to index variables in \( \mathcal{I}_m \) depend only on the length \( m \) prefix of \( s \). Therefore, for each \( p \in \text{LGP}_m \) we can define the function \( \beta_{p} : \mathcal{I}_{[1, m]} \rightarrow (\bigcup \mathcal{B}^l) \) such that \( \beta_{p}(\mathcal{I}_{[1, m]}) = \beta_s(\mathcal{I}_{[1, m]}) \) for every extension \( s \) of \( p \).
3. Suppose \( s \in \text{LGS} \) contains a conflict at position \( i.j \). Given any tuple \( t \) containing both variable \( I_{i,j} \) and some variable \( I \in \mathcal{B} \setminus \mathcal{I}_{[1, i]} \) (i.e. a non-index variable or a member of \( \mathcal{I}_{[i+1, \bar{m}]} \)), the labelled tuple \( \beta_s(t) \) evidences the conflict at \( i.j \). That is, because \( \beta_s(I_{i,j}) = I_{i,j} \) and \( \beta_s(I) \) is assigned a label that starts with \( \bar{z}_1 \ldots \bar{z}_i \), from tuple \( \beta_s(t) \) we can infer that \( y_{i,j} = z_{i,j} \) and so conclude that sequence \( s \) has a conflict at position \( i.j \).

We now define the canonical database \( D_Q \) as follows.

\[
D_Q := \bigcup_{s \in \text{CF-LGS}} \beta_s(\text{body}_Q)
\] (A.20)

Because variable \( s \) ranges over only conflict-free label-generating sequences, database \( D_Q \) does not contain any tuple that evidences any conflicts. Furthermore, observe that by de-labelling \( D_Q \) we re-obtain \( \text{body}_Q \).

\[
\alpha^{-1}(D_Q) = \text{body}_Q
\] (A.21)

179
Example 71 Continuing Example 70, functions $\beta_{p_{123}}$, $\beta_{p_{12}}$, and $\beta_{p_3}$ work as follows (recall that we assumed $Q = Q_{s1}$).

$$
\beta_{p_{123}}(I_1) = \beta_{s_1}(I_1) = \beta_{s_2}(I_1) = \beta_{s_3}(I_1) = \{N^3, M^5, Y^4\} \\
\beta_{p_{12}}(I_{[1,2]}) = \beta_{s_1}(I_{[1,2]}) = \beta_{s_2}(I_{[1,2]}) = \{N^3, M^5, Y^4, T^{133.6}\} \\
\beta_{p_3}(I_{[1,2]}) = \beta_{s_3}(I_{[1,2]}) = \{N^3, M^5, Y^4, T^{134.1}\}
$$

Thus, applying functions $\beta_{s_1}$, $\beta_{s_2}$, and $\beta_{s_3}$ to body $Q_{s1}$ yields the following sets of tuples.

$$
\beta_{s_1}(\text{body}_{Q_{s1}}) = \{\text{Wrote}(N^3, T^{133.6}, Y^4), \text{Wrote}(M^5, T^{133.6}, Y^4)\} \\
\beta_{s_2}(\text{body}_{Q_{s1}}) = \{\text{Wrote}(N^3, T^{133.6}, Y^4), \text{Wrote}(M^5, T^{133.6}, Y^4)\} \\
\beta_{s_3}(\text{body}_{Q_{s1}}) = \{\text{Wrote}(N^3, T^{134.1}, Y^4), \text{Wrote}(M^5, T^{134.1}, Y^4)\}
$$

Database $D_{Q_{s1}}$ is constructed out of $|\text{CF-LGS}| = |\text{CF-LGC}_1| \times |\text{CF-LGC}_2| = 162000$ labelled copies of body $Q_{s1}$, one for each conflict-free label-generating sequence.\(^2\) Note that in this particular example, body $Q_{s1}$ does not contain any non-index variables which implies

$$
\beta_{s_2}(\text{body}_{Q_{s1}}) = \beta_{p_{12}}(\text{body}_{Q_{s1}}) = \beta_{s_1}(\text{body}_{Q_{s1}}) \subset D_{Q_{s1}}
$$
even though $s_2 \notin \text{CF-LGS}$; however, this is a special case. Indeed, any tuple containing both $T^{134.1}$ and $Y^4$ evidences a conflict at position 1.3, and so $\beta_{s_3}(\text{body}_{Q_{s1}}) \notin D_{Q_{s1}}$.

### A.3.2 Canonical Objects

Having just defined the canonical database $D_Q$, we now define the canonical objects encoded within $\Gamma$.

**Definition A.3.1 (Canonical Object)** Given any integer $m \in [1, d]$ and any label-generating prefix $p \in \text{LGP}_m$, we define the canonical object for prefix $p$—denoted $o_p$—as follows.

$$
o_p := \text{DECODE}(\Gamma[\beta_p(\overline{I}_{[1,m]}); \overline{\beta}_{[m+1,d]}])
$$

Observe that if $m = d$, the canonical object for prefix $p \in \text{LGP}_d$ is simply the tuple $\beta_p(\overline{V})$. The following two lemmas reveal how the design of database $D_Q$ endows canonical objects with specific properties.

**Lemma A.3.2** Given any prefix $p \in \text{LGP}_d$,

1. if $p$ is conflict-free then $\beta_p(\overline{I}_{[1,d]}) \in \text{adom}(\overline{I}_{[1,d]}; R)$; and
2. if $\beta_p(\overline{I}_{[1,d]}) \in \text{adom}(\overline{I}_{[1,d]}; R)$ then for any position $i, j$ for which $p$ contains a conflict, that conflict is evidenced within the tuple $\beta_p(\overline{I}_{[i+1,d]}; \overline{V})$.

\(^2\)That database $D_Q$ has exponential size compared to $Q$ is not important because we never materialize $D_Q$ nor evaluate any query against it. For the purposes of the proof, it suffices that $D_Q$ is finite.
PROOF. If \( p \) is conflict-free, then there exists some \( s \in CF-LGS \) that extends \( p \), and so \( \beta_p(\overline{I}_{[1,d]}) = \beta_s(\overline{I}_{[1,d]}) \in \text{adom}(\overline{I}_{[1,d]}, R) \) follows by equation A.20 Conversely, suppose both that \( \beta_p(\overline{I}_{[1,d]}) \in \text{adom}(\overline{I}_{[1,d]}, R) \) and that \( p \) contains a conflict at position \( i,j \); this entails \( i \in [1,d-1] \), \( j \in [1,|\mathcal{I}|] \), and \( |\mathcal{I}_{[i+1,d]}| > 0 \). If attribute \( I_{i,j} \in \mathcal{V} \), then tuple \( \beta_p(\overline{I}_{[i+1,d]}) \mathcal{V} \) is guaranteed to evidence the conflict (see Section A.3.1). If \( I_{i,j} \notin \mathcal{V} \), let \( s \in LGS \) be any label-generating sequence that extends \( p \). By the design of \( D_Q \), embedding \( \gamma := \beta_s \) proves \( \gamma = (\overline{I}_{[1,d]} \setminus \{I_{i,j}\}) \{ \vdash I_{i,j} \} \) which, by Definition 3.1.11, implies that \( Q \) is not in \( s^d\text{-NF} \) (a contradiction). \( \square \)

Lemma A.3.3 Given any integer \( m \in [1,d] \) and any prefix \( p \in CF-LGP_m \), canonical object \( o_p \) contains enough information to reconstruct the value \( \beta_p(\overline{I}_m) \).

PROOF. By induction on \( m \). Our assumption that \( Q \) is in \( s^d\text{-NF} \) implies that \( I_d \in \mathcal{V} \) (see Definition 3.1.11); therefore, the base case \( m = d \) is trivial because object \( o_p \) is a single tuple whose output values contain all of the values in \( \beta_p(\overline{I}_m) \). Now suppose that the lemma holds for all prefixes of length in the range \([m+1,d]\). Recall from equations A.18 and A.19 that \( p \) has the form \( p = (\overline{y}_1, \overline{z}_1) \ldots (\overline{y}_{(m-1)}, \overline{z}_{(m-1)}) \overline{y}_m \), but computing \( \beta_p(\overline{I}_m) \) only requires deducing the sequence \( \overline{z}_1 \ldots \overline{z}_{(m-1)} \overline{y}_m \). Consider each prefix \( q \in LGP_{(m+1)} \) that extends \( p \). If \( q \) is conflict-free, then by the induction hypothesis, \( o_q \) contains enough information to identify the value \( \beta_q(\overline{I}_{(m+1)}) \), from which we can deduce the values of sequence \( \overline{z}_1 \ldots \overline{z}_{(m-1)} \overline{z}_m \overline{y}_{(m+1)} \), where the values \( \overline{z}_m \) and \( \overline{y}_{(m+1)} \) vary for different choices of \( q \). If \( q \) contains a conflict, then Lemma A.3.2 guarantees that the existence of the conflict can be detected by examining \( o_q \). By examining all of the sub-objects of \( o_p \) we can determine which values of \( \overline{z}_m \) corresponds to a conflict-free extension of \( p \), from which we can deduce the value of \( \overline{y}_m \). \( \square \)

A.3.3 Pseudo-Trivial Certificate Nodes

We now show that sub-relations of \( \Gamma \) that encode canonical objects can only be equated to sub-relations of \( \Phi \) by pseudo-trivial certificate nodes.

Lemma A.3.4 Given

- any integer \( m \in [1,d] \),
- any prefix \( p \in CF-LGP_m \), and
- any sub-relation \( \Phi[\overline{a}_{[1,m]}] \) that encodes canonical object \( o_p \);

index tuple \( \overline{a}_{[1,m]} \) must contain all of the values in \( \beta_p(\overline{I}_m) \).

PROOF. For each \( j \in [1,|\overline{I}_m]| \), consider the index attribute \( I_{m,j} \in \mathcal{I}_m \) and the corresponding value \( \beta_p(I_{m,j}) = I^{\overline{z}_1 \ldots \overline{z}_{(m-1)} \overline{y}_{m,j}} \). By Lemma A.3.3, index tuple \( \overline{a}_{[1,m]} \) must contain enough information to compute the value \( \beta_p(I_{m,j}) \). In order to do this, either \( \overline{a}_{[1,m]} \) must contain \( \beta_p(I_{m,j}) \), or \( \overline{a}_{[1,m]} \) must contain enough values to isolate \( \beta_p(I_{m,j}) \) via data triangulation. By the symmetry in the construction
of \( D_Q \), every database constant that co-occurs with \( \beta_p(I_{m,j}) \) in a tuple also co-occurs symmetrically with at least \( N - 2 \) other constants of the form \( \overline{a}_{\pi_1 \ldots \pi_(m-1)} \). Therefore, isolating \( \beta_p(I_{m,j}) \) via data triangulation requires at least \( N - 1 > |\overline{Z}_{[1,d]}| \) triangulation points, which exceeds the size of \( \overline{a}_{[1,m]} \). Hence, \( \overline{a}_{[1,m]} \) must contain the value \( \beta_p(I_{m,j}) \).

\( \square \)

**Lemma A.3.5** Given

- any conflict-free label-generating sequence \( s \in CF-LGS \),
- any \( \overline{s} \)-certificate that proves \( \Gamma \models \Phi \), and
- any integer \( m \in [1, d] \);

every certificate node involving sub-relation \( \Gamma[\beta_s(\overline{Z}_{[1,m-1]}))] \) is pseudo-trivial w.r.t. \( \beta_s(\overline{Z}_m) \).

**Proof.** Suppose that \( Q \models \Phi \). We have not constrained which query was chosen as \( Q \) and which was chosen as \( Q' \), and so assume without loss of generality that if \( Q \) and \( Q' \) do not have the same number of index variables at every level, then \( Q \) was chosen so that \( |\overline{Z}_m| > |\overline{Z}'_m| \) at the first level \( m \) upon which they disagree, starting at level 1 and counting up to \( d \).

Choose any conflict-free label-generating sequence \( s \in CF-LGS \); within any \( s^d \)-certificate proving \( R \models \Phi \), the certificate must contain a tuple node proving \( \Gamma[\beta_s(\overline{Z}_{[1,d]}))] \models \Phi[\overline{a}_{[1,d]}] \) for some \( \overline{a}_{[1,d]} \in \text{adom}(\overline{Z}_{[1,d]}, \Phi) \). Lemma A.3.4 implies that \( \overline{a} \) contains all values in \( \beta_s(\overline{Z}_1) \), and so by choice of \( Q \) above conclude that \( |\overline{Z}_1| = |\overline{Z}'_1| \) and therefore \( \overline{a}_1 \) is a permutation of \( \beta_s(\overline{Z}_1) \). A simple induction proves that \( \overline{a}_m \) is a permutation of \( \beta_s(\overline{Z}_m) \) for every \( m \in [1, d] \).

\( \square \)

**Lemma A.3.6** If \( \overline{s} = s^d \) then there exists index-covering homomorphisms in both directions between \( Q \) and \( Q' \).

**Proof.** Choose any conflict-free label-generating sequence \( s \in CF-LGS \). Consider any \( s^d \)-certificate proving \( \Gamma \models \Phi \). The certificate must contain a tuple node proving \( \Gamma[\beta_s(\overline{Z}_{[1,d]}))] \models \Phi[\overline{a}_{[1,d]}] \) for some \( \overline{a}_{[1,d]} \in \text{adom}(\overline{Z}_{[1,d]}, \Phi) \). Choose any \( \phi \in \Phi[\overline{a}_{[1,d]}] \).

Define \( h := \alpha^1 \circ \phi \). Then, \( h \) is a mapping from \( B' \) to \( \alpha^1(\text{adom}(D_Q)) = B \cup C_Q \) that satisfies the following properties.

1. \( h(\text{body}_Q) = \alpha^1 \circ \phi(\text{body}_{Q'}) \subseteq \alpha^1(\text{D}_Q) = \text{body}_Q \)
2. \( h(\overline{V}) = \alpha^1 \circ \phi(\overline{V'}) = \alpha^1 \circ \beta_s(\overline{V}) = \overline{V} \)
3. \( \forall i \in [1, d], h(\overline{I}_i) = \alpha^1 \circ \phi(\overline{I}'_i) = \alpha^1 \circ \beta_s(\overline{I}_i) = \overline{I}_i \)

By Lemma A.3.5 the node proving \( \Gamma[\beta_s(\overline{Z}_{[1,i-1]}))] \models \Phi[\overline{a}_{[1,i-1]}] \) is pseudo-trivial w.r.t. \( \beta_s(\overline{Z}_i) \), and so \( \phi(\overline{Z}_i) = \overline{a}_i \) is a permutation of \( \beta_s(\overline{Z}_i) \).

Hence, \( h \) satisfies Definition 3.2.1. The proof is analogous for \( h' \) from \( Q \) to \( Q' \).
### A.4 Normalized Bag Equivalence: Symmetry + Counting

Whereas bag equivalence requires agreement of absolute frequencies between corresponding sub-objects, normalized bag equivalence only requires agreement of relative frequencies. We introduced *colour-diversification* in Section A.2 as a technique for controlling the frequency of sub-objects, and in Lemma A.2.9 we showed that by applying colour-diversification to any database we can guarantee the existence of pseudo-trivial nodes within the $\bar{\mathcal{S}}$-certificate (assuming $Q \simeq T Q'$). While colour-diversification is still a key technique in our proof for normalized bag equivalence, the following example shows that we require more care when selecting the database to diversify. For the duration of this section we assume that the encoding signature contains only normalized bag semantics ($\bar{\mathcal{S}} = n^d$).

**Example 72** Consider the following two CEQs, which assume the same relation schema $\text{Wrote}(\text{Name}, \text{Title}, \text{Year})$ as used in Example 68.

\[
\begin{align*}
Q_{86}(T, Y; Y) :&= \text{Wrote}(N, T, Y) \\
Q_{87}(N, T, Y; Y) :&= \text{Wrote}(N, T, Y)
\end{align*}
\]

Intuitively, both queries characterize the relative annual publication rate. They differ in that $Q_{86}$ counts titles (per year), whereas $Q_{87}$ counts (author name, title) pairs (per year).

Consider our approach for testing bag equivalence (Section A.2), using the canonical database defined by equation A.13. Start with database $D_{13}$ formed from body $Q_{86}$,

\[
D_{13} = \{ \text{Wrote}(N, T, Y) \}
\]

assume $\text{adom}(D_{13})$ has order $\{N, T, Y\}$, and choose any 3-distinguishing diversification point $\bar{T}$. For conciseness of presentation we will use $\bar{T} = [2, 3, 2]$ even though it is not 3-distinguishing. Encoding relations $R_{28}$ and $R_{29}$ in Figure A.6 shows the result of evaluating $Q_{86}$ and $Q_{87}$, respectively, over database $D_{14} = \Delta^{[2,3,2]}(D_{13})$. Whereas $D_{14}$ discriminates between $R_{28}$ and $R_{29}$ under $b$-equality, it does not discriminate between them under $n$-equality because the effect of multiplicative factors $r_1 = 2$ and $r_2 = 3$ (corresponding to number of colours for $N$ and $T$, respectively) is voided by the normalization.

We can create a database that discriminates between $Q_{86}$ and $Q_{87}$ under $n$-equality by a slight modification of the above approach. Start with database $D_{15}$ formed from two distinct copies of body $Q_{86}$,

\[
D_{15} = \{ \text{Wrote}(N, T, Y), \text{Wrote}(N', T', Y') \}
\]

assume $\text{adom}(D_{13})$ has order $\{N, T, Y, N', T', Y'\}$, and choose $\bar{T} = [2, 3, 2, 1, 1, 1]$. Figure A.7 shows the result of evaluating $Q_{86}$ and $Q_{87}$ over database $\Delta^{[2,3,2,1,1,1]}(D_{15})$, along with the differing normalized bags obtained after decoding.
Figure A.6: Database $D_{14} = \Delta^{[2,3,2]}(D_{13})$, query results $R_{28} = (Q_{86})^{D_{14}}$ and $R_{29} = (Q_{87})^{D_{14}}$, and object $o_{17} = \text{DECODE}(R_{28}, n) = \text{DECODE}(R_{29}, n)$

$\begin{array}{|c|c|c|c|}
\hline
N & T & Y \\
N & T & Y \\
N & Y & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
\hline
\end{array}$

$\begin{array}{|c|c|c|}
\hline
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
\hline
\end{array}$

$\begin{array}{|c|c|c|}
\hline
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
Q_{86} & Q_{87} & D_{14} \\
\hline
\end{array}$

Figure A.7: Database $D_{16} = \Delta^{[2,3,2,1,1]}(D_{15})$, query results $R_{30} = (Q_{86})^{D_{16}}$ and $R_{31} = (Q_{87})^{D_{16}}$, and objects $o_{18} = \text{DECODE}(R_{30}, n)$ and $o_{19} = \text{DECODE}(R_{31}, n)$.

$\begin{array}{|c|c|c|}
\hline
N & T & Y \\
N & T & Y \\
N & Y & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
N & T & Y \\
\hline
\end{array}$

$\begin{array}{|c|c|c|}
\hline
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
T & Y & Y \\
\hline
\end{array}$

$\begin{array}{|c|c|c|}
\hline
Q_{86} & Q_{87} & D_{16} \\
Q_{86} & Q_{87} & D_{16} \\
Q_{86} & Q_{87} & D_{16} \\
Q_{86} & Q_{87} & D_{16} \\
Q_{86} & Q_{87} & D_{16} \\
Q_{86} & Q_{87} & D_{16} \\
Q_{86} & Q_{87} & D_{16} \\
\hline
\end{array}$

184
The key lesson from the above example is that in order for colour diversification to distinguish the results of two encoding queries under \( n \)-equality, the encoded normalized bag must contain two sub-objects whose bag cardinalities are described by polynomials (equations A.11 and A.12 in Section A.2) without any common multiplicative factors. In this case, the choice of a \( (|I_{1,d}| + |I'_{1,d}|) \)-distinguishing diversification point guarantees that the encoded bag cannot be normalized further. Extending this to nested normalized bags requires guaranteeing that there exists a recursive descent of the encoding relation in which at each level the encoded normalized bag contains two sufficiently-different encoded sub-objects. To ensure this, we next devise a canonical database containing canonical objects defined in terms of a system of label-generating prefixes, similar to the one we devised for nested sets.

### A.4.1 Labels, Prefixes, and Canonical Database \( D_Q \)

The requirements of our labelling system in this section are significantly simpler than the requirements in Section A.3; however, it is expedient to adopt the same terminology of label-generating components/sequences/prefixes and simply modify the definitions.

For each level \( i \in [1, d] \), let \( LGC_i \) denote the set of label-generating components at level \( i \), defined as follows.

\[
LGC_i := \{1, 2\} 
\]  
(A.22)

Let \( LGS \) denote the set of label-generating sequences, composed out of label-generating components as follows.

\[
LGS := \{c_1.c_2 \ldots c_d \mid \forall i \in [1, d] : c_i \in LGC_i\} 
\]  
(A.23)

For each integer \( m \in [1, d] \), let \( LGP_m \) denote the set of label-generating prefixes of length \( m \), defined as follows.

\[
LGP_m := \{c_1.c_2 \ldots c_m \mid \forall i \in [1, m]. (c_i \in LGC_i)\} 
\]  
(A.24)

Let \( LGP \) denote the set of all label-generating prefixes. We do not require the notion of a conflict, and so for uniformity of terminology we simply define \( CF-LGC := LGC \) which then entails \( CF-LGP_m = LGP_m, CF-LGP = LGP, \) and \( CF-LGS = LGS \).

As in Section A.3, we define a system of labels in terms of the label-generating components/sequences/prefixes defined above. For each label \( l \), we define the set \( B^l \) and the labelling function \( \alpha^l : B \to B^l \) and the de-labelling function \( \alpha^{-1} : (\bigcup L^B) \to B \) exactly as in Section A.3. For each label-generating sequence \( s \in LGS \), we define the following label-generating function \( \beta_s : B \to (\bigcup L^B) \) which performs a particular
labelling of the variables of $Q$.

$$
\beta_{c_1,c_2...c_d}(x) :=
\begin{cases}
\alpha^{c_1}(x) & \text{if } x \in I_1 \\
\alpha^{c_1-c_2}(x) & \text{if } x \in I_2 \\
\vdots & \\
\vdots & \\
\alpha^{c_1...c_{(d-1)}}(x) & \text{if } x \in I_{(d-1)} \\
\alpha^{c_1...c_d}(x) & \text{if } x \in I_d \\
\alpha^{c_1...c_d}(x) & \text{otherwise}
\end{cases}
$$

(A.25)

We immediately extend function $\beta_s$ to tuples, sets, and atoms and with identity on constants in $C$. Equation A.25 is much simpler than equation A.19, and in this case it suffices that $L := LGP$ (which was not true for nested sets). We observe that $\alpha^{-1}$ again acts as an inverse of $\beta_s$, and for any label-generating sequence $s \in LGS$, again define the function $\beta_p : I_{[1,m]} \rightarrow (\bigcup B^l)$ as in Section A.3.

We now define the canonical database $D_Q$ in two stages. First, we form the database $D_Q^{\text{pre}}$ in terms of the label-generating sequences just defined, analogous to the canonical database used for nested sets (equation A.20).

$$
D_Q^{\text{pre}} := \bigcup_{s \in CF-LGS} \beta_s(\text{body}_Q)
$$

(A.26)

Next, we let $\bar{r}$ be any $(|I_{[1,d]}|+|I'_{[1,d]}|)$-distinguishing diversification point for $\text{adom}(D_Q^{\text{pre}})$ and we use colour-diversification to define canonical database $D_Q$, as we did with nested bags (equation A.13).

$$
D_Q := \Delta \bar{r}(D_Q^{\text{pre}})
$$

(A.27)

Observe that by first whitewashing and then de-labelling $D_Q$ we re-obtain $\text{body}_Q$.

$$
\alpha^{-1} \circ \delta^{-1}(D_Q) \models \text{body}_Q
$$

(A.28)

**Lemma A.4.1** Given

- any integer $i \in [1, d]$, 
- any embedding $\gamma \in \Gamma$, and 
- any label-generating sequence $s \in LGS$;

if $\gamma \left(I_{[1,d]} \setminus (I_i \setminus \mathcal{V})\right) = \beta_s \left(I_{[1,d]} \setminus (I_i \setminus \mathcal{V})\right)$ and $\delta^{-1} \circ \gamma(I_i \setminus \mathcal{V}) \neq \beta_s(I_i \setminus \mathcal{V})$ 

then $\alpha^{-1} \circ \delta^{-1} \circ \gamma(I_i \setminus \mathcal{V}) \neq (I_i \setminus \mathcal{V})$.

**Proof.** Assume w.l.o.g. that $s = c_1 \ldots c_d$. Assume to the contrary that

- $\gamma \left(I_{[1,d]} \setminus (I_i \setminus \mathcal{V})\right) = \beta_s \left(I_{[1,d]} \setminus (I_i \setminus \mathcal{V})\right)$, 
- $\delta^{-1} \circ \gamma(I_i \setminus \mathcal{V}) \neq \beta_s(I_i \setminus \mathcal{V})$, and 
- $\alpha^{-1} \circ \delta^{-1} \circ \gamma(I_i \setminus \mathcal{V}) = (I_i \setminus \mathcal{V})$.

Let $j$ be any integer such that $I_{i,j} \notin \mathcal{V}$. Variable $I_{i,j}$ is a core index (assumption [A2]), and so by Definition 3.1.11 (with the assumption $\bar{s} = n^d$) and Theorem 3.1.10 it follows that
1. \(|(I_i \cap \mathcal{V}) \cup I_{[i+1,d]}| > 0\), and
2. set \(I_{[1,i-1]}\) is not a strong \((I_{i,j}, (I_i \cap \mathcal{V}) \cup I_{[i+1,d]}-)\)-articulation set of hypergraph \(H^Q\). In other words, if we delete vertices \(I_{[1,i-1]}\) from hypergraph \(H^Q\), the remaining hypergraph must contain an \((I_{i,j}, I)\)-path for some variable \(I \in (I_i \cap \mathcal{V}) \cup I_{[i+1,d]}\).

Now consider the function \(\alpha^{-1} \circ \delta^{-1} \circ \gamma\). Because whitewashing function \(\delta^{-1}\) acts as the identity over \(\text{colour}_1\) and because \(\alpha^{-1}\) is the inverse of \(\beta_s\), it follows that \(\alpha^{-1} \circ \delta^{-1} \circ \gamma (I_{[1,d]} \setminus (I_i \setminus \mathcal{V})) = (I_{[1,d]} \setminus (I_i \setminus \mathcal{V}))\). Combined with the above assumption and with equation A.28, this implies that \(\alpha^{-1} \circ \delta^{-1} \circ \gamma (I_{[1,d]} - I_{[1,i-1]} \cup I_i \cap \mathcal{V}) = (I_{[1,d]} \setminus (I_i \setminus \mathcal{V}))\) and \(\alpha^{-1} \circ \delta^{-1} \circ \gamma (\text{body}_Q) \subseteq \text{body}_Q\). Because \(\text{body}_Q\) is minimal (assumption [A3]), it follows from Theorem 3.1.2 that \(\alpha^{-1} \circ \delta^{-1} \circ \gamma\) is a bijection from \(\mathcal{B}\) onto \(\mathcal{B}\); hence, hypergraphs \(H^Q\) and \(\gamma(H^Q)\) must be isomorphic.

Now consider deleting vertices \(\gamma(I_{[1,i-1]}\) from hypergraph \(\gamma(H^Q)\). By isomorphism of \(H^Q\) and \(\gamma(H^Q)\) the remaining hypergraph contains a path between vertices \(\gamma(I_{i,j})\) and \(\gamma(I)\). By the bijectivity of \(\alpha^{-1} \circ \delta^{-1} \circ \gamma\), all of the remaining vertices are labelled copies of either non-index variables or variables in \(I_{[1,d]}\)—and so by equations A.27 and A.25 all of the other vertices in the same connected component as \(\gamma(I)\) (including \(\gamma(I_{i,j})\)) must have labels that begin with \(c_1 \ldots c_i\). By \(\alpha^{-1} \circ \delta^{-1} \circ \gamma (I_i \setminus \mathcal{V}) = (I_i \setminus \mathcal{V})\), conclude that \(\delta^{-1} \circ \gamma (I_{i,j}) \in \beta_s (I_i \setminus \mathcal{V})\).

Repeating this argument for each value of \(j \in [1, |I_i|]\) such that \(I_{i,j} \notin \mathcal{V}\), and combining it with \(\alpha^{-1} \circ \delta^{-1} \circ \gamma (I_i \setminus \mathcal{V}) = (I_i \setminus \mathcal{V})\), conclude the contradiction \(\delta^{-1} \circ \gamma (I_i \setminus \mathcal{V}) = \beta_s (I_i \setminus \mathcal{V})\). \qed

### A.4.2 Canonical Objects

Having just defined the canonical database \(D_Q\), we now define the *canonical objects* encoded within \(\Gamma\) exactly the same as we defined them for nested sets (Definition A.3.1). That is, the canonical object \(o_p\) for prefix \(p \in \text{LGP}_m\) is defined as follows.

\[
o_p := \text{DECODE}(\Gamma [\beta_p (\overline{I}_{[1,m]}), \overline{s}_{[m+1,d]})
\]

Observe that given any integer \(m \in [1, d - 1]\) and any prefix \(p \in \text{LGP}_m\), canonical object \(o_p\) always contains canonical sub-objects \(o_{p,1}\) and \(o_{p,2}\).

### A.4.3 Pseudo-Trivial Certificate Nodes

We now show that sub-relations of \(\Gamma\) that encode canonical objects can only be equated to sub-relations of \(\Phi\) by pseudo-trivial certificate nodes.

**Lemma A.4.2** Given

- any label-generating sequence \(s \in \text{LGS}\),
- any \(\overline{s}\)-certificate that proves \(\Gamma \preceq \overline{s} \Phi\), and
- any level \(m \in [1, d]\);
every certificate node involving sub-relation \( \Gamma[\beta_s(\overline{I}_{[1,m-1]}))] \) is potentially pseudo-trivial w.r.t. \( \beta_s(\overline{I}_m) \).

**Proof.** The proof is by induction on \( m \). First, however, note that we have never constrained which query was chosen as \( Q \) and which was chosen as \( Q' \) (i.e., which query should be used to create \( D_Q \)). W.l.o.g. assume that if \( Q \) and \( Q' \) do not have the same number of index variables at every level, then \( Q \) was chosen so that \( |\mathcal{I}| > |\mathcal{I}'| \) at the first level \( l \) upon which \( Q \) and \( Q' \) disagree, starting at level \( d \) and counting down. Also, assume \( s = c_1 \ldots c_d \) and without loss of generality assume that \( c_m = 1 \) (the proof is symmetric if \( c_m = 2 \)).

**Base Case:** Suppose \( m = d \). Define prefix \( p := c_1 \ldots c_{(d-1)} \). If \( |\mathcal{I}_d| = 0 \) then every certificate node at level \( d \) is trivially pseudo-trivial; therefore, assume that \( |\mathcal{I}_d| > 0 \). The proof is lengthy, and so we will break it into numbered steps for readability.

1. Let \( n^a \) be any certificate node proving \( \Gamma[\overline{a}] \models_{\delta_d} \Phi[\overline{a}'] \) for \( \overline{a} := \beta_s(\overline{I}_{[1,d-1]})) \) and some \( \overline{a}' \in \text{adom}(\overline{I}_{[1,d-1]}, \Phi) \). Relation \( \Gamma[\overline{a}] \) encodes canonical object \( o_p \), which contains canonical sub-objects \( o_{p_1} \) and \( o_{p_2} \). Let sets \( S_{o_{p_1}} \) and \( S_{o_{p_2}} \) be defined as in equation A.10, containing the tuples of non-output values that index sub-objects \( o_{p_1} \) and \( o_{p_2} \), respectively, within \( \Gamma[\overline{a}] \). Define \( S'_{o_{p_1}} \) and \( S'_{o_{p_2}} \) analogously for \( \Phi[\overline{a}'] \). Define polynomials \( f_{S_{o_{p_1}}}, f_{S_{o_{p_2}}}, f_{S'_{o_{p_1}}}, \) and \( f_{S'_{o_{p_2}}} \) as in equations A.11 and A.12.

2. By definition of \( \Gamma[\overline{a}] \), every assignment \( \gamma \in \Gamma[\overline{a}] \) satisfies

   \[
   \gamma(\overline{I}_{[1,d-1]}) = \beta_{p_1}(\overline{I}_{[1,d-1]}) = \beta_s(\overline{I}_{[1,d-1]})
   \]  

   while by definition of canonical object \( o_{p_1} \) every \( \gamma \in \Gamma[\overline{a}]|_{o_{p_1}} \) satisfies the following.

   \[
   \gamma(\mathcal{I}_d \cap \mathcal{V}) = \beta_{p_1}(\mathcal{I}_d \cap \mathcal{V}) = \beta_s(\mathcal{I}_d \cap \mathcal{V})
   \]  

   Tuples in \( S_{o_{p_1}} \) are formed by projecting \( \overline{I}_d \) from the assignments in \( \Gamma[\overline{a}]|_{o_{p_1}} \) (see equation A.10, and recall \( \mathcal{J}_d := \mathcal{I}_d \setminus \mathcal{V} \)). Therefore, there cannot exist a tuple \( t \in S_{o_{p_1}} \) such that

   \( a \to 1 \circ \delta^{-1}(t) \) contains all of the variables in \( \mathcal{J}_d \), and

   (b) the label \( p_2 \) occurs within \( t \).

   If \( t \) did exist, then it would correspond to an assignment \( \gamma \) satisfying

   \[
   \delta^{-1} \circ \gamma(\mathcal{J}_d) = \beta_{p_1}(\mathcal{J}_d) = \beta_s(\mathcal{J}_d)
   \]

   which contradicts Lemma A.4.1. In contrast, the design of \( D_Q \) implies that \( \beta_{p_1}(\overline{I}_d) \in \text{adom}(\overline{I}_d, \Gamma[\overline{a}]) \), and the definition of canonical object \( o_{p_1} \) further implies \( \beta_{p_1}(\overline{I}_d) \in \text{adom}(\overline{I}_d, \Gamma[\overline{a}]|_{o_{p_1}}) \); therefore, \( S_{o_{p_1}} \) does contain the tuple \( \beta_s(\overline{J}_d) = \beta_{p_1}(\overline{J}_d) \), for some ordering \( \overline{J}_d \) of \( \mathcal{J}_d \).
3. Each monomial in $f_{S_{op,1}}$ (resp., $f_{S_{op,2}}$) corresponds to a collection of values occurring within a tuple in $S_{op,1}$ (resp., $S_{op,2}$). Define polynomial $f_{GCD} := \text{GCD}(f_{S_{op,1}}, f_{S_{op,2}})$. By definition of greatest common divisor, there exists two polynomials $f_1, f_2$ with $\text{GCD}(f_1, f_2) = 1$ that satisfy $f_{S_{op,1}} = f_1 \times f_{GCD}$ and $f_{S_{op,2}} = f_2 \times f_{GCD}$. We will now prove by contradiction that $\text{degree}(f_{GCD}) = 0$. Suppose that $\text{degree}(f_{GCD}) = k > 0$. Because $S_{op,1}$ contains tuple $\beta_{p,1}(\mathcal{J}_d)$, polynomial $f_{S_{op,1}}$ contains a monomial corresponding to the values $\beta_{p,1}(\mathcal{J}_d)$. Therefore, there must exist some subset $J'_d \subseteq J_d$ with $|J'_d| = k$ such that polynomial $f_{GCD}$ contains a monomial corresponding to values $\beta_{p,1}(J'_d)$. By symmetry of $\mathcal{D}_Q$, polynomial $f_{GCD}$ must be symmetric between labels $p,1$ and $p,2$ which means that $f_{GCD}$ also contains a monomial corresponding to values $\beta_{p,2}(J'_d)$. However, this implies that $f_{S_{op,1}}$ contains a monomial corresponding to values $\beta_{p,2}(J'_d \cup \beta_{p,2}(J'_d))$, which further implies the existence of tuple $t \in S_{op,1}$ with $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\mathcal{J}_d) = \mathcal{J}_d$ and $\delta^{-1} \circ \gamma(\mathcal{J}_d) \neq \beta_{p,1}(\mathcal{J}_d) = \beta_s(\mathcal{J}_d)$—which contradicts the previous paragraph. Hence, $\text{GCD}(f_{S_{op,1}}, f_{S_{op,2}})$ is a constant.

4. Because $\Gamma[\overline{a}]$ and $\Phi[\overline{a'}]$ encode the same normalized bag, we know

$$\frac{f_{S_{op,1}}(\overline{r})}{f_{S_{op,2}}(\overline{r})} = \frac{f_{S_{op,1}}'(\overline{r})}{f_{S_{op,2}}'(\overline{r})}$$

and so

$$f_{S_{op,1}}(\overline{r}) \times f_{S_{op,1}}'(\overline{r}) = f_{S_{op,2}}(\overline{r}) \times f_{S_{op,2}}'(\overline{r}) \quad (A.33)$$

follows. During construction of database $\mathcal{D}_Q$ point $\overline{r}$ was chosen to be $(|\mathcal{I}_d| + |\mathcal{I}'_d|)$-distinguishing, while each side of equation $A.33$ is a polynomial of degree $(|\mathcal{I}_d| + |\mathcal{I}'_d|)$; therefore the two sides must be the same polynomial.

$$f_{S_{op,1}} \times f_{S_{op,2}}' = f_{S_{op,2}} \times f_{S_{op,1}}' \quad (A.34)$$

5. From Lemma A.1.1 we know that $|\mathcal{I}_d \cap \mathcal{V}| = |\mathcal{I}'_d \cap \mathcal{V}'|$. From equation $A.30$ it follows that every $\phi \in \Phi[\overline{a}']_{op,1}$ must satisfy $\phi(\mathcal{I}'_d \cap \mathcal{V}') = \beta_{p,1}(\mathcal{I}_d \cap \mathcal{V})$.

6. We know $|\mathcal{I}_d| \geq |\mathcal{I}'_d|$ by choice of $Q$, and so by $|\mathcal{I}_d \cap \mathcal{V}| = |\mathcal{I}'_d \cap \mathcal{V}'|$ conclude the following.

$$\text{degree}(f_{S_{op,1}}) = |\mathcal{I}_d \setminus \mathcal{V}| \geq |\mathcal{I}'_d \setminus \mathcal{V}'| = \text{degree}(f_{S_{op,2}}) \quad (A.35)$$

Because $\text{GCD}(f_{S_{op,1}}, f_{S_{op,2}})$ is a constant, from equation $A.34$ conclude that $f_{S_{op,1}}$ must be a constant multiple of $f_{S_{op,2}}'$. Therefore, for each tuple $t \in S_{op,1}$ there exists a permutation of $t$ within $S_{op,1}$, and vice versa.

7. By the definitions of $\overline{a}$ and canonical object $o_{p,1}$ By $\beta_{s}(\mathcal{I}_{[1,d-1]}) = \overline{a}$ and $\beta_{s}(\mathcal{I}_d) = \beta_{p,1}(\mathcal{I}_d)$, we know that $\Gamma[\overline{a}\beta_{s}(\mathcal{I}_d)]$ is a sub-relation of $\Gamma[\overline{a}]$ that encodes $o_{p,1}$. From the previous two steps it follows that there must be some tuple $\overline{x'} \in \text{adom}(\mathcal{I}'_d, \Phi[\overline{a'}])$ such that $\Phi[\overline{a'}]$ encodes $o_{p,1}$ and $\overline{x'}$ is
Lemma A.4.3 If $\overline{s} = n^d$ then there exists index-covering homomorphisms in both directions between $Q$ and $Q'$.

Proof. Choose any label-generating sequence $s \in LGS$. Consider any $n^d$-certificate proving $\Gamma \models \Phi$. By Lemma A.4.2, the root node of the certificate is potentially pseudo-trivial w.r.t. $\beta_s(\overline{I}_1)$; therefore, re-arrange the mappings within the node until it is pseudo-trivial w.r.t. $\beta_s(\overline{I}_1)$. By recursing into any child $n^{d-1}$-certificate involving sub-relation $\Gamma[\beta_s(\overline{I}_1)]$ and repeating this procedure, we eventually arrive at a tuple node proving $\Gamma[\beta_s(\overline{I}_{[1,d]})] \models \Phi[\overline{s}_{[1,d]}]$ for some $\overline{s}_{[1,d]} \in \text{dom}(\overline{I}_{[1,d]}, \Phi)$ such that for each $i \in [1,d]$, $\overline{s}_i$ is a permutation of $\beta_s(\overline{I}_i)$. Choose any $\phi \in \Phi[\overline{s}_{[1,d]}]$.

Define $h := \alpha^{-1} \circ \delta^{-1} \circ \phi$. Then, $h$ is a mapping from $B'$ to $\alpha^{-1} \circ \delta^{-1}(\text{dom}(D_Q)) = B \cup C_Q$ that satisfies the following properties.

1. $h(\text{body}_{Q'}) = \alpha^{-1} \circ \delta^{-1} \circ \phi(\text{body}_{Q'}) \subseteq \alpha^{-1} \circ \delta^{-1}(D_Q) = \text{body}_{Q}$
2. $h(\overline{V}) = \alpha^{-1} \circ \delta^{-1} \circ \phi(\overline{V}) = \alpha^{-1} \circ \delta^{-1} \circ \beta_s(\overline{V}) = \overline{V}$
3. $\forall i \in [1,d], h(\overline{I}_i) = \alpha^{-1} \circ \delta^{-1} \circ \phi(\overline{I}_i) = \alpha^{-1} \circ \delta^{-1} \circ \beta_s(\overline{I}_i) = \overline{I}_i$

which follows because $\phi(\overline{I}_i) = \overline{s}_i$ is a permutation of $\beta_s(\overline{I}_i)$.
Hence, $h$ satisfies Definition 3.2.1. The proof is analogous for $h'$ from $Q$ to $Q'$.

\[ \square \]

### A.5 Arbitrary Signatures

We can combine the proof from the previous three sections to obtain a general proof of Theorem 3.2.4 that handles signatures denoting arbitrary combinations of nested sets, bags, and normalized bags. Like the proof for normalized bags, the general proof relies on both colour-diversification and a labelling system in respect to which certain objects are deemed to be canonical.

We use the same vocabulary of label-generating components/prefixes/sequences as we used in both Sections A.3 and A.4. For any level $m$ in which $\xi_m = s$, we define $LGC_m$, $CF-LGC_m$, $LGP_m$, and $CF-LGP_m$ as we defined them in Section A.3. When $\xi_m = n$ we instead use the corresponding definitions from Section A.4. When $\xi_m = b$ we could use either labelling system, but we prefer the one from Section A.4 because it is simpler. The set of label-generating sequences $LGS$ is defined relative to the interleaved definitions of $LGC_m$, and so the label-generating function $\beta_s : \mathcal{B} \to (\bigcup \mathcal{B}^\ell)$ is formed from equations A.19 and A.25 by selecting the labelling system appropriate to the semantics of each level. The canonical database $D_Q$ is then constructed using both colour-diversification and labelling (exactly as in Section A.4), and the same definition of canonical objects is used as we used for both sets and normalized bags.

We can prove the existence of (potentially) pseudo-trivial certificate nodes inductively, with the inductive argument varying depending upon the semantics of the level. When $\xi_m = b$, we use Lemma A.2.9 to argue that nodes involving $\Gamma[\beta_s(\overline{I}_{1,m-1})]$ are potentially pseudo-trivial w.r.t. $\beta_s(\overline{I}_m)$. When $\xi_m = s$, the argument uses Lemmas A.3.4 and A.3.5, and when $\xi_m = n$ the argument uses Lemma A.4.2. The construction of index-covering homomorphisms is then identical to Lemma A.4.3.
References


