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

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

List of Symbols

Page numbers indicate the location of definitions.

E

F

G

H

I

L

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,[1] 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.

[1] 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:
```
SELECT *
FROM Customer, Order o1, Order o2
WHERE Customer.cid = o1.cid
  AND Customer.cid = o2.cid
```

Q':
```
SELECT *
FROM C ⋈cid O1 ⋈cid O2
```

Abbr	Table Schema	Foreign Keys
C	Customer(<u>cid</u>, cname, ctype)	
O	Order(<u>oid</u>, cid, date)	date:Date(date)
LI	LineItem(<u>oid</u>, <u>lineno</u>, price, qty)	oid:Order(oid)
A	Agent(<u>aid</u>, aname)	
OA	OrderAgent(<u>oid</u>, <u>aid</u>)	oid:Order(oid), aid:Agent(aid)
D	Date(<u>date</u>, qtr)	

Logical (non-Materialized) Views:

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

```
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)

```
SELECT qtr, cid, cname, ctype
FROM C ⋈cid QCO
```

Materialized Views:

OV OrderValues(oid, oval)

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

QAS QuarterlyAgentSales(aid, qtr, ctype, avgOval)

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

QCO QuarterlyCustomerOrders(qtr, cid, ordCount)

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

QMCO QuarterlyMaxCustomerOrders(qtr, ctype, maxOrdCount)

```
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:
```
SELECT AS1.aname, qtr,
       AVG(AS1.oval) AS avgRsale,
       AVG(AS2.oval) AS avgCsale
FROM (AS1 ⋈date D1) ⋈{aid, qtr} (AS2 ⋈date D2)
WHERE AS1.ctype = 'R'
  AND AS2.ctype = 'C'
GROUP BY aid, AS1.aname, qtr
```

Q_2:
```
SELECT A1.aname, qtr,
       AVG(OV1.oval) AS avgRsale,
       AVG(OV2.oval) AS avgCsale
FROM  (C1 ⋈cid O1 ⋈oid OV1 ⋈oid OA1 ⋈aid A1 ⋈date D1) ⋈{aid, qtr}
       (C2 ⋈cid O2 ⋈oid OV2 ⋈oid OA2 ⋈aid A2 ⋈date D2)
WHERE C1.ctype = 'R'
  AND C2.ctype = 'C'
GROUP BY aid, A1.aname, qtr
```

Q_3:
```
SELECT aname, qtr,
       QAS1.avgOval AS avgRsale,
       QAS2.avgOval AS avgCsale
FROM A ⋈aid QAS1 ⋈{aid, qtr} QAS2
WHERE QAS1.ctype = 'R'
  AND QAS2.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 SUM aggregate within `AgentSales` in order to rewrite over two occurrences of the materialized view `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.[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 `QuarterlyAgentSales`, yielding a drastically more efficient execution plan. (Even if view `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 `QuarterlyCustomerOrders` which totals the number of orders by quarter and customer id. She then created the logical view `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 *QuarterlyResidentialCustomer*

[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 MAX or 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 AVG in materialized views we added the appropriate SUM and COUNT expressions to the view and then tested the query both with the original AVG expressions and with manual decompositions into SUM and COUNT.

Q_4:
```
SELECT qtr, MAX(countOrders)
FROM (SELECT qtr, cid, cname,
             (SELECT COUNT(*)
              FROM D ⋈_date O
              WHERE D.qtr = QC.qtr
                AND O.cid = QC.cid
             ) AS countOrders
       FROM (SELECT qtr, cid, cname
             FROM QC
             WHERE ctype = 'R'
             ) AS QuarterlyResidentialCustomer
      ) AS QuarterlyResCustWithOrderCounts
GROUP BY qtr
```

Q_5:
```
SELECT qtr, MAX(countOrders)
FROM (SELECT qtr, cid, cname,
             (SELECT COUNT(*)
              FROM QCO_2
              WHERE QCO_2.qtr = QCO_1.qtr
                AND QCO_2.cid = QCO_1.cid
             ) AS countOrders
       FROM C ⋈_cid QCO_1
       WHERE ctype = 'R'
      ) AS QuarterlyResCustWithOrderCounts
GROUP BY qtr
```

Q_6:
```
SELECT qtr, maxOrdCount
FROM QMCO
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 $\texttt{Bag}(c)$—is the bag of elements that occur in c, while the *set projection of* c—$\texttt{Set}(c)$—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—$\texttt{NormBag}(c)$—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.

$$B_1 := \{\!|\, x, x, y, y, y \,|\!\} \qquad B_3 := \{\!|\, x, x, y \,|\!\}$$
$$B_2 := \{\!|\, x, x, x, x, y, y, y, y, y, y \,|\!\} \qquad B_4 := \{\!|\, x, x, x, x, y, y \,|\!\}$$

The first and second have the same normalized bag projection

$$N_1 := \{\!\{\, x, x, y, y, y \,\}\!\} = \texttt{NormBag}(B_1) = \texttt{NormBag}(B_2)$$

as do the third and fourth.

$$N_2 := \{\!\{\, x, x, y \,\}\!\} = \texttt{NormBag}(B_3) = \texttt{NormBag}(B_4)$$

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

$$\texttt{Set}(B_1) = \texttt{Set}(B_2) = \texttt{Set}(B_3) = \texttt{Set}(B_4) = \texttt{Set}(N_1) = \texttt{Set}(N_1) = \{\, x, y \,\}$$

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 `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*)* τ *is syntactically defined by the following recursive grammar:*

$$\tau \;:=\; \texttt{dom} \;\mid\; \{\,\tau\,\} \;\mid\; \{\!|\,\tau\,|\!\} \;\mid\; \{\!\{\,\tau\,\}\!\} \;\mid\; \langle\,\tau, \ldots, \tau\,\rangle$$

We call sort $\texttt{dom}$ an *atomic* sort; sorts $\{\,\tau\,\}$, $\{\!|\,\tau\,|\!\}$, and $\{\!\{\,\tau\,\}\!\}$ *collection* sorts; and sort $\langle\,\tau, \ldots, \tau\,\rangle$ 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* τ *is defined recursively as follows.*

$$\begin{aligned}
\texttt{depth}(\texttt{dom}) &= 0 \\
\texttt{depth}(\{\,\tau\,\}) &= 1 + \texttt{depth}(\tau) \\
\texttt{depth}(\{\!|\,\tau\,|\!\}) &= 1 + \texttt{depth}(\tau) \\
\texttt{depth}(\{\!\{\,\tau\,\}\!\}) &= 1 + \texttt{depth}(\tau) \\
\texttt{depth}(\langle\,\tau_1, \ldots, \tau_k\,\rangle) &= \max(0, \texttt{depth}(\tau_1), \ldots, \texttt{depth}(\tau_k))
\end{aligned}$$

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

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.

$$\begin{aligned}
\tau_1 &:= \langle\,\texttt{dom}, \texttt{dom}, \texttt{dom}\,\rangle & \tau_2 &:= \{\,\langle\,\texttt{dom}, \texttt{dom}\,\rangle\,\} \\
\tau_3 &:= \{\,\langle\,\langle\,\texttt{dom}\,\rangle, \langle\,\texttt{dom}, \texttt{dom}\,\rangle\,\rangle\,\} & \tau_4 &:= \{\!\{\,\{\,\langle\,\texttt{dom}\,\rangle\,\}\,\}\!\} \\
\tau_5 &:= \{\,\{\!|\,\langle\,\texttt{dom}\,\rangle\,|\!\}\,\} & \tau_6 &:= \{\,\langle\,\texttt{dom}, \{\!|\,\texttt{dom}\,|\!\}\,\rangle\,\} \\
\tau_7 &:= \{\!|\,\langle\,\texttt{dom}, \{\,\langle\,\texttt{dom}\,\rangle\,\}\,\rangle\,|\!\} & \tau_8 &:= \{\,\langle\,\{\!|\,\langle\,\{\,\langle\,\texttt{dom}\,\rangle\,\}\,\rangle\,|\!\}, \{\,\langle\,\texttt{dom}\,\rangle\,\}\,\rangle\,\} \\
\tau_9 &:= \{\,\langle\,\{\!|\,\langle\,\{\!\{\,\langle\,\texttt{dom}\,\rangle\,\}\!\}\,\rangle\,|\!\}, \{\,\langle\,\{\!|\,\langle\,\texttt{dom}\,\rangle\,|\!\}\,\rangle\,\}, \{\!\{\,\langle\,\texttt{dom}\,\rangle\,\}\!\}\,\rangle\,\}
\end{aligned}$$

An equivalent tree representation is shown in Figure 2.1. Sort τ_1 has depth zero, τ_2 and τ_3 have depth one, τ_4–τ_7 have depth two, and τ_8 and τ_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.

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

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

- Atomic sort $\texttt{dom}$ is non-branching.

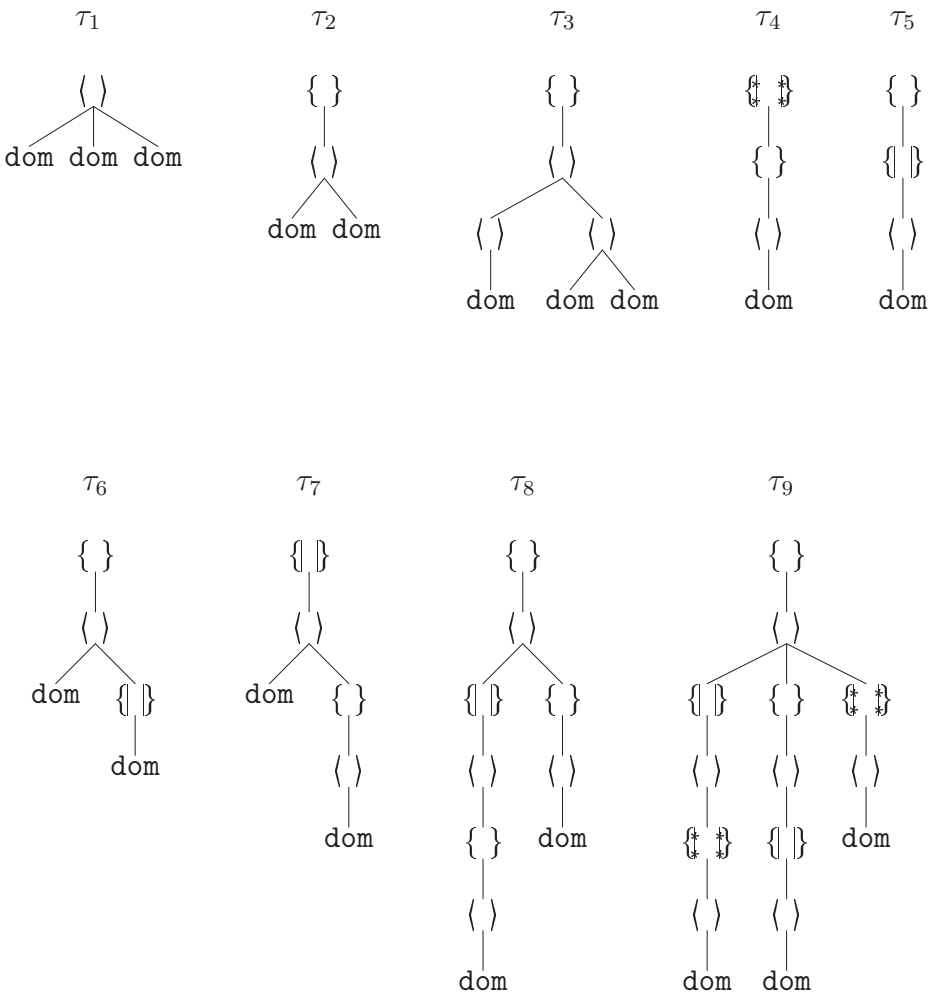


Figure 2.1: Graphical representation of sorts

- Collection sorts $\{\,\tau\,\}$, $\{\!|\,\tau\,|\!\}$, and $\{\!\{\,\tau\,\}\!\}$ are branching if τ is a branching sort; otherwise they are non-branching.
- Tuple sort $\langle\,\tau_1, \ldots, \tau_k\,\rangle$ is branching if either
  1. one of its child sorts τ_i is branching; or
  2. at least two of its child sorts τ_i and τ_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, τ_8 and τ_9 in Figure 2.1 are branching sorts, while τ_1–τ_7 are non-branching sorts. By our convention on the structuring of tuple sorts, within a non-branching tuple sort $\langle\,\tau_1, \ldots, \tau_k\,\rangle$ only τ_k can have depth greater than zero. Hence, non-branching sorts contain only "right deep" nesting of collections, as illustrated by sorts τ_6 and τ_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 τ_1, τ_2, τ_4, and τ_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 $\overline{\S}$ called its *signature*.

Definition 2.1.3 (Signature) *A* semantic indicator $\S$ *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 $\overline{\S}$ *of length d is a finite sequence of semantic indicators $\S_1\S_2\cdots\S_d$.*

Any chain sort of depth d can be represented by a pair $(\overline{\S},\ k)$, where $\overline{\S}$ 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 τ_1, τ_2, τ_4, and τ_5 from Figure 2.1 can be represented as $(\varnothing,\ 3)$, (`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 τ, *denoted* $[\![\,\tau\,]\!]$, *is defined recursively as follows.*

$$
\begin{aligned}
[\![\,\texttt{dom}\,]\!] &= \texttt{dom} \\
[\![\,\{\,\tau\,\}\,]\!] &= \{\,\{\,v_1, \ldots, v_j\,\} \mid j \geq 0, v_i \in [\![\,\tau\,]\!], i \in [1, j]\,\} \\
[\![\,\{\!|\,\tau\,|\!\}\,]\!] &= \{\,\{\!|\,v_1, \ldots, v_j\,|\!\} \mid j \geq 0, v_i \in [\![\,\tau\,]\!], i \in [1, j]\,\} \\
[\![\,\{\!\{\,\tau\,\}\!\}\,]\!] &= \{\,\{\!\{\,v_1, \ldots, v_j\,\}\!\} \mid j \geq 0, v_i \in [\![\,\tau\,]\!], i \in [1, j]\,\} \\
[\![\,\langle\,\tau_1, \ldots, \tau_k\,\rangle\,]\!] &= \{\,\langle\,v_1, \ldots, v_k\,\rangle \mid v_j \in [\![\,\tau_j\,]\!], j \in [1, k]\,\}
\end{aligned}
$$

For each integer $i \geq 0$, we use $\texttt{dom}^i$ to denote the (infinite) set of all finite elements belonging to interpretations of sorts with depth at most i.

$$\texttt{dom}^i := \bigcup_{\tau \in \mathcal{T}^i} [\![\tau]\!] \tag{2.2}$$

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

$$\texttt{dom}^\infty := \bigcup_{\tau \in \mathcal{T}} [\![\tau]\!] \tag{2.3}$$

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

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

$$\begin{aligned}
o_1 &:= \langle\, a, b, 0 \,\rangle \\
o_2 &:= \{\, \langle\, a, 1 \,\rangle, \langle\, a, 2 \,\rangle, \langle\, b, 2 \,\rangle, \langle\, b, 3 \,\rangle \,\} \\
o_3 &:= \{\, \langle\, \langle\, a \,\rangle, \langle\, 2, 5 \,\rangle \,\rangle, \langle\, \langle\, a \,\rangle, \langle\, 3, 4 \,\rangle \,\rangle, \langle\, \langle\, b \,\rangle, \langle\, 3, 4 \,\rangle \,\rangle \,\} \\
o_4 &:= \{\!\{\, \{\, \langle\, 1 \,\rangle, \langle\, 2 \,\rangle \,\}, \{\, \langle\, 1 \,\rangle, \langle\, 2 \,\rangle \,\}, \{\, \langle\, 1 \,\rangle, \langle\, 3 \,\rangle \,\} \,\}\!\} \\
o_5 &:= \{\, \{\!|\, \langle\, 1 \,\rangle, \langle\, 1 \,\rangle, \langle\, 2 \,\rangle \,|\!\}, \{\!|\, \langle\, 1 \,\rangle, \langle\, 2 \,\rangle \,|\!\}, \{\!|\, \langle\, 1 \,\rangle, \langle\, 3 \,\rangle, \langle\, 3 \,\rangle \,|\!\} \,\} \\
o_6 &:= \{\, \langle\, 1, \{\!|\, a, a, c \,|\!\} \,\rangle, \langle\, 2, \{\!|\, a, c \,|\!\} \,\rangle \,\} \\
o_7 &:= \{\!|\, \langle\, a, \{\, \langle\, 1 \,\rangle, \langle\, 2 \,\rangle \,\} \,\rangle, \langle\, b, \{\, \langle\, 1 \,\rangle, \langle\, 3 \,\rangle \,\} \,\rangle, \langle\, c, \{\,\} \,\rangle \,|\!\} \\
o_8 &:= \{\, \langle\, \{\!|\,|\!\}, \{\,\} \,\rangle \,\} \\
o_9 &:= \{\, \langle\, \{\!|\, \langle\, \{\!\{\, \langle\, a \,\rangle, \langle\, c \,\rangle, \langle\, c \,\rangle \,\}\!\} \,\rangle, \\
&\qquad\quad \langle\, \{\!\{\, \langle\, a \,\rangle, \langle\, c \,\rangle, \langle\, c \,\rangle \,\}\!\} \,\rangle, \\
&\qquad\quad \langle\, \{\!\{\,\}\!\} \,\rangle \,|\!\}, \\
&\qquad \{\,\}, \\
&\qquad \{\!\{\, \langle\, 5 \,\rangle \,\}\!\} \,\rangle \,\} \\
o_{10} &:= \{\,\} \\
o_{11} &:= \{\, a, \{\!|\, a, a, c \,|\!\}, c, \{\!|\, a, c \,|\!\} \,\}
\end{aligned}$$

Objects o_1–o_9 are complex objects because they conform to sorts τ_1–τ_9, respectively, from Example 4. Object o_{10} conforms to any sort rooted by a set, including τ_2, τ_3, τ_5, τ_6, τ_8, and τ_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 τ_1–τ_9 from Example 4. Objects o_1–o_7 are all non-branching because they conform to τ_1–τ_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 τ_8. Similarly, o_9 is a branching object because any sort to which it conforms is necessarily branching, including τ_9. Object o_{10} is a *non-branching* object even though it conforms to the *branching* sort τ_9—it also conforms to the *non-branching* sort τ_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 τ_1, τ_2, τ_4, τ_5, and τ_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.

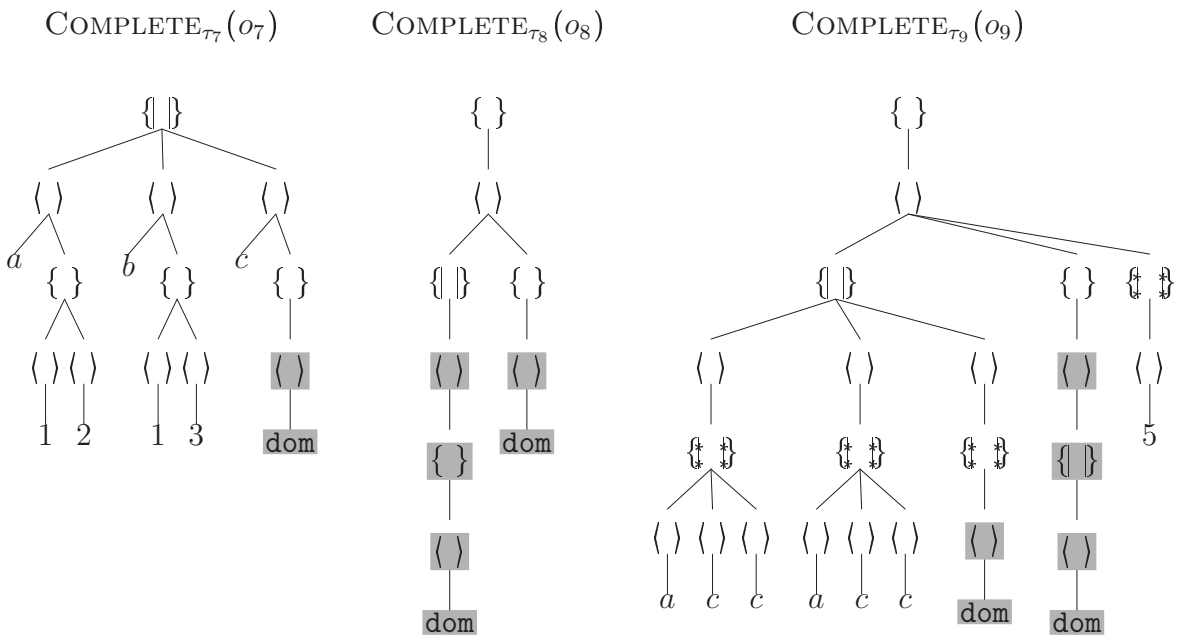


Figure 2.2: Completion of objects o_7, o_8, and o_9

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

Example 6 Consider the (non-trivial) incomplete objects o_7, o_8, and o_9 from Example 5 that conform to sorts τ_7, τ_8, and τ_9, respectively, from Example 4. Figure 2.2 illustrates the completion of each object with respect to its associated sort.

Object $\textsc{Complete}_\tau(o)$ may contain "shaded" delimiter pairs {| |}, { }, {# #}, or ⟨ ⟩; it also may contain the symbol `dom` which is used as an atomic value. If $\textsc{Complete}_\tau(o)$ 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 `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.

$$\begin{aligned}
[\![\, \texttt{dom} \,]\!] &= \texttt{dom} \cup \{\texttt{dom}\} \\
[\![\, \{\, \tau \,\} \,]\!] &= \{\{\, v_1, \ldots, v_j \,\} \mid j \geq 0, v_i \in [\![\, \tau \,]\!], i \in [1, j]\} \\
&\quad \cup \{\{\, v_1, \ldots, v_j \,\} \mid j \geq 0, v_i \in [\![\, \tau \,]\!], i \in [1, j]\} \\
[\![\, \langle\, \tau_1, \ldots, \tau_k \,\rangle \,]\!] &= \{\langle\, v_1, \ldots, v_k \,\rangle \mid v_j \in [\![\, \tau_j \,]\!], j \in [1, k]\} \\
&\quad \cup \{\langle\, v_1, \ldots, v_k \,\rangle \mid v_j \in [\![\, \tau_j \,]\!], j \in [1, k]\}
\end{aligned}$$

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

$$\textsc{Complete}_\tau : [\![\, \tau \,]\!] \to [\![\, \tau \,]\!]$$

and so, by Definition (2.1.5), $\textsc{Complete}_\tau(o)$ is a complex object.

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

1. *The root node is always unshaded.*
2. *The only shaded atomic value is* `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 τ 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 $\textsc{Chain}(o)$ is the invocation of Distribute on line 10 to remove tuple sorts with arity of two or more. The 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 `dom` as an atomic value, any sort τ can be viewed as a complete complex object (τ conforms to itself, and $\textsc{Complete}_\tau(\tau) = \tau$), 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

if o is atomic
  then return $\langle\, o \,\rangle$
elseif $o = [\, o_1, \ldots, o_n \,]$ ▷ Where $[\,\cdot\,]$ is any (possibly shaded) collection type
  then return $[\, \textsc{Chain}(o_1), \ldots, \textsc{Chain}(o_n) \,]$ ▷ preserving collection type $[\,\cdot\,]$
elseif $o = \langle\, \rangle$ or $o = \langle\!\langle\, \rangle\!\rangle$
  then return $\langle\, \rangle$
elseif $o = \langle\, o_1 \,\rangle$ or $o = \langle\!\langle\, o_1 \,\rangle\!\rangle$
  then return $\textsc{Chain}(o_1)$
elseif ($o = \langle\, o_1, \ldots, o_n \,\rangle$ or $o = \langle\!\langle\, o_1, \ldots, o_n \,\rangle\!\rangle$) and $n > 1$
  then return $\textsc{Distribute}(\textsc{Chain}(o_1), \textsc{Chain}(\langle\, o_2, \ldots, o_n \,\rangle))$

DISTRIBUTE(o_a, o_b)

▷ **Input:** *chain* object o_a of sort $(\overline{\S}^a, k)$
  Assume that o_a is a tree whose m leaves are
  the k-ary tuples $\langle\, a_1^1, \ldots a_k^1 \,\rangle$, …, $\langle\, a_1^m, \ldots a_k^m \,\rangle$
▷ **Input:** *chain* object o_b of sort $(\overline{\S}^b, l)$
  Assume that o_b is a tree whose n leaves are
  the l-ary tuples $\langle\, b_1^1, \ldots b_l^1 \,\rangle$, …, $\langle\, b_1^n, \ldots b_l^n \,\rangle$
▷ **Output:** chain object of sort $(\overline{\S}^a \cdot \overline{\S}^b, k + l)$
  formed by distributing o_b over each leaf of o_a
  and pushing down atomic values

$o \leftarrow$ copy of o_a
foreach $i \in [1, m]$
  do $o^i \leftarrow$ copy of o_b
    foreach $j \in [1, n]$
      do substitute tuple $\langle\, a_1^i, \ldots, a_k^i, b_1^j, \ldots, b_l^j \,\rangle$
        for tuple $\langle\, b_1^j, \ldots, b_l^j \,\rangle$ within o^i
    substitute o^i for tuple $\langle\, a_1^i, \ldots, a_k^i \,\rangle$ within o
return o

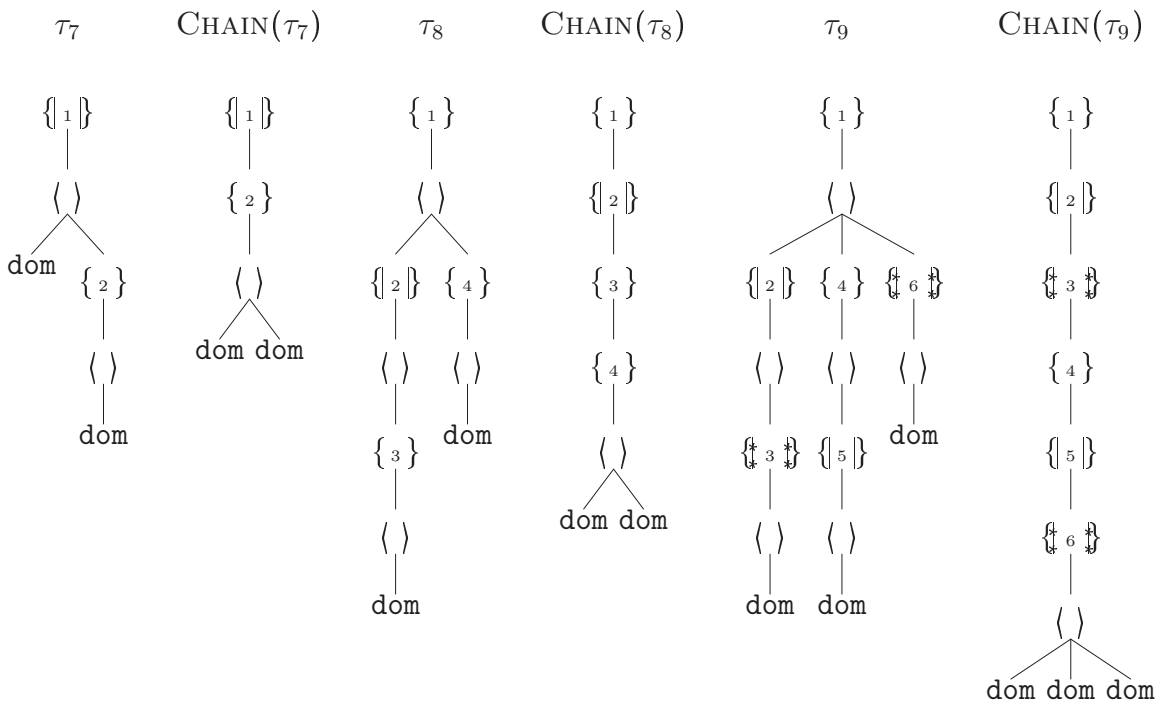


Figure 2.3: Linearization of sorts τ_7, τ_8, and τ_9

Observation 2.1.7 *If o is a complete object conforming to sort τ, then* CHAIN(o) *is a complete chain object conforming to sort* CHAIN(τ).

Definition 2.1.8 (Object Linearization) *Given an object o conforming to sort τ, the* linearization of o with respect to τ *is the complete chain object given by the following function* LINEARIZE$_\tau : \tau \to$ CHAIN(τ).

$$\text{LINEARIZE}_\tau(o) := \text{CHAIN} \circ \text{COMPLETE}_\tau(o) \tag{2.4}$$

Given an arbitrary sort τ, we may at times refer to the chain sort CHAIN(τ) as the *linearization of* τ. This is consistent with Definition 2.1.8, because if we treat τ as an object of sort τ to be linearized, we deduce the following.

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

Example 7 Sorts τ_7, τ_8, and τ_9 are shown in Figure 2.3 along with their linearizations CHAIN(τ_7), CHAIN(τ_8), and CHAIN(τ_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 τ_7, τ_8, and τ_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.

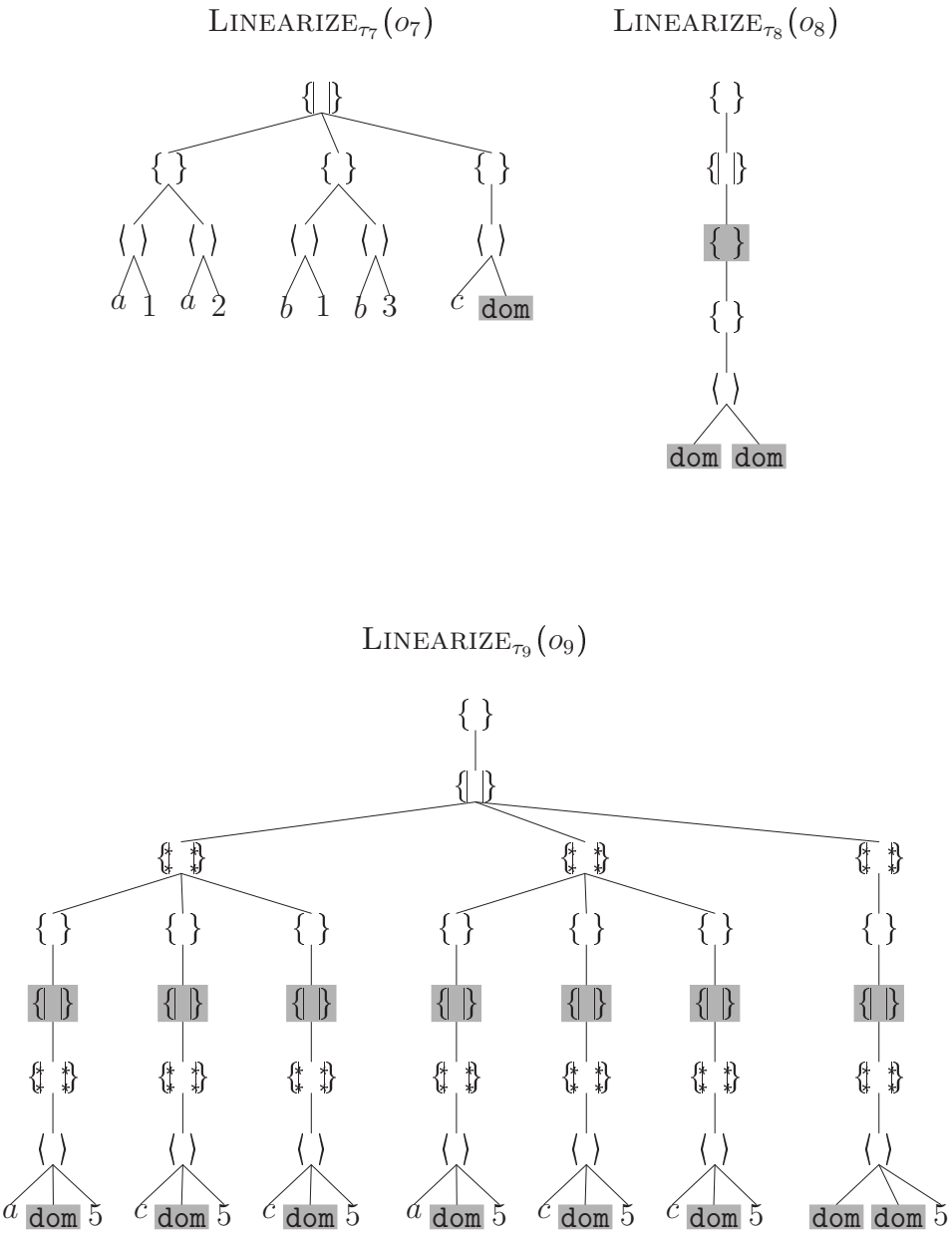


Figure 2.4: Linearization of objects o_7, o_8, and o_9

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 $\textsc{Distribute}(o_a, o_b)$ we can reconstruct both o_a and o_b because either both are trivial chain objects (in which case $\textsc{Distribute}(o_a, o_b)$ 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 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 $\textsc{Complete}_{\tau_9}(o_9)$ (cf. Figure 2.2) to $\textsc{Linearize}_{\tau_9}(o_9)$ (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 $\textsc{Complete}_\tau(o)$ were inserted by the completion of o with respect to some sort τ, *and* τ 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 τ not containing any zero-ary tuple sorts,* $\textsc{Linearize}_\tau$ *has a retraction.*

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

$$o = o' \iff \textsc{Linearize}_\tau(o) = \textsc{Linearize}_\tau(o')$$

Finally, we note that the properties in Observation 2.1.6 may be violated by application of Algorithm 1. For example, linearized objects $\textsc{Linearize}_{\tau_8}(o_8)$ and $\textsc{Linearize}_{\tau_9}(o_9)$ 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 τ to which it conforms, linearized object* $\textsc{Linearize}_\tau(o)$ *obeys the following properties:*

1. *The root node is always unshaded.*
2. *The only shaded atomic value is* `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 $\textsc{Complete}_\tau(o)$ (by Observation 2.1.6) and are preserved by Algorithm 1 (although preservation of the third property is not obvious and relies upon

COMPLETE$_\tau(o)$ satisfying all of the properties in Observation 2.1.6). The fourth property holds because procedure 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" (COCQL$^{\mathcal{F}}$)[2] which is intended to model a specific subclass of SQL-92 [63] queries containing aggregation. As such, COCQL$^{\mathcal{F}}$ 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 COCQL$^{\mathcal{F}}$, 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 COCQL$^{\mathcal{F}}$ that is incapable of constructing non-trivial incomplete objects (i.e., objects containing empty nested collections), which we will denote as COCQL$^{\mathcal{F}-}$.

Data Model

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

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

$$T(A_1 : \tau_1, \ldots, A_k : \tau_k) \tag{2.5}$$

for some $k \geq 0$, where T is a table name from tname*, $A_1, \ldots, A_k$ are distinct attribute names from* aname*, and $\tau_1, \ldots, \tau_k$ are sorts. If $\tau_1, \ldots, \tau_k$ are all* 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* $(\mathbb{S}(T), \mathbb{T}(T))$ *where* $\mathbb{S}(T)$ *is a table schema* $T(A_1 : \tau_1, \ldots, A_k : \tau_k)$ *and* $\mathbb{T}(T)$ *is a* finite bag-instance *of* $\mathbb{S}(T)$*.*

$$\mathbb{T}(T) \in [\![\, \{\!|\, \langle\, \tau_1, \ldots, \tau_k \,\rangle \,|\!\} \,]\!]$$

We call T flat if $\mathbb{S}(T)$ is a flat schema.

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

$$\mathbb{T}(R) \in [\![\, \{\, \langle \overbrace{\texttt{dom}, \ldots, \texttt{dom}}^{k} \rangle \,\} \,]\!]$$

[2]Pronounced "cockle".

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

$$\texttt{dom}(A_i) \coloneqq [\![\, \texttt{sort}(A_i) \,]\!] = [\![\, \tau_i \,]\!] \tag{2.6}$$

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

$$\texttt{sort}(\overline{A}) \coloneqq \langle\, \tau_{i_1}, \ldots, \tau_{i_j} \,\rangle \tag{2.7}$$

$$\begin{aligned} \texttt{dom}(\overline{A}) \coloneqq \{ \langle\, v_{i_1}, \ldots, v_{i_j} \,\rangle \mid \langle\, v_{i_1}, \ldots, v_{i_j} \,\rangle \in [\![\, \texttt{sort}(\overline{A}) \,]\!] \\ \wedge\, \forall l, m \in [1, j] (A_{i_l} = A_{i_m} \implies v_{i_l} = v_{i_m}) \} \end{aligned} \tag{2.8}$$

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

$$\texttt{sort}(\mathbb{S}(T)) \coloneqq \texttt{sort}(A_1 \cdots A_k) = \langle\, \tau_1, \ldots, \tau_k \,\rangle \tag{2.9}$$

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

$$\texttt{dom}(\mathbb{S}(T)) \coloneqq \texttt{dom}(A_1 \cdots A_k) = [\![\, \langle\, \tau_1, \ldots, \tau_k \,\rangle \,]\!] = [\![\, \texttt{sort}(\mathbb{S}(T)) \,]\!] \tag{2.10}$$

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 $\mathbb{T}(T)$ 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 $\mathbb{D}$ *is a finite set of table names*

$$\{T_1, \ldots, T_m\} \subset \texttt{tname}$$

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

We define $\texttt{adom}(\mathbb{D}) \subset \texttt{dom}$ to be the active domain of database $\mathbb{D}$, which is the set of values that occur within any attribute of any tuple within $\mathbb{D}$.

COCQL$^{\mathcal{F}}$ and COCQL$^{\mathcal{F}-}$ 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, COCQL$^{\mathcal{F}}$ is intended as an extension of conjunctive queries that allows for the construction of complex objects.

Definition 2.1.17 (COCQL$^{\mathcal{F}}$ Query) *A COCQL$^{\mathcal{F}}$ query Q is an expression of the form*

$$\textsc{Eval}(T, f, \mathbf{E}(T))$$

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*
- *$\textsc{Eval}()$ 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 $\mathbb{D}$, the result of evaluating Q over $\mathbb{D}$—denoted $(Q)^{\mathbb{D}}$—is a complex object of sort $\mathtt{sort}(Q) := \mathtt{sort}(f(T))$.

If for every database instance $\mathbb{D}$ the object $(Q)^{\mathbb{D}}$ 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 COCQL$^{\mathcal{F}}$ query contains an algebraic expression, it also contains some additional non-algebraic syntax. These non-algebraic constructs lead to several features that distinguish COCQL$^{\mathcal{F}}$ 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 $\mathtt{aname}$ with which a sort $\mathtt{sort}(V_i)$ is implicitly associated, and $v_i \in \mathtt{dom}(V_i) = [\![\, \mathtt{sort}(V_i) \,]\!]$. 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.

Operator Name	Query Expression	Output Table T	
		Schema $\mathbb{S}(T)$	**Instance $\mathbb{T}(T)$**
Base Relation	$T(A_1, \ldots, A_k)$ where $A_1, \ldots A_k \in \texttt{aname}$ are "fresh" attribute names	$T(A_1, \ldots, A_k)$	$\mathbb{T}(T) \in \mathbb{D}$
Cartesian Product	$T_1 \times T_2$	$T(A_1, \ldots, A_{k_1}, B_1, \ldots, B_{k_2})$	$\{\!\!\{ \langle t_1[A_1], \ldots, t_1[A_{k_1}], t_2[B_1], \ldots, t_2[B_{k_2}] \rangle \mid t_1 \in T_1, t_2 \in T_2 \}\!\!\}$
Selection	$\sigma_{A=B}(T_1)$ where $\texttt{sort}(A) = \texttt{sort}(B) = \texttt{dom}$	$T(A_1, \ldots, A_{k_1})$	$\{\!\!\{ \langle t_1[A_1], \ldots, t_1[A_{k_1}] \rangle \mid t_1 \in T_1, t_1[A] = t_1[B] \}\!\!\}$
Duplicate-Preserving Projection	$\Pi^{dup}_{A_{i_1}, \ldots, A_{i_k}}(T_1)$	$T(A_{i_1}, \ldots, A_{i_k})$	$\{\!\!\{ \langle t_1[A_{i_1}], \ldots, t_1[A_{i_k}] \rangle \mid t_1 \in T_1 \}\!\!\}$
Duplicate-Eliminating Projection	$\Pi_{A_{i_1}, \ldots, A_{i_k}}(T_1)$ where $\texttt{sort}(A_{i_1}) = \ldots = \texttt{sort}(A_{i_k}) = \texttt{dom}$	$T(A_{i_1}, \ldots, A_{i_k})$	$\{ \langle t_1[A_{i_1}], \ldots, t_1[A_{i_k}] \rangle \mid t_1 \in T_1 \}$
Scalar Subquery	$\lambda_{X=f(T_2)}(T_1)$ where $X \in \texttt{aname}$ is a "fresh" attribute name and $f \in \mathcal{F}$	$T(A_1, \ldots, A_{k_1}, X)$ where $\texttt{sort}(X) = \texttt{sort}(f(T_2))$	$\{\!\!\{ \langle t_1[A_1], \ldots, t_1[A_{k_1}], \textsc{Eval}(T_2, f, \mathbf{E}(T) \cup \{A_1 \leftarrow t_1[A_1], \ldots, A_{k_1} \leftarrow t_1[A_{k_1}]\}) \rangle \mid t_1 \in T_1 \}\!\!\}$
Conjunctive Selection	$\sigma_{p_1 \wedge p_2}(T_1)$	$\sigma_{p_2}(\sigma_{p_1}(T_1))$	
Join	$T_1 \bowtie_p T_2$	$\sigma_p(T_1 \times T_2)$	
Generalized Projection	$\Pi^{X=f(A_{j_1}, \ldots, A_{j_l})}_{A_{i_1}, \ldots, A_{i_k}}(T_1)$ where $X \in \texttt{aname}$ is a "fresh" attribute name and $\texttt{sort}(A_{i_1}) = \ldots = \texttt{sort}(A_{i_k}) = \texttt{dom}$	$\lambda_{X=f(\Pi^{dup}_{A'_{j_1}, \ldots, A'_{j_l}}(\sigma_{A'_{i_1}=A_{i_1} \wedge \ldots \wedge A'_{i_k}=A_{i_k}}(T'_1)))}(\Pi_{A_{i_1}, \ldots, A_{i_k}}(T_1))$ where T'_1 is a copy of T_1 with schema $T'_1(A'_1, \ldots, A'_{k_1})$	

Figure 2.5: $\texttt{COCQL}^{\mathcal{F}}$ 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 $\texttt{sort}(f(T))$ 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

$$\mathcal{F} = \{f_{\texttt{b}}, f_{\texttt{n}}, f_{\texttt{s}}\} \tag{2.11}$$

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

$$\begin{array}{ll} f_{\texttt{b}}(T) := \texttt{Bag}(\mathbb{T}(T)) & \texttt{sort}(f_{\texttt{b}}(T)) = \{\!|\, \texttt{sort}(\mathbb{S}(T)) \,|\!\} \\ f_{\texttt{n}}(T) := \texttt{NormBag}(\mathbb{T}(T)) & \texttt{sort}(f_{\texttt{n}}(T)) = \{\!|\!|\, \texttt{sort}(\mathbb{S}(T)) \,|\!|\!\} \\ f_{\texttt{s}}(T) := \texttt{Set}(\mathbb{T}(T)) & \texttt{sort}(f_{\texttt{s}}(T)) = \{\, \texttt{sort}(\mathbb{S}(T)) \,\} \end{array}$$

3. **Query Composability:** A $\texttt{COCQL}^{\mathcal{F}}$ query $Q = \textsc{Eval}(T, f, \mathbf{E}(T))$ takes as inputs a set of tables and outputs a complex object of sort $\texttt{sort}(Q) = \texttt{sort}(f(T))$. Hence, $\texttt{COCQL}^{\mathcal{F}}$ 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 λ 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 $\mathbb{T}(T_1)$.[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 $A \in \texttt{aname} \cup \texttt{dom}$ 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 $A \in \texttt{aname}$ but is not an attribute in $\mathbb{S}(T_1)$, then $\mathbf{E}(T)$ must contain some assignment $A \leftarrow v$ and $t_1[A] := v$.
   - If $A \in \texttt{dom}$, then $t_1[A] := A$ and $\texttt{dom}(A) := \texttt{dom}$.

The definitions in Figure 2.5 for the core algebraic operators *base relation*, *Cartesian product*, *selection*, and *projection* (duplicate-preserving and duplicate-eliminating) are all fairly standard, as are the syntactic extensions *conjunctive selection* and *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 $\texttt{dom}^{\infty}$ 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 $\mathbb{D}_1$.

Prof

pid
a
b
c

Course

cid	pid
c_1	a
c_2	a
c_3	b
c_4	b
c_5	b

Student

sid
1
2
3
4

Enrol

cid	sid
c_1	1
c_1	2
c_2	2
c_3	1
c_4	3

The following $\texttt{COCQL}^{\{f_{\mathtt{b}},f_{\mathtt{s}},f_{\mathtt{n}}\}}$ query Q_7 returns the set of pairs (p,s) such that professor p teaches a course in which student s is enrolled.

$$Q_7\colon \quad \textsc{Eval}(\Pi^{dup}_{B,D}(\texttt{Course}(A,B) \bowtie_{A=C} \texttt{Enrol}(C,D)), f_{\mathtt{s}}, \varnothing)$$

The reader can verify that Q_7 returns the following sort

$$\texttt{sort}(Q_7) = \{\!\langle \texttt{dom}(B), \texttt{dom}(D) \rangle\!\} = \{\!\langle \texttt{dom}, \texttt{dom} \rangle\!\} = \tau_2 \qquad \text{from Example 4}$$

and that $Q_7^{\mathbb{D}_1}$—the result of evaluating Q_7 over database $\mathbb{D}_1$—yields the following complex object.

$$Q_7^{\mathbb{D}_1} = \{\!\langle a,1 \rangle, \langle a,2 \rangle, \langle b,2 \rangle, \langle b,3 \rangle\!\} = o_2 \qquad \text{from Example 5}$$

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 λ to emphasize the obvious similarity to a lambda-calculus expression. Our λ operator is similar to the *Apply* operator that Galindo-Legaria and Joshi introduce to handle correlated subqueries [38, 39], but whereas *Apply* returns a table for the subquery which is then crossed with T_1, λ 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 λ. 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 $\mathbb{D}_1$ as in Example 9, consider the following $\texttt{COCQL}^{\{f_\texttt{b},f_\texttt{s},f_\texttt{n}\}}$ query Q_8 which returns the sets of students who have been taught by each professor.

$$Q_8: \qquad \textsc{Eval}(\lambda_{F=f_\texttt{s}(\Pi^{dup}_D(\sigma_{B=E}(\texttt{Course}(A,B)\bowtie_{A=C}\texttt{Enrol}(C,D))))}(\texttt{Prof}(E)), f_\texttt{b}, \varnothing)$$

The reader can verify that Q_8 returns the following sort

$$\texttt{sort}(Q_8) = \{\!|\, \langle\, \texttt{dom}(E), \texttt{dom}(F) \,\rangle \,|\!\} = \{\!|\, \langle\, \texttt{dom}, \{\, \langle\, \texttt{dom} \,\rangle \,\} \,\rangle \,|\!\} = \tau_7 \qquad \text{from Example 4}$$

and that $Q_8^{\mathbb{D}_1}$ yields the following complex object.

$$Q_8^{\mathbb{D}_1} = \{\!|\, \langle\, a, \{\, \langle\, 1 \,\rangle, \langle\, 2 \,\rangle \,\} \,\rangle, \langle\, b, \{\, \langle\, 1 \,\rangle, \langle\, 3 \,\rangle \,\} \,\rangle, \langle\, c, \{\ \} \,\rangle \,|\!\} = o_7 \qquad \text{from Example 5}$$

Query Q_8 is able to generate *non-trivial incomplete* object o_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 $\texttt{COCQL}^{\mathcal{F}}$ queries is those that are known to always output complete objects.

Definition 2.1.18 ($\texttt{COCQL}^{\mathcal{F}\text{-}}$ **Query)** *The language $\texttt{COCQL}^{\mathcal{F}\text{-}}$ is the subclass of $\texttt{COCQL}^{\mathcal{F}}$ queries that do not contain an explicit scalar subquery operator λ. Queries in $\texttt{COCQL}^{\mathcal{F}\text{-}}$ 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 $\mathcal{F}$, while the query algebra simply treats $\texttt{dom}^\infty$ as an abstract domain. This is why we christen $\texttt{COCQL}^{\mathcal{F}}$ as an "object-constructing" query language, rather than an "object query language."

Relating $\texttt{COCQL}^{\mathcal{F}}$ 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 $\texttt{COCQL}^{\mathcal{F}}$. First, the SQL data model allows the base tables stored by the RDBMS to contain duplicates, whereas $\texttt{COCQL}^{\mathcal{F}}$ 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 dom (the underlying domain for the $\texttt{COCQL}^{\mathcal{F}}$ data model). Third, SQL tables only allow atomic-valued attributes, whereas $\texttt{COCQL}^{\mathcal{F}}$ queries can construct arbitrary complex-valued attributes. As

previously mentioned, this third difference is relatively minor since the $\texttt{COCQL}^{\mathcal{F}}$ algebraic operators cannot look inside a complex-valued attribute, and so $\texttt{COCQL}^{\mathcal{F}}$ essentially treats $\texttt{dom}^{\infty}$ as an atomic domain.

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

Proposition 2.1.19 *$\texttt{COCQL}^{\mathcal{F},f_\mathtt{b}}$ corresponds to a restricted class of queries that can be expressed in $\texttt{SQL}^{\mathcal{F}}$ using only the following constructs:*

1. *SELECT clauses composed of attribute names, constants, scalar subqueries, and aggregation expressions of the form $f(A_1, \ldots, A_k)$, $f \in \mathcal{F}$;*
2. *FROM clauses composed of a list of base relation names and nested $\texttt{SQL}^{\mathcal{F}}$ table expressions;*
3. *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. *GROUP BY clauses composed only of constants and attributes originating from base relations.*

We denote this subset of $\texttt{SQL}^{\mathcal{F}}$ expressions as $\texttt{CSQL}^{\mathcal{F}}$ (for "Conjunctive $\texttt{SQL}^{\mathcal{F}}$").

Proposition 2.1.20 *$\texttt{COCQL}^{\mathcal{F},f_\mathtt{b}\text{-}}$ corresponds to the restricted class of $\texttt{CSQL}^{\mathcal{F}}$ queries in which*

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

(i.e., the subset of $\texttt{CSQL}^{\mathcal{F}}$ that permits scalar aggregation *only at the root block). We denote this subset of $\texttt{CSQL}^{\mathcal{F}}$ expressions as $\texttt{CSQL}^{\mathcal{F}\text{-}}$.*

A $\texttt{COCQL}^{\mathcal{F},f_\mathtt{b}}$ query always returns a single complex object (treated as a scalar value from abstract domain $\texttt{dom}^{\infty}$). In contrast, a $\texttt{CSQL}^{\mathcal{F}}$ query always returns a table which—per Definition 2.1.14—is a table schema paired with a bag-object. Corresponding to any $\texttt{CSQL}^{\mathcal{F}}$ query is a $\texttt{COCQL}^{\mathcal{F},f_\mathtt{b}}$ query whose root invocation of EVAL applies the function $f_\mathtt{b}$ to construct the same bag-object as the table instance returned by the $\texttt{CSQL}^{\mathcal{F}}$ query.

Example 11 Assume that a database schema contains relation `Person(id, fname, lname, phone)`. $\texttt{CSQL}^{\mathcal{F}}$ query Q_9 returns a table of (fname, phone) pairs for persons with the last name DeHaan.

Q_9: SELECT fname, phone FROM Person WHERE lname = 'DeHaan'

Query Q_{10} shows a corresponding COCQL$^{\mathcal{F},f_\mathtt{b}}$ query which returns a bag of binary tuples.

Q_{10}: $\textsc{Eval}(\Pi^{dup}_{B,D}(\sigma_{C=\text{'DeHaan'}}(\mathtt{Person}(A,B,C,D))), f_\mathtt{b}, \varnothing)$

Given an arbitrary COCQL$^{\mathcal{F}}$ (or COCQL$^{\mathcal{F},f_\mathtt{b}}$) query whose root invocation of Eval applies the function f, the translation of the query into CSQL$^{\mathcal{F}}$ depends upon f. If $f = f_\mathtt{b}$, 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 $f \neq f_\mathtt{b}$, 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}: $\textsc{Eval}(\Pi^{dup}_{B,D}(\sigma_{C=\text{'DeHaan'}}(\mathtt{Person}(A,B,C,D))), f, \varnothing)$

If $f = f_\mathtt{b}$, 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}: SELECT f(fname, phone) 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$^{\mathcal{F},f_\mathtt{b}}$ query Q_{14}.

Q_{13}: SELECT f(fname) FROM Person GROUP BY lname
Q_{14}: $\textsc{Eval}(\Pi^{dup}_{E}(\Pi^{E=f(B)}_{C}(\mathtt{Person}(A,B,C,D))), f_\mathtt{b}, \varnothing)$

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

Q_{15}: $\textsc{Eval}(\ \Pi^{dup}_{E}(\lambda_{E=f(\Pi^{dup}_{B'}(\sigma_{C'=C}(\texttt{Person}(A',B',C',D'))))}(\Pi_C(\texttt{Person}(A,B,C,D)))),$
$f_{\mathfrak{b}}, \varnothing)$

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

$\texttt{CSQL}^{\mathcal{F}}$ queries Q_{13} and Q_{16} are equivalent; however, this equivalence relies upon the fact that attribute `lname` does not contain NULL values (as per our definition of language $\texttt{SQL}^{\mathcal{F}}$). If the data model were extended with NULL values, the `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 $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}}$ and $\texttt{CSQL}^{\mathcal{F}}$ posited in Proposition 2.1.19 can be verified by

1. defining a canonical form for $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}}$ algebra expressions,
2. demonstrating the correspondence between $\texttt{CSQL}^{\mathcal{F}}$ expressions and $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}}$ expressions in canonical form, and
3. showing that every $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}}$ can be converted to the canonical form by a sequence of equivalence-preserving algebraic transformations.

The correspondence between $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}\text{-}}$ and $\texttt{CSQL}^{\mathcal{F}\text{-}}$ posited in Proposition 2.1.20 can be verified in a similar fashion.

Deciding Equivalence of $\texttt{COCQL}^{\mathcal{F}}$ Queries

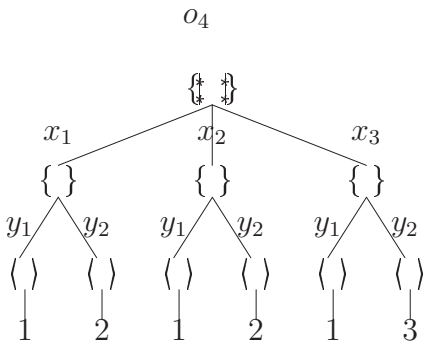
In the introduction to this thesis we purport to study the equivalence of `SQL` queries containing nested aggregation functions. Having just introduced the algebraic query language $\texttt{COCQL}^{\mathcal{F}}$ and shown its relationship to a specific class of `SQL` queries, we briefly define formally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

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

Proposition 2.1.22 *The query equivalence problems for $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}}$ and $\texttt{CSQL}^{\mathcal{F}}$ are polynomial-time reducible to each other.*

Proposition 2.1.23 *The query equivalence problems for $\texttt{COCQL}^{\mathcal{F},f_{\mathfrak{b}}\text{-}}$ and $\texttt{CSQL}^{\mathcal{F}\text{-}}$ are polynomial-time reducible to each other.*

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



R_1

X	Y	Z
x_1	y_1	1
x_1	y_2	2
x_2	y_1	1
x_2	y_2	2
x_3	y_1	1
x_3	y_2	3

R_2

X	Y_1	Y_2	Z
x_1	y	a	1
x_1	y	b	2
x_2	y	a	1
x_2	y	b	2
x_3	y	a	1
x_3	y	b	3

Figure 2.6: Encoding of complete chain object o_4

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" $\overline{\mathcal{I}}_i$ is a sequence*

$$\overline{\mathcal{I}}_i := I_{i.1} \cdots I_{i.k_i}$$

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 $\mathcal{I}_i$ to denote the set *of attributes occurring in $\overline{\mathcal{I}}_i$. For convenience, we use $\overline{\mathcal{I}}_{[i,j]}$ to denote the sequence $\overline{\mathcal{I}}_i\overline{\mathcal{I}}_{i+1}\cdots\overline{\mathcal{I}}_j$, and $\mathcal{I}_{[i,j]}$ to denote the corresponding set $\bigcup_{l\in[i,j]}\mathcal{I}_l$.*

- *The "tuple of output attributes" $\overline{\mathcal{V}}$ is a sequence*

$$\overline{\mathcal{V}} := V_1 \cdots V_{k_v}$$

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 $\overline{\mathcal{V}}$.*

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

$$\mathcal{I}_1 \cup \ldots \cup \mathcal{I}_d \cup \mathcal{V} = \{A_1, \ldots, A_k\} \tag{2.12}$$

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

For convenience, we use the notation

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

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

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

$$\mathbb{T}(R[\overline{a}]) := \Pi_{\overline{\mathcal{I}}_{[l,d]}\overline{\mathcal{V}}}(\sigma_{\overline{\mathcal{I}}_{[1,l-1]}=\overline{a}}(R)) \tag{2.14}$$

where Π and σ are relational projection and selection, respectively. Intuitively, if R encodes an object of depth d, then $R[\overline{a}]$ encodes a sub-object of depth $d-l+1$ that

[4]At times we may refer to encoding relation R as having "relation schema $R(\overline{\mathcal{I}}_{[1,d]}\overline{\mathcal{V}})$" or containing "tuples over $\mathtt{dom}(\overline{\mathcal{I}}_{[1,d]}\overline{\mathcal{V}})$." This is technically incorrect because tables (and hence relations) are required to have distinct attribute names, while the sequence $\overline{\mathcal{I}}_{[1,d]}\overline{\mathcal{V}}$ 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 $\mathtt{dom}(\overline{\mathcal{I}}_{[1,d]}\overline{\mathcal{V}}) = \mathtt{dom}(ABCDC)$ since agreement of values for repeated attributes is guaranteed by equation 2.8.

R_3

A	B	C	D	C
a_1	b_1	1	d_1	1
a_1	b_1	2	d_1	2
a_1	b_2	1	d_1	1
a_1	b_2	2	d_1	2
a_2	b_1	1	d_3	1
a_2	b_1	1	d_4	1
a_2	b_1	2	d_3	2
a_2	b_2	1	d_3	1
a_2	b_2	2	d_3	2
a_3	b_1	1	d_5	1
a_3	b_1	3	d_5	3
a_3	b_1	3	d_6	3
a_3	b_2	1	d_5	1
a_3	b_2	3	d_5	3
a_3	b_2	3	d_6	3

$R_3[a_2b_1]$

C	D	C
1	d_3	1
1	d_4	1
2	d_3	2

$R_3[a_2b_1 1\, d_3]$

C
1

Figure 2.7: Encoding relation R_3 and select sub-relations

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 $R_3[a_2b_1 1\, d_3]$ 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 $\overline{\S}$ of length d, the function DecodeSimple$(R, \overline{\S})$ shown in Algorithm 2 computes a complete chain object of sort $(\overline{\S}, |\overline{\mathcal{V}}|)$. If R is non-empty, then DecodeSimple$(R, \overline{\S})$ returns a hierarchy of d (possibly zero) levels of nested collections with tuples of arity $|\overline{\mathcal{V}}|$ at the leaves. If R is empty, then Definition 2.2.2 requires that $d > 0$, and so DecodeSimple$(R, \overline{\S})$ returns a single empty collection of type $\S_1$. The structuring of Algorithm 2 into two procedures DecodeSimple and 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 `ns` yields object o_4.

$$
\begin{aligned}
&\textsc{DecodeSimple}(R_1, \mathtt{ns}) = \textsc{InnerDecodeSimple}(R_1, \mathtt{ns}) \\
&\quad = \{\!\!\{\, \textsc{InnerDecodeSimple}(R_1[x_1], \mathtt{s}), \\
&\qquad \textsc{InnerDecodeSimple}(R_1[x_2], \mathtt{s}), \\
&\qquad \textsc{InnerDecodeSimple}(R_1[x_3], \mathtt{s}) \,\}\!\!\} \\
&\quad = \{\!\!\{\, \{\, \textsc{InnerDecodeSimple}(R_1[x_1y_1], \varnothing), \textsc{InnerDecodeSimple}(R_1[x_1y_2], \varnothing) \,\}, \\
&\qquad \{\, \textsc{InnerDecodeSimple}(R_1[x_2y_1], \varnothing), \textsc{InnerDecodeSimple}(R_1[x_2y_2], \varnothing) \,\}, \\
&\qquad \{\, \textsc{InnerDecodeSimple}(R_1[x_3y_1], \varnothing), \textsc{InnerDecodeSimple}(R_1[x_3y_2], \varnothing) \,\} \,\}\!\!\}
\end{aligned}
$$

Algorithm 2 Reconstructing an encoded chain object

DecodeSimple$(R, \overline{\S})$

▷ **Input:** encoding relation $R(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$
▷ **Input:** compatible signature $\overline{\S}$
▷ **Output:** complete or trivial chain object of sort $(\overline{\S}, |\overline{\mathcal{V}}|)$

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

InnerDecodeSimple$(R, \overline{\S})$

▷ **Input:** non-empty encoding relation $R(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$
▷ **Input:** compatible signature $\overline{\S}$
▷ **Output:** complete or trivial chain object of sort $(\overline{\S}, |\overline{\mathcal{V}}|)$

1 **if** $d = 0$ ▷ R contains singleton tuple $\langle\, a_1, \ldots, a_{|\overline{\mathcal{V}}|} \,\rangle$
2 **then return** $\langle\, a_1, \ldots, a_{|\overline{\mathcal{V}}|} \,\rangle$
3 **elseif** $\overline{\S} = \mathtt{s}\overline{Y}$
4 **then return** $\{\ \text{InnerDecodeSimple}(R[\overline{a}], \overline{Y}) \mid \overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)\ \}$
5 **elseif** $\overline{\S} = \mathtt{b}\overline{Y}$
6 **then return** $\{\!|\ \text{InnerDecodeSimple}(R[\overline{a}], \overline{Y}) \mid \overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)\ |\!\}$
7 **else** ▷ $\overline{\S} = \mathtt{n}\overline{Y}$
8 **return** $\{\!\{\ \text{InnerDecodeSimple}(R[\overline{a}], \overline{Y}) \mid \overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)\ \}\!\}$

$= \{\!\{ \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 3\rangle\} \}\!\}$
$= o_4$

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 τ_4.

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

$$\begin{aligned}
\textsc{DecodeSimple}(R_3, \texttt{ns}) &= \{\!\{ \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 1\rangle, \langle 2\rangle\}, \\
&\qquad \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 3\rangle, \langle 3\rangle\}, \{\langle 1\rangle, \langle 3\rangle, \langle 3\rangle\} \}\!\} \\
&= \{\!\{ \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 2\rangle\}, \\
&\qquad \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 3\rangle\}, \{\langle 1\rangle, \langle 3\rangle\} \}\!\} \\
&= \{\!\{ \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 2\rangle\}, \{\langle 1\rangle, \langle 3\rangle\} \}\!\} \\
&= o_4
\end{aligned}$$

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 - denote a designated atomic element not occurring in the atomic domain `dom`.

R_4

A	B	C	D
a_1	b_1	a	1
a_1	b_2	a	2
a_2	b_1	b	1
a_2	b_2	b	3
a_3	b_1	c	-

R_5

A	B	C	D	E	F
a_1	-	c_1	d_1	-	-

R_6

A	B	C	D	E	F	G	H	I
a_1	b_1	c_1	-	e_1	f_1	a	-	5
a_1	b_1	c_2	-	e_1	f_1	c	-	5
a_1	b_1	c_3	-	e_1	f_1	c	-	5
a_1	b_2	c_1	-	e_1	f_1	a	-	5
a_1	b_2	c_2	-	e_1	f_1	c	-	5
a_1	b_2	c_3	-	e_1	f_1	c	-	5
a_1	b_3	-	-	e_1	f_1	-	-	5

Figure 2.8: Encoding relations R_4, R_5, and R_6

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 -. 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 - symbol if and only if the edge leads into a shaded node.
2. Each occurrence of the atomic value dom should be replaced by - within the output attributes of the encoding relation.

Example 16 Applying the modified encoding algorithm to objects $\textsc{Linearize}_{\tau_7}(o_7)$, $\textsc{Linearize}_{\tau_8}(o_8)$, and $\textsc{Linearize}_{\tau_9}(o_9)$ 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 Decode. The key modification is that in the recursive invocation to decode sub-relation $R[\overline{a}]$, we test the predicate $\overline{a} = \text{-}^{|\overline{a}|}$ which returns TRUE when $\overline{a}$ is a tuple composed *entirely* of the - 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 - symbols within the output are translated into the atomic value 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 - may not occur within the base relations of a database, or as an explicit constant within an object-constructing $\texttt{COCQL}^{\mathcal{F}}$ query.

Algorithm 3 Reconstructing an encoded *linearized* object

DECODE($R, \overline{\S}$)

▷ **Input:** encoding relation $R(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$
▷ **Input:** compatible signature $\overline{\S}$
▷ **Output:** complete or trivial chain object of sort $(\overline{\S}, |\overline{\mathcal{V}}|)$, possibly containing shaded nodes

if R is non-empty
  then return INNERDECODE($R, \overline{\S}$, FALSE)
  else ▷ $d > 0$ and R encodes a trivial empty collection
    if $\overline{\S} = \mathtt{s}\overline{Y}$
      then return { }
    elseif $\overline{\S} = \mathtt{b}\overline{Y}$
      then return {| |}
    else ▷ $\overline{\S} = \mathtt{n}\overline{Y}$
      return {* *}

INNERDECODE($R, \overline{\S}, shade$)

▷ **Input:** non-empty encoding relation $R(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$
▷ **Input:** compatible signature $\overline{\S}$
▷ **Input:** boolean *shade* indicating if shading is required
▷ **Output:** complete or trivial chain object of sort $(\overline{\S}, |\overline{\mathcal{V}}|)$, possibly containing shaded nodes

if $d = 0$ ▷ R contains singleton tuple $\langle a_1, \ldots, a_{|\overline{\mathcal{V}}|} \rangle$
  then $t \leftarrow \langle a_1, \ldots, a_{|\overline{\mathcal{V}}|} \rangle$
    Replace each occurrence of - within t with dom
    return t
elseif $\overline{\S} = \mathtt{s}\overline{Y}$
  then if *shade* = TRUE
    then return { INNERDECODE($R[\overline{a}]$, $\overline{Y}$, $(\overline{a} = \text{-}^{|\overline{a}|})$) | $\overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)$ }
    else return { INNERDECODE($R[\overline{a}]$, $\overline{Y}$, $(\overline{a} = \text{-}^{|\overline{a}|})$) | $\overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)$ }
elseif $\overline{\S} = \mathtt{b}\overline{Y}$
  then if *shade* = TRUE
    then return {| INNERDECODE($R[\overline{a}]$, $\overline{Y}$, $(\overline{a} = \text{-}^{|\overline{a}|})$) | $\overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)$ |}
    else return {| INNERDECODE($R[\overline{a}]$, $\overline{Y}$, $(\overline{a} = \text{-}^{|\overline{a}|})$) | $\overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)$ |}
else ▷ $\overline{\S} = \mathtt{n}\overline{Y}$
  if *shade* = TRUE
    then return {* INNERDECODE($R[\overline{a}]$, $\overline{Y}$, $(\overline{a} = \text{-}^{|\overline{a}|})$) | $\overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)$ *}
    else return {* INNERDECODE($R[\overline{a}]$, $\overline{Y}$, $(\overline{a} = \text{-}^{|\overline{a}|})$) | $\overline{a} \in \Pi_{\overline{\mathcal{I}}_1}(R)$ *}

R_7	S	T	U	V	W	X	Y	Z
	s_1	-	-	-	w_1	x_1	-	-
	s_2	-	-	v_1	w_1	x_1	-	-
	s_2	-	-	v_1	-	x_1	-	-
	s_2	-	-	v_1	-	-	-	-

Figure 2.9: Encoding relation R_7

Example 17 The reader can verify the following for the relations shown in Figure 2.8.

$$\begin{aligned}
\textsc{Decode}(R_4, \texttt{bs}) &= \textsc{Linearize}_{\tau_7}(o_7) \\
\textsc{Decode}(R_5, \texttt{sbss}) &= \textsc{Linearize}_{\tau_8}(o_8) \\
\textsc{Decode}(R_6, \texttt{sbnsbn}) &= \textsc{Linearize}_{\tau_8}(o_8)
\end{aligned}$$

Decoding relation R_7 in Figure 2.9 with signature sbss also yields object $\textsc{Linearize}_{\tau_8}(o_8)$.

$$\begin{aligned}
&\textsc{Decode}(R_7, \texttt{sbss}) = \textsc{InnerDecode}(R_7, \texttt{sbss}, \textsc{false}) \\
&\quad = \{\, \textsc{InnerDecode}(R_7[s_1], \texttt{bss}, \textsc{false}), \\
&\qquad \textsc{InnerDecode}(R_7[s_2], \texttt{bss}, \textsc{false}) \,\} \\
&\quad = \{\, \{\!|\, \textsc{InnerDecode}(R_7[s_1\texttt{--}], \texttt{ss}, \textsc{true}) \,|\!\}, \\
&\qquad \{\!|\, \textsc{InnerDecode}(R_7[s_2\texttt{--}], \texttt{ss}, \textsc{true}) \,|\!\} \,\} \\
&\quad = \{\, \{\!|\, \{\, \textsc{InnerDecode}(R_7[s_1\texttt{---}w_1], \texttt{s}, \textsc{false}) \,\} \,|\!\}, \\
&\qquad \{\!|\, \{\, \textsc{InnerDecode}(R_7[s_2\texttt{--}v_1w_1], \texttt{s}, \textsc{false}), \\
&\qquad\quad \textsc{InnerDecode}(R_7[s_2\texttt{--}v_1\texttt{-}], \texttt{s}, \textsc{false}) \,\} \,|\!\} \,\} \\
&\quad = \{\, \{\!|\, \{\, \{\, \textsc{InnerDecode}(R_7[s_1\texttt{---}w_1x_1], \varnothing, \textsc{false}) \,\} \,\} \,|\!\}, \\
&\qquad \{\!|\, \{\, \{\, \textsc{InnerDecode}(R_7[s_2\texttt{--}v_1w_1x_1], \varnothing, \textsc{false}) \,\}, \\
&\qquad\quad \{\, \textsc{InnerDecode}(R_7[s_2\texttt{--}v_1\texttt{-}x_1], \varnothing, \textsc{false}), \\
&\qquad\quad\quad \textsc{InnerDecode}(R_7[s_2\texttt{--}v_1\texttt{--}], \varnothing, \textsc{true}) \,\} \,\} \,|\!\} \,\} \\
&\quad = \{\, \{\!|\, \{\, \{\, \langle\, \texttt{dom}, \texttt{dom} \,\rangle \,\} \,\} \,|\!\}, \\
&\qquad \{\!|\, \{\, \{\, \langle\, \texttt{dom}, \texttt{dom} \,\rangle \,\}, \{\, \langle\, \texttt{dom}, \texttt{dom} \,\rangle, \langle\, \texttt{dom}, \texttt{dom} \,\rangle \,\} \,\} \,|\!\} \,\} \\
&\quad = \{\, \{\!|\, \{\, \{\, \langle\, \texttt{dom}, \texttt{dom} \,\rangle \,\} \,\} \,|\!\} \,\} \\
&\quad = \textsc{Linearize}_{\tau_8}(o_8)
\end{aligned}$$

Algorithm 3 is clearly backwards-compatible with Algorithm 2. That is, for any encoding relation R and compatible signature $\overline{\S}$ such that R does not contain the symbol -, $\textsc{Decode}(R, \overline{\S}) = \textsc{DecodeSimple}(R, \overline{\S})$. 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 $\mathtt{ns}$, 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 $\mathtt{ns}$ used in the decoding. When decoded with signature $\mathtt{sb}$, 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 $\overline{\S}$, if*

$$\textsc{Decode}(R, \overline{\S}) = \textsc{Decode}(R', \overline{\S})$$

then we say that R and R' are encoding-equal w.r.t. signature $\overline{\S}$ *(abbreviated* $\overline{\S}$-equal*, and denoted $R \doteq_{\overline{\S}} R'$).*

We use $R \doteq_{\overline{\S}} R'$ to denote $\overline{\S}$-equality, as distinct from $R = R'$ which—as an extension of traditional relational equality—implies that $\mathbb{S}(R)$ and $\mathbb{S}(R')$ have identical arities of index attributes at each index level and of output attributes, and that $\mathbb{T}(R) = \mathbb{T}(R')$. Observe that $R \doteq_{\overline{\S}} R'$ implies that the encoding schemas $\mathbb{S}(R)$ and $\mathbb{S}(R')$ both have depth $|\overline{\S}|$ 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, $\mathtt{s}$-equality of encoding relations $R(\overline{\mathcal{I}}_1; \overline{\mathcal{V}})$ and $R'(\overline{\mathcal{I}}'_1, \overline{\mathcal{V}})$ corresponds to relational equality of *projections onto the output attributes*—that is, $\Pi_{\overline{\mathcal{V}}}(R) = \Pi_{\overline{\mathcal{V}}}(R')$, but it does not imply $\mathbb{T}(R) = \mathbb{T}(R')$ because index values could differ.

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

Definition 2.2.4 ($\overline{\S}$-Certificate) *Given encoding relations R and R' and signature $\overline{\S}$, a $\overline{\S}$-certificate is a tree rooted by a node $n_{(R,R')}$ that proves $R \doteq_{\overline{\S}} R'$. When $\S_1 = \mathtt{s}$ then $n_{(R,R')}$ is a set node; when $\S_1 = \mathtt{b}$ then $n_{(R,R')}$ is a bag node; when $\S_1 = \mathtt{n}$ then $n_{(R,R')}$ is a normalized bag node; and when $\overline{\S} = \varnothing$ then $n_{(R,R')}$ is a tuple node.*

Definition 2.2.5 (Set Node) *A set node $n^{\mathtt{s}}_{(R,R')}$ proves $R \doteq_{\mathtt{s}\overline{Y}} R'$ for some R, R', and $\overline{Y}$. It contains a function $f : \mathtt{adom}(\overline{\mathcal{I}}'_1, R') \to \mathtt{adom}(\overline{\mathcal{I}}_1, R)$ satisfying*

$$\forall \overline{x}' \in \mathtt{adom}(\overline{\mathcal{I}}'_1) \left[\begin{array}{c} \big(R[f(\overline{x}')] \doteq_{\overline{Y}} R'[\overline{x}']\big) \wedge \\ \big(\overline{x}' = \blacksquare^{|\overline{x}'|} \iff f(\overline{x}') = \blacksquare^{|f(\overline{x}')|}\big) \end{array} \right] \tag{2.15}$$

and a function $f' : \mathtt{adom}(\overline{\mathcal{I}}_1) \to \mathtt{adom}(\overline{\mathcal{I}}'_1)$ satisfying

$$\forall \overline{x} \in \mathtt{adom}(\overline{\mathcal{I}}_1) \left[\begin{array}{c} \big(R[\overline{x}] \doteq_{\overline{Y}} R'[f'(\overline{x})]\big) \wedge \\ \big(\overline{x} = \blacksquare^{|\overline{x}|} \iff f'(\overline{x}) = \blacksquare^{|f'(\overline{x})|}\big) \end{array} \right] \tag{2.16}$$

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

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

$$\begin{aligned}\textsc{Decode}(R'[\overline{x}'], \overline{Y}) &= \textsc{InnerDecode}(R'[\overline{x}'], \overline{Y}, \textsc{false}) \\ &= \textsc{InnerDecode}(R[f(\overline{x}')], \overline{Y}, \textsc{false}) \\ &= \textsc{Decode}(R[f(\overline{x}')], \overline{Y})\end{aligned}$$

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 $\textsc{InnerDecode}(R'[\overline{x}'], \overline{Y}, (\overline{x}' = \blacksquare^{|\overline{x}'|}))$ and $\textsc{InnerDecode}(R[f(\overline{x}')], \overline{Y}, (f(\overline{x}') = \blacksquare^{|f(\overline{x}')|}))$, respectively. Therefore, the second conjunct in formula 2.15 is needed to ensure that the predicates $\overline{x}' = \blacksquare^{|\overline{x}'|}$ and $f(\overline{x}') = \blacksquare^{|f(\overline{x}')|}$ agree, so that the roots of the decoded sub-objects are either both or neither shaded. Formula 2.15 thus guarantees $\textsc{Decode}(R', \mathtt{s}\overline{Y}) \subseteq \textsc{Decode}(R, \mathtt{s}\overline{Y})$. Formula 2.16 likewise guarantees $\textsc{Decode}(R, \mathtt{s}\overline{Y}) \subseteq \textsc{Decode}(R', \mathtt{s}\overline{Y})$.

Definition 2.2.6 (Bag Node) *A bag node $n^{\mathtt{b}}_{(R,R')}$ proves $R \doteq_{\mathtt{b}\overline{Y}} R'$ for some R, R', and $\overline{Y}$. It contains a* bijective *function $f : \mathtt{adom}(\overline{\mathcal{I}}'_1, R') \to \mathtt{adom}(\overline{\mathcal{I}}_1, R)$ satisfying*

$$\forall \overline{x}' \in \mathtt{adom}(\overline{\mathcal{I}}'_1) \left[\begin{array}{c} (R[f(\overline{x}')] \doteq_{\overline{Y}} R'[\overline{x}']) \wedge \\ \left(\overline{x}' = \blacksquare^{|\overline{x}'|} \iff f(\overline{x}') = \blacksquare^{|f(\overline{x}')|}\right) \end{array} \right] \tag{2.17}$$

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

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 $\mathtt{adom}(\overline{\mathcal{I}}'_1, R)$ and $\mathtt{adom}(\overline{\mathcal{I}}_1, R)$, we enforce that equal sub-objects of $\textsc{Decode}(R, \mathtt{s}\overline{Y})$ and $\textsc{Decode}(R', \mathtt{s}\overline{Y})$ can be paired up, guaranteeing identical absolute frequencies.

Definition 2.2.7 (Normalized Bag Node) *A normalized bag node $n^{\mathtt{n}}_{(R,R')}$ proves $R \doteq_{\mathtt{n}\overline{Y}} R'$ for some R, R', and $\overline{Y}$. It contains two* finite *domains D_1 and D_2, and two* surjective *functions $\rho : \mathtt{adom}(\overline{\mathcal{I}}_1, R) \to D_1$ and $\varrho : \mathtt{adom}(\overline{\mathcal{I}}'_1, R') \to D_2$ that satisfy*

$$\forall p \in D_1 . \forall q \in D_2 \left[(R)^p \doteq_{\mathtt{b}\overline{Y}} (R')^q \right]$$

where $(R)^p$ is the (non-empty) encoding relation defined by $\mathbb{S}((R)^p) := \mathbb{S}(R)$ and $\mathbb{T}((R)^p) := \sigma_{\rho(\overline{\mathcal{I}}_1)=p}(R)$, and $(R')^q$ is defined analogously using ϱ. For each pair $(p,\ q)$ such that $p \in D_1$ and $q \in D_2$, node $n^{\mathtt{n}}_{(R,R')}$ has a child $\mathtt{b}\overline{Y}$-certificate that proves $(R)^p \doteq_{\mathtt{b}\overline{Y}} (R')^q$.

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 $\mathtt{b}\overline{Y}$. 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

$$B_R \otimes |D_2| = B_{R'} \otimes |D_1| \tag{2.18}$$

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

Definition 2.2.8 (Tuple Node) *A tuple node $n^{\mathtt{t}}_{(R,R')}$ proves $R \doteq_{\varnothing} R'$ 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 $n^{\mathtt{t}}_{(R,R')}$ 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 $\overline{\S}$, $R \doteq_{\overline{\S}} R'$ if and only if there exists a $\overline{\S}$-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) $\mathtt{ns}$-certificate proving that $R_1 \doteq_{\mathtt{ns}} R_3$.

Observe that the nodes within a $\overline{\S}$-certificate are organized into levels of identical node type. For any index level i, if $\overline{\S}_i = \mathtt{b}$ then "the nodes in (bag) level i" are the equi-depth bag nodes that each prove $\mathtt{b}\overline{\S}_{[i+1,d]}$-equality of some pair of sub-relations. Similarly, if $\overline{\S}_i = \mathtt{s}$ then "the nodes in (set) level i" are the equi-depth set nodes that prove $\mathtt{s}\overline{\S}_{[i+1,d]}$-equality. If $\overline{\S}_i = \mathtt{n}$ then "the nodes in (normalized bag) level i" are the equi-depth normalized bag nodes that prove $\mathtt{n}\overline{\S}_{[i+1,d]}$-equality—note, however, that immediately underneath a normalized bag level is an extra layer of bag nodes proving $\mathtt{b}\overline{\S}_{[i+1,d]}$-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 $\mathtt{COCQL}^{\{f_{\mathtt{b}},f_{\mathtt{s}},f_{\mathtt{n}}\}}$ query equivalence (see Definition 2.1.21) to deciding "encoding-equivalence" between encoding queries. In

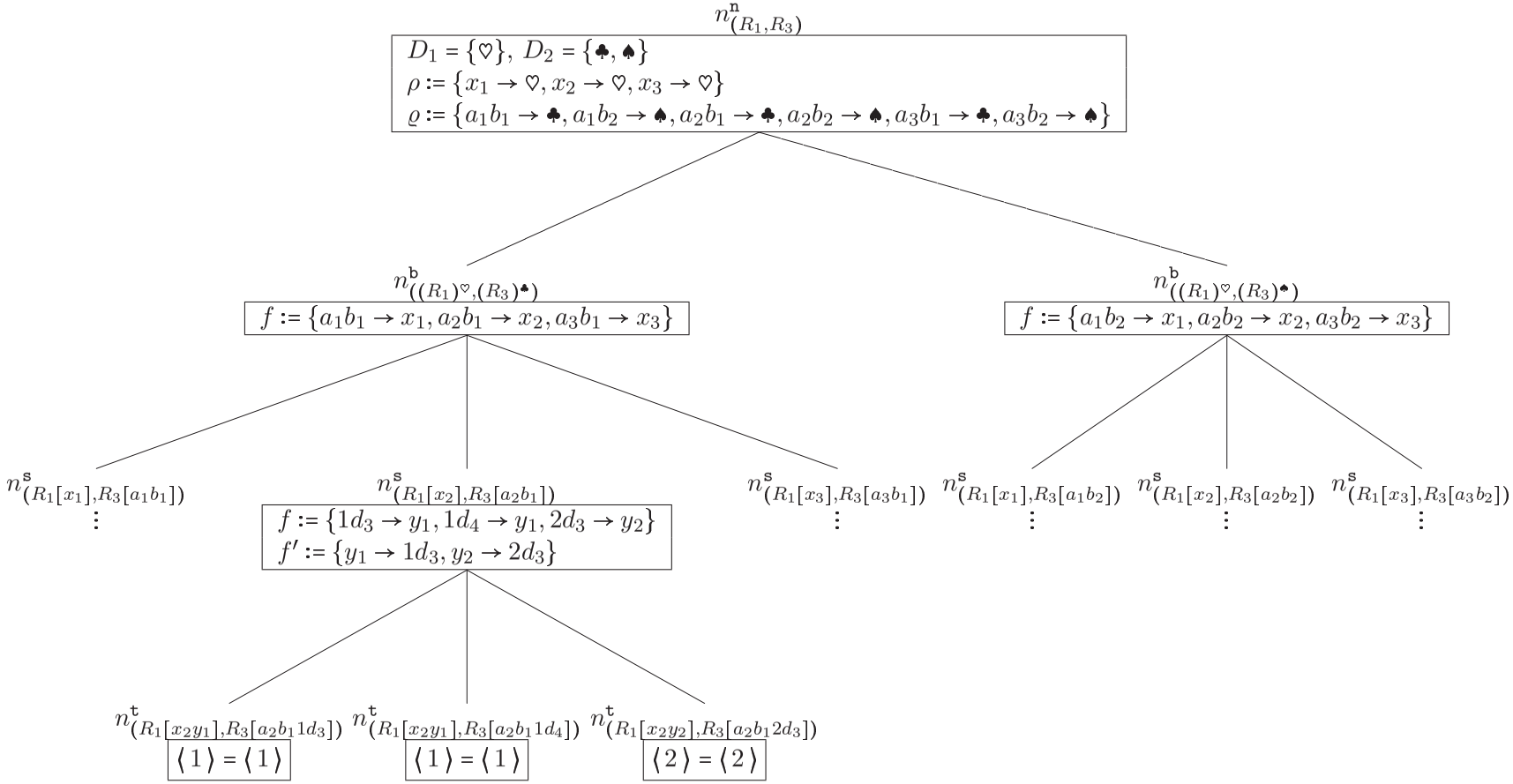


Figure 2.10: ns-certificate proving $R_1 \doteq_{\mathtt{ns}} R_3$

Section 2.3.1 we define a language for specifying encoding queries. In Section 2.3.2 we define a mapping from each $\texttt{COCQL}^{\{f_\texttt{b},f_\texttt{s},f_\texttt{n}\}}$ 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 *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 $\mathbb{D}$ be a relational database with schema $\mathbb{S}(\mathbb{D})$.

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

$$Q(\overline{\mathcal{W}}) \coloneq R_1(\overline{X}_1), \ldots, R_m(\overline{X}_m), Y_1 = Z_1, \ldots, Y_n = Z_n \tag{2.19}$$

where $\overline{\mathcal{W}}$ is a tuple composed of constants and variables, each R_i is a relation name in $\mathbb{S}(\mathbb{D})$, 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 $Q(\overline{\mathcal{W}})$ is termed the query *head*, each expression $R_i(\overline{X}_i)$ is called an *atom*, and each expression $Y_j = Z_j$ is called a *predicate*. The *body* of Q—denoted $\texttt{body}_Q$—is the set of atoms $\texttt{atoms}_Q$ and predicates $\texttt{pred}_Q$ 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 $\overline{\mathcal{W}}$.
- $\mathcal{C}$ denotes an infinite set of constants, and $\mathcal{C}_Q \subset \mathcal{C}$ is the set of constants occurring anywhere in Q.
- $\mathcal{P} \subseteq \mathcal{B}$ is a distinguished set of variables called the *parameters*.
- $\mathcal{L} \coloneq \mathcal{B} \setminus \mathcal{P}$ 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 *parameterized* when $|\mathcal{P}| > 0$. Query Q is called *boolean* if $|\overline{\mathcal{W}}| = 0$. Query Q is called *unsatisfiable* if $\texttt{pred}_Q$ transitively 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 $\texttt{pred}_Q$ 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 $\texttt{pred}_Q$ only reference constants and parameter variables.

An *embedding of* $\texttt{body}_Q$ *into* $\mathbb{D}$—denoted $\gamma : \texttt{body}_Q \to \mathbb{D}$, but for conciseness we often write $\gamma : Q \to \mathbb{D}$—is a mapping $\gamma : \mathcal{B} \to \texttt{dom}$ (extended with identity on the constants in $\mathcal{C}$) such that γ satisfies every atom in $\texttt{atoms}_Q$ and every predicate in $\texttt{pred}_Q$. Atom $R_i(\overline{X}_i)$ is satisfied by γ if relation $R_i \in \mathbb{D}$ and tuple $\gamma(\overline{X}_i) \in \mathbb{T}(R_i)$. Predicate $Y_j = Z_j$ is satisfied by γ if γ maps Y_j and Z_j to the same element of $\texttt{dom}$.

The result of evaluating unparameterized conjunctive query Q over database $\mathbb{D}$—denoted $Q^{\mathbb{D}}$—is a flat relation $(\mathbb{S}(Q^{\mathbb{D}}), \mathbb{T}(Q^{\mathbb{D}}))$. Schema $\mathbb{S}(Q^{\mathbb{D}})$ is deduced from the query head by associating with each element[6] of $\overline{\mathcal{W}}$ a unique attribute name from $\texttt{aname}$. Instance $\mathbb{T}(Q^{\mathbb{D}})$ is the set of tuples for which there exists a supporting embedding of $\texttt{body}_Q$ into $\mathbb{D}$.

$$\mathbb{T}(Q^{\mathbb{D}}) := \{\gamma(\overline{\mathcal{W}}) \mid \gamma : Q \to \mathbb{D}\} \tag{2.20}$$

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

$$\mathbb{T}(Q^{\mathbb{D}}|_\upsilon) := \{\gamma(\overline{\mathcal{W}}) \mid \gamma : Q \to \mathbb{D} \wedge \gamma(\overline{\mathcal{P}}) = \upsilon(\overline{\mathcal{P}})\} \tag{2.21}$$

(Within equation 2.21, $\overline{\mathcal{P}}$ denotes an arbitrary sequencing of set $\mathcal{P}$ 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. $\texttt{body}_Q$ *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 $v \in \texttt{body}_Q$, we use* $\texttt{body}_v$ *to denote the CQ$^=$ body occurring within vertex v,* $\texttt{body}_v^{\texttt{anc}}$ *to denote the CQ$^=$ body formed by conjoining the CQ$^=$ bodies of all of the ancestors of v, and* $\texttt{body}_v^{+}$ *to denote the CQ$^=$ body formed by conjoining* $\texttt{body}_v$ *with* $\texttt{body}_v^{\texttt{anc}}$*; the sets* $\texttt{atoms}_v$, $\texttt{atoms}_v^{\texttt{anc}}$, $\texttt{atoms}_v^{+}$, $\texttt{pred}_v$, $\texttt{pred}_v^{\texttt{anc}}$, *and* $\texttt{pred}_v^{+}$, *are defined analogously.*
2. *Sets $\mathcal{B}$, $\mathcal{C}_Q \subset \mathcal{C}$, $\mathcal{P}$, and $\mathcal{L}$ are defined the same as for CQ$^=$s. Because the overall query Q is parameter-free, set $\mathcal{P} = \varnothing$ and therefore $\mathcal{L} = \mathcal{B}$.*
3. *We use $\mathcal{B}_v$ to denote the set of variables occurring anywhere in* $\texttt{body}_v$*, with $\mathcal{B}_v^{\texttt{anc}}$ and $\mathcal{B}_v^{+}$ likewise corresponding to* $\texttt{body}_v^{\texttt{anc}}$ *and* $\texttt{body}_v^{+}$*. We use $\mathcal{P}_v$ to denote the set of designated parameter variables in* $\texttt{body}_v$*, and we require that $\mathcal{P}_v := \mathcal{B}_v \cap \mathcal{B}_v^{\texttt{anc}}$. Similarly, we use $\mathcal{L}_v := \mathcal{B}_v \setminus \mathcal{B}_v^{\texttt{anc}}$ to denote the set of local variables of* $\texttt{body}_v$*, and for any two distinct vertices u, v in* $\texttt{body}_Q$*, we require that local variables $\mathcal{L}_u$ and $\mathcal{L}_v$ be disjoint. As a consequence, for each variable*

[6] Technically, the attribute names in a relation schema must be unique (per Definitions 2.1.12 and 2.1.13) and so if $\overline{\mathcal{W}}$ 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.

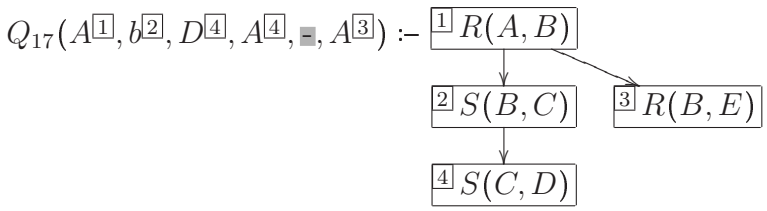


Figure 2.11: Hierarchical conjunctive query Q_{17}

$X \in \mathcal{B}$ there exists precisely one node u such that $X \in \mathcal{L}_u$, and X only occurs within the subtree rooted at u. We use $\mathcal{P}_v^{\texttt{anc}}$ 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 $\texttt{body}_Q$.

$$\mathcal{P}_v^{\texttt{anc}} := \bigcup_{u \in \texttt{ancestors}(v)} \mathcal{P}_u \qquad \mathcal{P}_v^+ := \mathcal{P}_v^{\texttt{anc}} \cup \mathcal{P}_v \qquad \mathcal{P}_Q^+ := \bigcup_{u \in \texttt{body}_Q} \mathcal{P}_u$$

(The analogous sets $\mathcal{L}_v^{\texttt{anc}}$, $\mathcal{L}_v^+$, and $\mathcal{L}_Q^+$ would be redundant, because the above definitions imply $\mathcal{L}_v^{\texttt{anc}} = \mathcal{B}_v^{\texttt{anc}}$, $\mathcal{L}_v^+ = \mathcal{B}_v^+$, and $\mathcal{L}_Q^+ = \mathcal{B}$.)

4. *The head tuple $\overline{\mathcal{W}}$ is composed from elements of $\mathcal{B} \cup \mathcal{C} \cup \{\text{-}\}$ (that is, variables, constants, and the special - constant). We use $\mathcal{W}$ to denote the set of elements occurring within $\overline{\mathcal{W}}$.*
5. *Each component of head tuple $\overline{\mathcal{W}}$ that is not the - constant is labelled with a vertex $v \in \texttt{body}_Q$. We write $W_i^{\boxed{v}}$ to denote that the i^{th} component of $\overline{\mathcal{W}}$ is labelled with vertex v. If $W_i \in \mathcal{C}$, then v is allowed to be any vertex in $\texttt{body}_Q$. 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 $\overline{\mathcal{W}}$ in which each position L_i contains the label of the corresponding head component W_i (or is empty if $W_i = \text{-}$). 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 $\texttt{body}_Q$ the equality predicates in $\texttt{body}_v$ 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 $\overline{L} = \langle \boxed{1}, \boxed{2}, \boxed{4}, \boxed{4}, , \boxed{3} \rangle$ and head label set $L = \{\boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}\}$.

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

Let $\texttt{trees}(T)$ 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 $\texttt{trees}(T)$ contains at least the root of T. We now restrict our attention to trees in the set $\texttt{trees}(\texttt{body}_Q)$. For each such tree T, define $\mathcal{B}_T \coloneqq \cup_{v \in T} \mathcal{B}_v$ and define $\mathcal{L}_T$, $\texttt{atoms}_T$, and $\texttt{pred}_T$ 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 $T \in \texttt{trees}(\texttt{body}_Q)$, let $\texttt{body}_T^+$ denote the CQ$^=$ body formed by conjoining the CQ$^=$ bodies of all vertices within T. An *embedding* of $T \in \texttt{trees}(\texttt{body}_Q)$ into $\mathbb{D}$—denoted $\gamma : T \to \mathbb{D}$—is a mapping $\gamma : \mathcal{B}_T \to \texttt{dom}$ (extended with identity on the constants in $\mathcal{C}$) that is an embedding of $\texttt{body}_T^+$ into $\mathbb{D}$. Observe that in the degenerate case where $\texttt{body}_Q$ is a single vertex, this definition of an embedding corresponds to the definition given earlier for CQ$^=$s. We use $\texttt{body}_Q^+$ to denote the single CQ$^=$ formed by conjoining the CQ$^=$s from all of the vertices in $\texttt{body}_Q$, which corresponds to the maximal case where T contains all of the vertices within $\texttt{body}_Q$.

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

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

$$\hat{\gamma}(W_i) \coloneqq \begin{cases} \gamma(W_i) & \text{if } \exists v \in T \text{ such that } W_i \text{ is labelled with } v \\ \texttt{-} & \text{otherwise (includes case } W_i = \texttt{-}) \end{cases} \tag{2.22}$$

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

$$\mathbb{T}(Q^{\mathbb{D}}) \coloneqq \{\hat{\gamma}(\overline{\mathcal{W}}) \mid \exists T \in \texttt{trees}(\texttt{body}_Q), \gamma : T \rightsquigarrow \mathbb{D}\} \tag{2.23}$$

As a shorthand notation, we use $\gamma : Q \rightsquigarrow \mathbb{D}$ to denote that γ is a terminal embeddings of $\texttt{body}_Q$, and so Equation 2.23 can be rewritten as follows.

$$\mathbb{T}(Q^{\mathbb{D}}) \coloneqq \{\hat{\gamma}(\overline{\mathcal{W}}) \mid \gamma : Q \rightsquigarrow \mathbb{D}\} \tag{2.24}$$

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

R

col_1	col_2
1	2
2	3
2	4
3	4
3	5
5	7
6	8

S

col_1	col_2
2	4
4	5
5	6
7	8

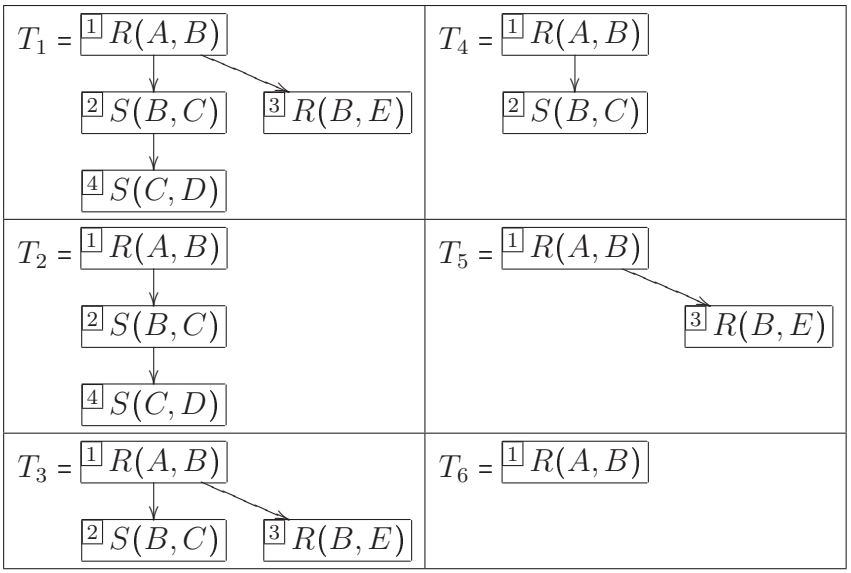


Figure 2.12: The six trees in $\texttt{trees}(Q_{17})$

The set $\texttt{trees}(Q_{17})$ contains the six trees $T_1, \ldots, T_6$ shown in Figure 2.12. For each $i \in [1,6]$, let Γ^i denote the set of embeddings of T_i into $\mathbb{D}_2$. In Figure 2.13 we tabularly depict each Γ^i and we have shaded rows corresponding to terminal embeddings. Finally, let Γ denote the set of terminal embeddings of members of $\texttt{trees}(\texttt{body}_Q)$ into $\mathbb{D}_2$. Relation $(Q_{17})^{\mathbb{D}_2}$ is obtained by duplicate-eliminating projection of columns $A^{[1]}$, $b^{[2]}$, $D^{[4]}$, $A^{[4]}$, - , and $A^{[3]}$ from "relation" Γ shown in Figure 2.14.

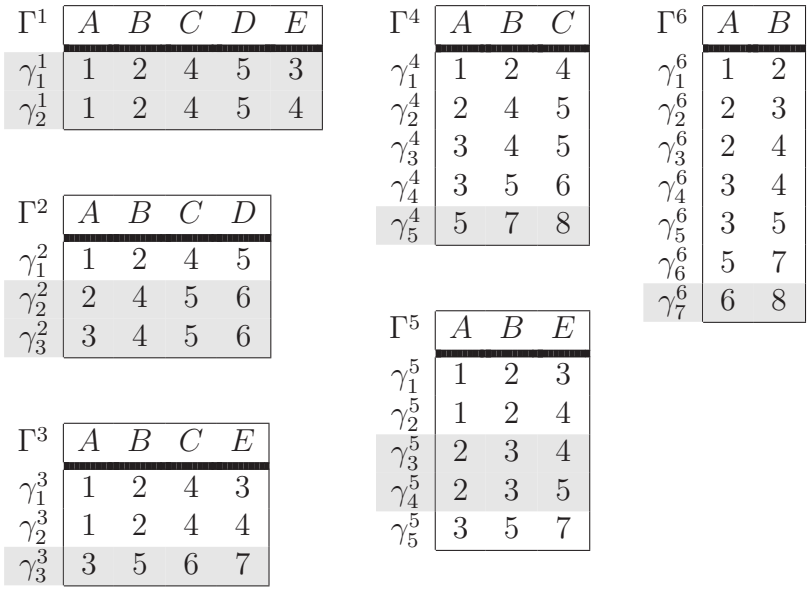
Definition 2.3.3 (Encoding Query (CEQ or HCEQ)) *A* hierarchical conjunctive encoding query Q *of depth* $d \geq 1$ *is an HCQ in which the head tuple* $\overline{\mathcal{W}}$ *is partitioned into* $d+1$ *sub-tuples as follows:*

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

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

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

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



Γ^1	A	B	C	D	E
γ_1^1	1	2	4	5	3
γ_2^1	1	2	4	5	4

Γ^2	A	B	C	D
γ_1^2	1	2	4	5
γ_2^2	2	4	5	6
γ_3^2	3	4	5	6

Γ^3	A	B	C	E
γ_1^3	1	2	4	3
γ_2^3	1	2	4	4
γ_3^3	3	5	6	7

Γ^4	A	B	C
γ_1^4	1	2	4
γ_2^4	2	4	5
γ_3^4	3	4	5
γ_4^4	3	5	6
γ_5^4	5	7	8

Γ^5	A	B	E
γ_1^5	1	2	3
γ_2^5	1	2	4
γ_3^5	2	3	4
γ_4^5	2	3	5
γ_5^5	3	5	7

Γ^6	A	B
γ_1^6	1	2
γ_2^6	2	3
γ_3^6	2	4
γ_4^6	3	4
γ_5^6	3	5
γ_6^6	5	7
γ_7^6	6	8

Figure 2.13: Embeddings of $\texttt{trees}(Q_{17})$ into database $\mathbb{D}_2$

Γ	A	B	C	D	E	b	$A^{\boxed{1}}$	$b^{\boxed{2}}$	$D^{\boxed{4}}$	$A^{\boxed{4}}$	-	$A^{\boxed{3}}$
γ_1^1	1	2	4	5	3	b	1	b	5	1	-	1
γ_2^1	1	2	4	5	4	b	1	b	5	1	-	1
γ_3^5	2	3			4	b	2	-	-	-	-	2
γ_4^5	2	3			5	b	2	-	-	-	-	2
γ_2^2	2	4	5	6		b	2	b	6	2	-	-
γ_3^2	3	4	5	6		b	3	b	6	3	-	-
γ_3^3	3	5	6		7	b	3	b	-	-	-	3
γ_5^4	5	7	8			b	5	b	-	-	-	-
γ_7^6	6	8				b	6	-	-	-	-	-

Figure 2.14: Terminal embeddings and their partial applications to $\overline{\mathcal{W}}$

$\hat{\mathcal{I}}_i$ to denote the set of *labelled* constants/variables composing $\overline{\mathcal{I}}_i$, and we use $\mathcal{I}_i$ to denote the set of *unlabelled* constants/variables composing $\overline{\mathcal{I}}_i$. We further define tuple $\overline{\mathcal{I}}_{[i,j]}$ 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 $\mathbb{D}$ the relation $Q^{\mathbb{D}}$ 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 $V_i^{\boxed{v}}$ of $\overline{\mathcal{V}}$ where $V_i \in \mathcal{B}$ (i.e., V_i is a variable), index set $\hat{\mathcal{I}}_{[1,d]}$ contains $V_i^{\boxed{v}}$.*
2. *For every component $V_i^{\boxed{v}}$ of $\overline{\mathcal{V}}$ where $V_i \in \mathcal{C}$ (i.e., V_i is a constant) and v is a* non-root *vertex of $\mathtt{body}_Q$, index set $\hat{\mathcal{I}}_{[1,d]}$ contains an index labelled with vertex v (i.e., $\exists X.(X^{\boxed{v}} \in \hat{\mathcal{I}}_{[1,d]})$).*

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 $\mathbb{D}$ the relation $Q^{\mathbb{D}}$ satisfies the functional dependency $\overline{\mathcal{I}}_{[1,d]} \rightarrow \overline{\mathcal{V}}$.

Consider any two terminal embeddings $\gamma_1, \gamma_2 : Q \rightsquigarrow \mathbb{D}$ that satisfy $\hat{\gamma}_1(\overline{\mathcal{I}}_{[1,d]}) = \hat{\gamma}_2(\overline{\mathcal{I}}_{[1,d]})$. Consider in turn each element $V_i^{\boxed{v}}$ of tuple $\overline{\mathcal{V}}$. If V_i is a variable then $\hat{\gamma}_1(V_i^{\boxed{v}}) = \hat{\gamma}_2(V_i^{\boxed{v}})$ follows trivially from $V_i^{\boxed{v}} \in \hat{\mathcal{I}}_{[1,d]}$. Otherwise, V_i must be a constant. If v is the root vertex of $\mathtt{body}_Q$, it follows from the definition of partial application that $\hat{\gamma}_1(V_i^{\boxed{v}}) = \hat{\gamma}_2(V_i^{\boxed{v}}) = V_i$. Otherwise v is a non-root vertex of $\mathtt{body}_Q$, and so $\hat{\mathcal{I}}_{[1,d]}$ must contain some index $X^{\boxed{v}}$. It again follows from the definition of partial application that if $\hat{\gamma}(X^{\boxed{v}}) \in \mathtt{dom}$ then $\hat{\gamma}_1(V_i^{\boxed{v}}) = \hat{\gamma}_2(V_i^{\boxed{v}}) = V_i$, whereas if $\hat{\gamma}(X^{\boxed{v}}) = \blacksquare$ then $\hat{\gamma}_1(V_i^{\boxed{v}}) = \hat{\gamma}_2(V_i^{\boxed{v}}) = \blacksquare$. □

2.3.2 Reducing Equivalence to Encoding-Equivalence

In this section we reduce the problem of deciding equivalence of satisfiable unparameterized $\mathtt{COCQL}^{\{f_\mathtt{b}, f_\mathtt{s}, f_\mathtt{n}\}}$ 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 $\mathtt{COCQL}^{\{f_\mathtt{b}, f_\mathtt{s}, f_\mathtt{n}\}}$ 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,

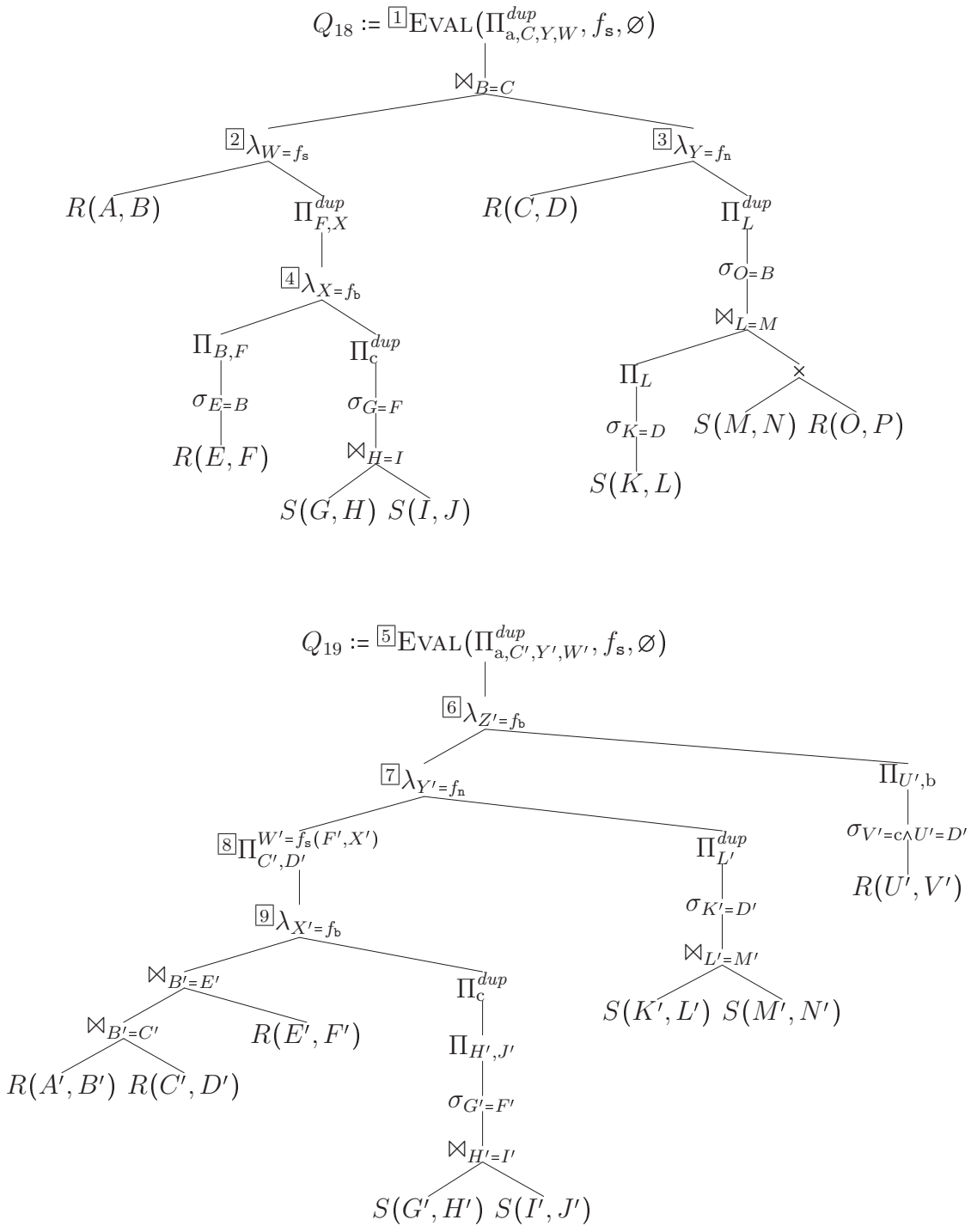


Figure 2.15: $\texttt{COCQL}^{\{f_{\mathtt{b}},f_{\mathtt{s}},f_{\mathtt{n}}\}}$ queries Q_{18} and Q_{19}

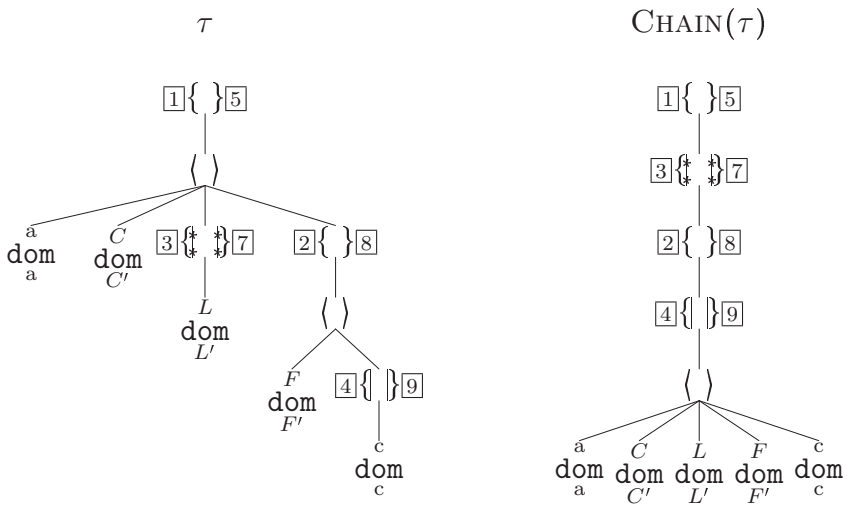


Figure 2.16: Output sort τ and its linearization $\textsc{Chain}(\tau)$ = $(\texttt{snsb}, 5)$

both Q_{18} and Q_{19} output objects conforming to the sort τ shown in Figure 2.16. We have labelled the atomic sorts in τ with the corresponding output variables/constants from Q_{18} (above) and Q_{19} (below). We have labelled the collection sorts in τ with the corresponding node numbers from Q_{18} (left) and Q_{19} (right). Now consider evaluating Q_{18} and Q_{19} over database $\mathbb{D}_3$ containing the relations R and S below. To aid in visualizing the database instance, we also provide a graphical depiction of $\mathbb{D}_3$.

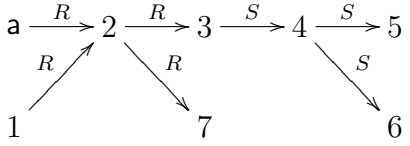
R

$\texttt{col}_1$	$\texttt{col}_2$
a	2
1	2
2	3
2	7

S

$\texttt{col}_1$	$\texttt{col}_2$
3	4
4	5
4	6

$\mathbb{D}_3$:=



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 τ; observe that it conforms to the chain sort $\textsc{Chain}(\tau)$ = $(\texttt{snsb}, 5)$ shown in Figure 2.16.

We will now describe a function Encode that maps satisfiable unparameterized $\texttt{COCQL}^{\{f_\text{b}, f_\text{s}, f_\text{n}\}}$ queries to hierarchical conjunctive encoding queries. Given an arbitrary satisfiable $\texttt{COCQL}^{\{f_\text{b}, f_\text{s}, f_\text{n}\}}$ query Q, the encoding query $\textsc{Encode}(Q)$ 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 Eval as well as each λ), as we have done to query Q_{18} in Figure 2.15.

2. For each node numbered $\boxed{i}$, create an algebraic expression E_i as follows:

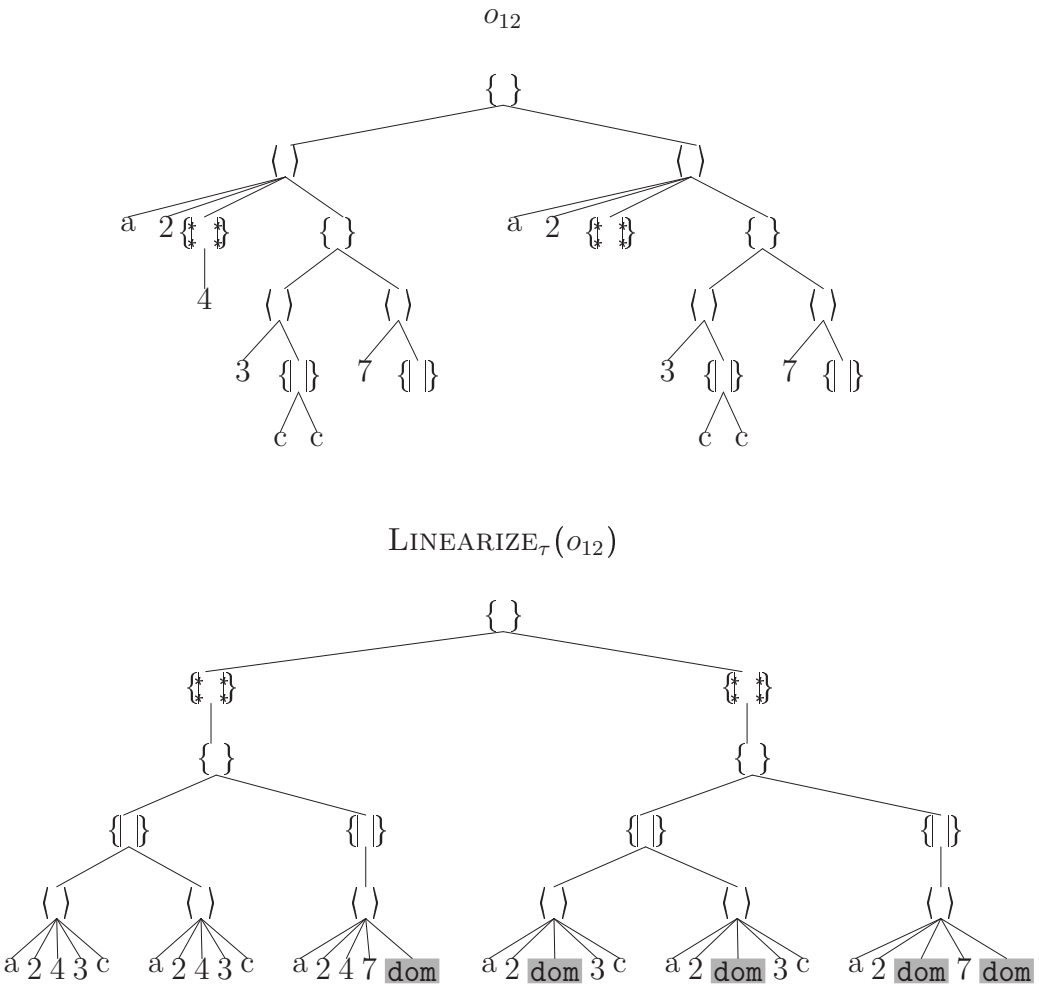


Figure 2.17: Query result $o_{12} = (Q_{18})^{\mathbb{D}_3} = (Q_{19})^{\mathbb{D}_3}$ and its linearization w.r.t. τ

(a) Initialize E_i with the algebraic expression whose result is the input to the aggregation function at node $\boxed{i}$ (i.e., the right input if node $\boxed{i}$ is an λ operator).

(b) Traverse E_i depth-first starting from its root.
- Whenever a scalar subquery operator (λ) is encountered, delete that operator and its entire right input expression, and continue traversing the left input expression.
- Whenever a duplicate-preserving projection operator (Π^{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.

$$E_1 = \Pi_{\mathrm{a},A,B,C,D}(\sigma_{B=C}(R(A,B) \times R(C,D)))$$
$$E_2 = \Pi_{B,F}(\sigma_{E=B}(R(E,F)))$$
$$E_3 = \Pi_{L,M,N,O,P}(\sigma_{K=D \wedge O=B \wedge L=M}(S(K,L) \times S(M,N) \times R(O,P)))$$
$$E_4 = \Pi_{\mathrm{c},G,H,I,J}(\sigma_{G=F \wedge H=I}(S(G,H) \times S(I,J)))$$

3. To construct the body of $\textsc{Encode}(Q)$:

(a) From each expression E_i above, create a vertex labelled $\boxed{i}$ containing a $\mathrm{CQ}^{=}$ 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 λ node labelled $\boxed{j}$, then draw an edge from vertex $\boxed{i}$ to vertex $\boxed{j}$.

4. To construct the head of $\textsc{Encode}(Q)$:

(a) Let τ be the output sort of Q. Label the atomic sorts in τ 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 $\textsc{Chain}(\tau)$ (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 $\textsc{Chain}(sort)$ starting at the root.

ii. If the i^{th} collection type enumerated has label $\boxed{j}$, form $\overline{\mathcal{I}}_i$ from the output list of E_j by labelling each constant/variable with $\boxed{j}$.

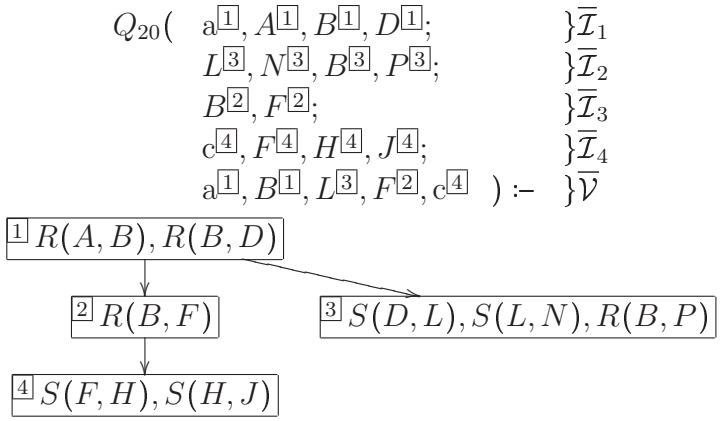


Figure 2.18: Encoding query $Q_{20} = \text{ENCODE}(Q_{18})$

For example, the collection types in $\text{CHAIN}(\tau)$ from query Q_{18} are enumerated in the order [1], [3], [2], [4], yielding the following indexes.

$$\overline{\mathcal{I}}_1 = \text{a}^{[1]}, A^{[1]}, B^{[1]}, C^{[1]}, D^{[1]} \quad \overline{\mathcal{I}}_2 = L^{[3]}, M^{[3]}, N^{[3]}, O^{[3]}, P^{[3]}$$
$$\overline{\mathcal{I}}_3 = B^{[2]}, F^{[2]} \quad \overline{\mathcal{I}}_4 = \text{c}^{[4]}, G^{[4]}, H^{[4]}, I^{[4]}, J^{[4]}$$

(c) The output list is constructed from τ 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 $\overline{\mathcal{V}} = \text{a}^{[1]}, C^{[1]}, L^{[3]}F^{[2]}, \text{c}^{[4]}$.

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 $\texttt{pred}_v \vDash X_i = Y$ 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 $X_i^{[v]} \in \hat{\mathcal{I}}_{[1,d]}$, delete all but the *left-most* occurrence of $X_i^{[v]}$ from the index sequence $\overline{\mathcal{I}}_{[1,d]}$.

Figure 2.18 shows the encoding query Q_{20} which is the final result of our running example $\text{ENCODE}(Q_{18})$.

We now extend the definition of $\text{ENCODE}(Q)$ to allow Q to contain generalized projection operators. Handling generalized projection operators directly is desirable because it simplifies both queries Q and $\text{ENCODE}(Q)$. Generalized projection occurs much more frequently than scalar aggregation in real workloads, and so restricting our attention to $\texttt{COCQL}^{\mathcal{F}-}$ expressions will allow for a simpler decision procedure to test equivalence for this common class of queries. (Specifically, reasoning tools presented in Chapter 3 for CEQs will suffice, rather than the more

general methods for HCEQs required by arbitrary $\texttt{COCQL}^{\mathcal{F}}$ queries.) We make the following modifications to $\textsc{Encode}(Q)$:

1. 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.

2. During the traversal of each expression E_i, whenever a generalized projection operator is encountered, delete from the attribute list the aggregation expression $X = f(A_{j_1}, \ldots, A_{j_l})$ (effectively converting the operator into duplicate-eliminating projection) and continue traversing its input expression. For example, query Q_{19} yields the following expressions.

$$E_5 = \Pi_{\mathrm{a},C',D'}(\sigma_{B'=C' \wedge B'=E'}(R(A',B') \times R(C',D') \times R(E',F')))$$
$$E_6 = \Pi_{U',\mathrm{b}}(\sigma_{V'=\mathrm{c} \wedge U'=D'}(R(U',V')))$$
$$E_7 = \Pi_{K',L',M',N'}(\sigma_{K'=D' \wedge L'=M'}(S(K',L') \times S(M',N')))$$
$$E_8 = \Pi_{A',B',C',D',E',F'}(\sigma_{B'=C' \wedge B'=E'}(R(A',B') \times R(C',D') \times R(E',F')))$$
$$E_9 = \Pi_{\mathrm{c},H',J'}(\sigma_{G'=F' \wedge H'=I'}(S(G',H') \times S(I',J')))$$

3. When creating the body of $\textsc{Encode}(Q)$, we ignore the expressions corresponding to generalized projection nodes (i.e, don't create $\mathrm{CQ}^{=}$ 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.

4. The head of $\textsc{Encode}(Q)$ is calculated from τ and each expression E_j almost as before, except that when constructing index tuple $\overline{\mathcal{I}}_i$ for collection type $\boxed{j}$, we label the components of $\overline{\mathcal{I}}_i$ with the label of the *nearest enclosing collection type within τ that is constructed by an λ operator*. For example, the collection types in query Q_{19} are enumerated in the order $\boxed{5}$, $\boxed{7}$, $\boxed{8}$, $\boxed{9}$, but type $\boxed{8}$ is constructed by a generalized projection operator whose nearest enclosing type within τ that is constructed by an λ operator is type $\boxed{5}$.

$$\overline{\mathcal{I}}_1 = \mathrm{a}^{\boxed{5}}, C'^{\boxed{5}}, D'^{\boxed{5}} \qquad \overline{\mathcal{I}}_2 = K'^{\boxed{7}}, L'^{\boxed{7}}, M'^{\boxed{7}}, N'^{\boxed{7}}$$
$$\overline{\mathcal{I}}_3 = A'^{\boxed{5}}, B'^{\boxed{5}}, C'^{\boxed{5}}, D'^{\boxed{5}}, E'^{\boxed{5}}, F'^{\boxed{5}} \qquad \overline{\mathcal{I}}_4 = \mathrm{c}^{\boxed{9}}, H'^{\boxed{9}}, J'^{\boxed{9}}$$

   Output attributes are labelled similarly, yielding $\overline{\mathcal{V}} = \mathrm{a}^{\boxed{5}}, C'^{\boxed{5}}, L'^{\boxed{7}} F'^{\boxed{5}}, \mathrm{c}^{\boxed{9}}$.

5. 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 $\textsc{Encode}(Q_{19})$. Although expression E_8 was not used when creating the body of Q_{21}, it was needed to determine the index tuple $\overline{\mathcal{I}}_3$ in the query head.

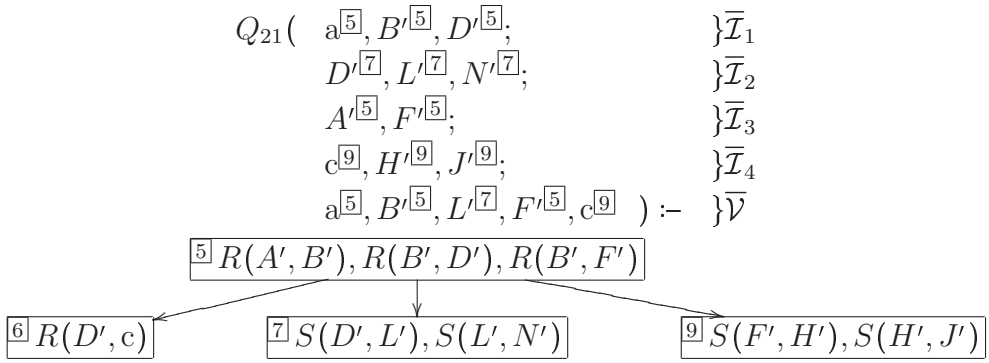


Figure 2.19: Encoding query Q_{21} = ENCODE(Q_{19})

Theorem 2.3.6 *Given any satisfiable unparameterized* COCQL$^{\{f_\mathrm{b}, f_\mathrm{s}, f_\mathrm{n}\}}$ *query Q, the query* ENCODE(Q) *is a valid encoding query.*

PROOF. When $\mathtt{body}_{\textsc{Encode}(Q)}$ is constructed by the algorithm ENCODE(Q), the λ 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 $\mathtt{body}_{\textsc{Encode}(Q)}$). This means that each labelled node in $\mathtt{body}_Q$ is only ever visited once, and hence every non-root node of $\mathtt{body}_{\textsc{Encode}(Q)}$ 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 λ 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, $\mathtt{body}_{\textsc{Encode}(Q)}$ is a proper HCQ body (as per Definition 2.3.2).

When $\mathtt{head}_{\textsc{Encode}(Q)}$ is constructed by algorithm ENCODE(Q), the variables in the head originate from attribute references in Q. Because Q is unparameterized, every variable in $\mathtt{head}_{\textsc{Encode}(Q)}$ will be local to $\mathtt{body}_{\textsc{Encode}(Q)}$. In order for $\mathtt{head}_{\textsc{Encode}(Q)}$ to contain some variable $W_i^{\boxed{v}}$, 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 $\mathtt{body}_Q$. The construction of $\mathtt{body}_{\textsc{Encode}(Q)}$ 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 $\mathtt{body}_{\textsc{Encode}(Q)}$. Hence, $\mathtt{head}_{\textsc{Encode}(Q)}$ satisfies Definitions 2.3.2 and 2.3.3 and so ENCODE(Q) is a proper HCEQ.

Finally, we need to establish that ENCODE(Q) is *valid* in that it satisfies Definition 2.3.4. Algorithm 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, ENCODE(Q) is valid. □

Theorem 2.3.7 *Given any satisfiable unparameterized* $\texttt{COCQL}^{\{f_b,f_s,f_n\}-}$ *query* Q*, the query* $\textsc{Encode}(Q)$ *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 $\textsc{Encode}(Q)$ avoids introducing hierarchical edges into the query body for generalized projection operators. □

Example 22 Consider the encoding queries $Q_{20} = \textsc{Encode}(Q_{18})$ from Figure 2.18 and $Q_{21} = \textsc{Encode}(Q_{19})$ 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 $\mathbb{D}_3$ from Example 21. The reader can verify that decoding either relation with the signature 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 $\textsc{Linearize}_\tau(o_{12})$ 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 $\widehat{\textsc{Chain}}$ 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 $\widehat{\textsc{Linearize}}_\tau$ as follows

$$\widehat{\textsc{Linearize}}_\tau(o) := \widehat{\textsc{Chain}} \circ \textsc{Complete}_\tau(o) \tag{2.25}$$

which then satisfies $o_{13} = \widehat{\textsc{Linearize}}_\tau(o_{12})$. It follows from Observation 2.1.6 that for each tuple sub-object within $\textsc{Complete}_\tau(o)$, the roots of the tuple components are either all shaded or all unshaded. Given sort τ and object $\widehat{\textsc{Linearize}}_\tau(o)$, 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* τ *not containing any zero-ary tuple sorts,* $\widehat{\textsc{Linearize}}_\tau$ *has a retraction.*

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

$$o = o' \iff \widehat{\textsc{Linearize}}_\tau(o) = \widehat{\textsc{Linearize}}_\tau(o')$$

Theorem 2.3.10 *Given any finite database* $\mathbb{D}$ *and any* $\texttt{COCQL}^{\{f_b,f_s,f_n\}}$ *query* Q *with output sort* τ *not containing any zero-ary tuples, let* $\overline{\S}$ *be the signature corresponding to the linearization of* τ*. Then, the complex object* $(Q)^{\mathbb{D}}$ *and the encoding relation* $(\textsc{Encode}(Q))^{\mathbb{D}}$ *are related as follows.*

$$\textsc{Decode}((\textsc{Encode}(Q))^{\mathbb{D}}, \overline{\S}) = \widehat{\textsc{Linearize}}_\tau((Q)^{\mathbb{D}}) \tag{2.26}$$

R_8

$\mathrm{a}^{\boxed{1}}$	$A^{\boxed{1}}$	$B^{\boxed{1}}$	$D^{\boxed{1}}$	$L^{\boxed{3}}$	$N^{\boxed{3}}$	$B^{\boxed{3}}$	$P^{\boxed{3}}$	$B^{\boxed{2}}$	$F^{\boxed{2}}$	$\mathrm{c}^{\boxed{4}}$	$F^{\boxed{4}}$	$H^{\boxed{4}}$	$J^{\boxed{4}}$	$\mathrm{a}^{\boxed{1}}$	$B^{\boxed{1}}$	$L^{\boxed{3}}$	$F^{\boxed{2}}$	$\mathrm{c}^{\boxed{4}}$
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	3	2	7	-	-	-	-	a	2	4	7	-
a	a	2	3	4	6	2	7	2	3	c	3	4	5	a	2	4	3	c
a	a	2	3	4	6	2	7	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	7	-	-	-	-	2	3	c	3	4	5	a	2	-	3	c
a	a	2	7	-	-	-	-	2	3	c	3	4	6	a	2	-	3	c
a	a	2	7	-	-	-	-	2	7	-	-	-	-	a	2	-	7	-
a	1	2	3	4	5	2	3	2	3	c	3	4	5	a	2	4	3	c
a	1	2	3	4	5	2	3	2	3	c	3	4	6	a	2	4	3	c
a	1	2	3	4	5	2	3	2	7	-	-	-	-	a	2	4	7	-
a	1	2	3	4	5	2	7	2	3	c	3	4	5	a	2	4	3	c
a	1	2	3	4	5	2	7	2	3	c	3	4	6	a	2	4	3	c
a	1	2	3	4	5	2	7	2	7	-	-	-	-	a	2	4	7	-
a	1	2	3	4	6	2	3	2	3	c	3	4	5	a	2	4	3	c
a	1	2	3	4	6	2	3	2	3	c	3	4	6	a	2	4	3	c
a	1	2	3	4	6	2	3	2	7	-	-	-	-	a	2	4	7	-
a	1	2	3	4	6	2	7	2	3	c	3	4	5	a	2	4	3	c
a	1	2	3	4	6	2	7	2	3	c	3	4	6	a	2	4	3	c
a	1	2	3	4	6	2	7	2	7	-	-	-	-	a	2	4	7	-
a	1	2	7	-	-	-	-	2	3	c	3	4	5	a	2	-	3	c
a	1	2	7	-	-	-	-	2	3	c	3	4	6	a	2	-	3	c
a	1	2	7	-	-	-	-	2	7	-	-	-	-	a	2	-	7	-

Figure 2.20: Encoding query result $R_8 = (Q_{20})^{\mathbb{D}_3}$ satisfying $\textsc{Decode}(R_8, \texttt{snsb}) = o_{13}$

R_9

a$^{\boxed{5}}$	$B'^{\boxed{5}}$	$D'^{\boxed{5}}$	$D'^{\boxed{7}}$	$L'^{\boxed{7}}$	$N'^{\boxed{7}}$	$A'^{\boxed{5}}$	$F'^{\boxed{5}}$	c$^{\boxed{9}}$	$H'^{\boxed{9}}$	$J'^{\boxed{9}}$	a$^{\boxed{5}}$	$B'^{\boxed{5}}$	$L'^{\boxed{7}}$	$F'^{\boxed{5}}$	c$^{\boxed{9}}$
a	2	3	3	4	5	a	3	c	4	5	*a*	2	4	3	c
a	2	3	3	4	5	a	3	c	4	6	*a*	2	4	3	c
a	2	3	3	4	5	a	7	-	-	-	*a*	2	4	7	-
a	2	3	3	4	5	1	3	c	4	5	*a*	2	4	3	c
a	2	3	3	4	5	1	3	c	4	6	*a*	2	4	3	c
a	2	3	3	4	5	1	7	-	-	-	*a*	2	4	7	-
a	2	3	3	4	6	a	3	c	4	5	*a*	2	4	3	c
a	2	3	3	4	6	a	3	c	4	6	*a*	2	4	3	c
a	2	3	3	4	6	a	7	-	-	-	*a*	2	4	7	-
a	2	3	3	4	6	1	3	c	4	5	*a*	2	4	3	c
a	2	3	3	4	6	1	3	c	4	6	*a*	2	4	3	c
a	2	3	3	4	6	1	7	-	-	-	*a*	2	4	7	-
a	2	7	-	-	-	a	3	c	4	5	a	2	-	3	c
a	2	7	-	-	-	a	3	c	4	6	a	2	-	3	c
a	2	7	-	-	-	a	7	-	-	-	a	2	-	7	-
a	2	7	-	-	-	1	3	c	4	5	a	2	-	3	c
a	2	7	-	-	-	1	3	c	4	6	a	2	-	3	c
a	2	7	-	-	-	1	7	-	-	-	a	2	-	7	-

Figure 2.21: Encoding query result $R_9 = (Q_{21})^{\mathbb{D}_3}$ satisfying $\textsc{Decode}(R_9, \texttt{snsb}) = o_{13}$

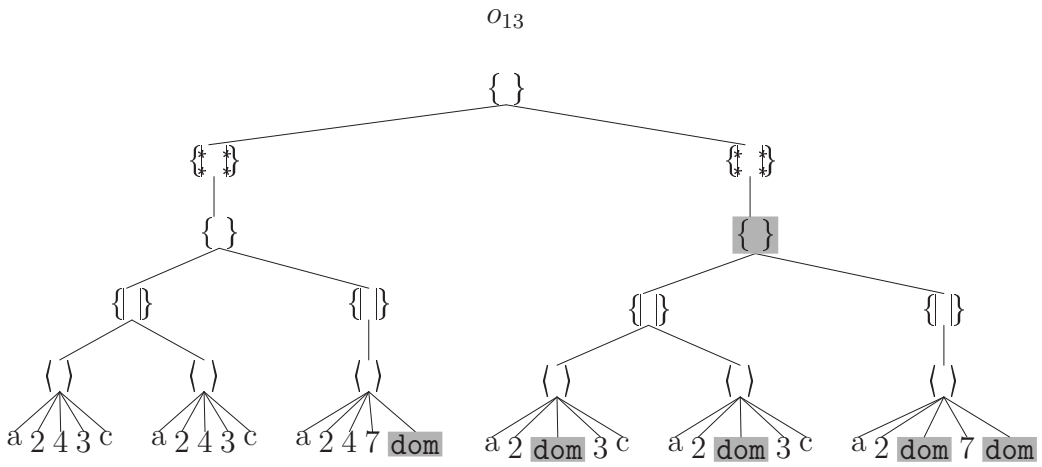


Figure 2.22: Query result o_{13} = DECODE($(Q_{20})^{\mathbb{D}_3}$, snsb) = DECODE($(Q_{21})^{\mathbb{D}_3}$, snsb)

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

If o is a trivial object, then LINEARIZE$_\tau(o)$ is one of {| |}, { }, or {{ }} (depending upon the first character of $\overline{\S}$). 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 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.

$$\widehat{\text{LINEARIZE}}_\tau(o) = \widehat{\text{CHAIN}} \circ \text{COMPLETE}_\tau(o) = o \tag{2.27}$$

The procedure ENCODE essentially partitions the algebraic tree of Q based upon the λ operators and reshapes 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 - symbol. In this simple case, procedure DECODE (Algorithm 3) is equivalent to procedure DECODESIMPLE (Algorithm 2). Because DECODESIMPLE essentially performs the same collection-construction operations as would be required to evaluate Q over $\mathbb{D}$, it is tedious but straightforward to verify the following.

$$\text{DECODE}(R, \overline{\S}) = \text{DECODESIMPLE}(R, \overline{\S}) = o \tag{2.28}$$

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 λ operator.

$$\widehat{\text{LINEARIZE}}_\tau(o) = \widehat{\text{CHAIN}} \circ \text{COMPLETE}_\tau(o) = \text{COMPLETE}_\tau(o) \tag{2.29}$$

In this case, function $\textsc{Complete}_\tau$ 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 $\mathbb{D}$ that assigns the value $\blacksquare$ to all of the variables in the index level corresponding to the shaded subobject. These embeddings yield index tuples satisfying the predicate $\overline{a} = \blacksquare^{|\overline{a}|}$ 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;

$$\widehat{\textsc{LinearIze}}_\tau(o) = \widehat{\textsc{Chain}} \circ \textsc{Complete}_\tau(o) \tag{2.30}$$

in this case, the effect of the $\textsc{Complete}_\tau$ function is mimicked by $\blacksquare$ values in R and in the decoding process as described above, but the effect of the $\widehat{\textsc{Chain}}$ function is non-trivial. The net effect of the $\widehat{\textsc{Chain}}$ function is three-fold: first, to linearize sibling collection types consistent with a pre-order traversal of τ (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 $\textsc{Encode}$ and $\textsc{Decode}$ 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 τ and when the signature $\overline{\S}$ used in the decoding process is calculated via a pre-order traversal of τ. 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 τ that correspond to collection types in $\textsc{Chain}(\tau)$ at depths i, j with $i < j$. Given any tuple that satisfies $\overline{\mathcal{I}}_{i-1} = \blacksquare^{|\overline{\mathcal{I}}_{i-1}|}$ but $\overline{\mathcal{I}}_j \neq \blacksquare^{|\overline{\mathcal{I}}_j|}$, the predicate test $\overline{\mathcal{I}}_{i-1} = \blacksquare^{|\overline{\mathcal{I}}_{i-1}|}$ 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 $\mathbb{S}(\mathbb{D})$ and a signature $\overline{\S}$, if for each relational database $\mathbb{D}$ with schema $\mathbb{S}(\mathbb{D})$ and finite instance $\mathbb{T}(\mathbb{D})$ the encoding-equality $(Q)^{\mathbb{D}} \doteq_{\overline{\S}} (Q')^{\mathbb{D}}$ holds, then we say that Q and Q' are* encoding-equivalent w.r.t. signature $\overline{\S}$ *(abbreviated* $\overline{\S}$-equivalent*, and denoted $Q \doteqdot_{\overline{\S}} Q'$).*

Theorem 2.3.12 *Given two satisfiable, unparameterized $\texttt{COCQL}^{\{f_\flat, f_\natural, f_\sharp\}}$ queries Q, Q' with the same output sort τ not containing zero-ary tuples,*

$$Q \equiv Q' \iff \textsc{Encode}(Q) \doteqdot_{\overline{\S}} \textsc{Encode}(Q')$$

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

Proof. Follows from the relationship between the output of queries Q and $\textsc{Encode}(Q)$ (Theorem 2.3.10) and the existence of a retraction for transformation $\widehat{\textsc{LinearIze}}_\tau$ (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 τ 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* $\mathtt{CSQL}^{\{f_\mathtt{b},f_\mathtt{s},f_\mathtt{n}\}}$ *queries is polynomial-time reducible to* $\overline{\S}$*-equivalence of hierarchical conjunctive encoding queries.*

PROOF. Follows from Proposition 2.1.22. □

Corollary 2.3.14 *Deciding equivalence of satisfiable* $\mathtt{CSQL}^{\{f_\mathtt{b},f_\mathtt{s},f_\mathtt{n}\}-}$ *queries is polynomial-time reducible to* $\overline{\S}$*-equivalence of conjunctive encoding queries.*

PROOF. Follows from Proposition 2.1.23 and Theorem 2.3.7. □

Example 23 The equivalence of $\mathtt{COCQL}^{\{f_\mathtt{b},f_\mathtt{s},f_\mathtt{n}\}}$ queries Q_{18} and Q_{19} can be decided by testing the condition $Q_{20} \doteq_{\mathtt{snsb}} Q_{21}$ (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 $f_\mathtt{n}$ with $f_\mathtt{b}$. The modified queries would not be equivalent, because Q_{20} and Q_{21} are not sbsb-equivalent. (The reader can easily verify from Figures 2.20 and 2.21 that $R_8 \not\doteq_{\mathtt{sbsb}} R_9$ because object $\text{DECODE}(R_8, \mathtt{sbsb})$ contains a sub-bag with cardinality four, while object $\text{DECODE}(R_9, \mathtt{sbsb})$ 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 $\texttt{COCQL}^{\{f_\text{b},f_\text{s},f_\text{n}\}}$ can be described informally as follows. The restricted subclass $\texttt{COCQL}^{\{f_\text{b},f_\text{s},f_\text{n}\}-}$ that we study in Chapter 3—queries that do not contain scalar subqueries 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 $\texttt{COCQL}^{\{f_b,f_s,f_n\}}$ is obtained.

Even though language $\texttt{COCQL}^{\mathcal{F}}$ 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 $\texttt{COCQL}^{\{f_s\}}$ out of some database of flat relations. Similarly, language $\texttt{COCQL}^{\{f_s\}-}$ is capable of constructing any object in class NL. This apparent discrepancy is due to the simple fact that $\texttt{COCQL}^{\{f_s\}-}$ 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 $\texttt{COCQL}^{\{f_b,f_s,f_n\}-}$ 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 $\texttt{COCQL}^{\{f_b,f_s,f_n\}-}$ 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. Sucui'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 a 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 $\texttt{COCQL}^{\mathcal{F}}$ that includes the ability to union collections, but not necessarily to a similar extension to $\texttt{COCQL}^{\mathcal{F}-}$ 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 $\texttt{COCQL}^{\{f_{\mathrm{s}}\}}$ 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 $Q(\overline{\mathcal{I}}_1;\ldots;\overline{\mathcal{I}}_d;\overline{\mathcal{V}})$ be a CQ whose head has been annotated to distinguish d sets of *index variables*, and define $\overline{\mathcal{I}} := (\overline{\mathcal{I}}_1;\ldots;\overline{\mathcal{I}}_d)$. Given two such queries, Q *simulates* Q' *to depth* d—denoted $Q \preceq_d Q'$—iff over every database instance the following equation holds:

$$\forall\overline{\mathcal{I}}_1.\exists\overline{\mathcal{I}}'_1 \ldots \forall\overline{\mathcal{I}}_d.\exists\overline{\mathcal{I}}'_d.\forall\overline{\mathcal{V}}\left[Q(\overline{\mathcal{I}};\overline{\mathcal{V}}) \Longrightarrow Q'(\overline{\mathcal{I}}';\overline{\mathcal{V}})\right] \tag{2.31}$$

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 $\texttt{COCQL}^{\{f_{\mathrm{s}}\}-}$ 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

Q_{22}: { SELECT $\{u.C\}$ FROM E AS x,
(SELECT $z.P$, $\{z.C\}$ AS C FROM E AS z
GROUP BY $z.P$) AS u
WHERE $x.C = u.P$ GROUP BY $x.C$ }

Q_{23}: { SELECT $\{u.C\}$ FROM E AS x, E AS y,
(SELECT $z.P$, $\{z.C\}$ AS C FROM E AS z
GROUP BY $z.P$) AS u
WHERE $x.C = u.P$ AND $y.C = u.P$ GROUP BY $x.P$, $y.P$ }

Q_{24}: { SELECT $\{C\}$ FROM E AS x,
(SELECT $z.P$, $\{z.C\}$ AS C
FROM E AS y, E AS z WHERE $y.C = z.P$
GROUP BY $y.P$, $z.P$) AS u
WHERE $x.C = u.P$ GROUP BY $x.P$ }

Figure 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 $Q \ll_d Q'$—iff:

$$\forall \overline{\mathcal{I}}_1 . \exists \overline{\mathcal{I}}'_1 \ldots \forall \overline{\mathcal{I}}_d . \exists \overline{\mathcal{I}}'_d . \forall \overline{\mathcal{V}} \left[Q(\overline{\mathcal{I}}; \overline{\mathcal{V}}) \iff Q'(\overline{\mathcal{I}}'; \overline{\mathcal{V}}) \right] \tag{2.32}$$

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 $\text{COCQL}^{\{f_s\}-}$ 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},

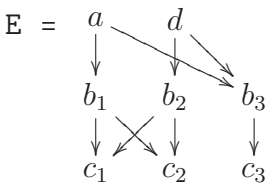


Figure 2.24: Database instance $\mathbb{D}_4$

Q'_{22}

A	B	C
a	b_1	c_1
a	b_1	c_2
a	b_3	c_3
d	b_2	c_1
d	b_2	c_2
d	b_3	c_3

Q'_{23}

AD	B	C
$a\ a$	b_1	c_1
$a\ a$	b_1	c_2
$a\ a$	b_3	c_3
$a\ d$	b_3	c_3
$d\ a$	b_3	c_3
$d\ d$	b_2	c_1
$d\ d$	b_2	c_2
$d\ d$	b_3	c_3

Q'_{24}

A	DB	C
a	$a\ b_1$	c_1
a	$a\ b_1$	c_2
a	$a\ b_3$	c_3
a	$d\ b_3$	c_3
d	$a\ b_3$	c_3
d	$d\ b_2$	c_1
d	$d\ b_2$	c_2
d	$d\ b_3$	c_3

Figure 2.25: Evaluating Q'_{22}, Q'_{23}, and Q'_{24} over $\mathbb{D}_4$

Q_{23}, and Q_{24} with the following indexed CQs.

$$
\begin{aligned}
&\overbrace{}^{\overline{\mathcal{I}}_1}\ ;\ \overbrace{}^{\overline{\mathcal{I}}_2}\ ;\overbrace{}^{\overline{\nu}}\\
&Q'_{22}(\ A\ ;\ B\ ;C\ ) \text{ :–} \mathtt{E}(A,B), \mathtt{E}(B,C)\\
&Q'_{23}(A,D;\ B\ ;C\ ) \text{ :–} \mathtt{E}(A,B), \mathtt{E}(B,C), \mathtt{E}(D,B)\\
&Q'_{24}(\ A\ ;D,B;C\ ) \text{ :–} \mathtt{E}(A,B), \mathtt{E}(B,C), \mathtt{E}(D,B)
\end{aligned}
$$

Consider the database $\mathbb{D}_4$ 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 $\mathbb{D}_4$ all six strong simulation conditions $Q'_{22} \ll_2 Q'_{23}$, $Q'_{23} \ll_2 Q'_{22}$, $Q'_{22} \ll_2 Q'_{24}$, $Q'_{24} \ll_2 Q'_{22}$, $Q'_{23} \ll_2 Q'_{24}$, and $Q'_{24} \ll_2 Q'_{23}$ 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 $\mathbb{D}_4$ 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 $\texttt{COCQL}^{\{f_\flat, f_\natural, f_\sharp\}\text{-}}$ expressions. This subclass is of practical interest because $\texttt{COCQL}^{\{f_\flat, f_\natural, f_\sharp\}\text{-}}$ includes 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 $\texttt{COCQL}^{\{f_\flat, f_\natural, f_\sharp\}\text{-}}$ 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 $\boxed{v}$ 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 $\mathbb{D}$ by unioning two copies of $\texttt{body}_Q$ 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, $\texttt{body}_Q$ is simply a set of atoms and can therefore be treated as a set of canonical tuples.) The query result $(Q)^{\mathbb{D}}$ then violates the functional dependency $\overline{\mathcal{I}}_{[1,d]} \to \overline{\mathcal{V}}$, and so is not a valid encoding relation. □

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 $\overline{\S}$. 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 $Q(\overline{\mathcal{W}})$ and $Q'(\overline{\mathcal{W}}')$, 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. $h(\mathtt{body}_{Q'}) \subseteq \mathtt{body}_Q$; *and*
2. $h(\overline{\mathcal{W}}') = \overline{\mathcal{W}}$.

We write $h : Q' \to Q$ to denote that h is a homomorphism from Q' to Q. A satisfiable CQ $Q(\overline{\mathcal{W}})$ is *contained* under CQ $Q'(\overline{\mathcal{W}}')$ 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 $|\mathtt{body}_{Q'}| < |\mathtt{body}_Q|$. 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 $Q(\overline{\mathcal{W}})$, any mapping $h : \mathcal{B} \to \mathcal{B} \cup \mathcal{C}$ (extended with identity on constants) that satisfies $h(\mathcal{W}) = \mathcal{W}$ and $h(\mathtt{body}_Q) \subseteq \mathtt{body}_Q$ forms a bijection between $\mathcal{B}$ and itself.*

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

Implied Multivalued Dependencies

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

$$R = \Pi_{XY}(R) \bowtie_X \Pi_{XZ}(R) \tag{3.1}$$

Given an CQ Q that yields a relation over $\mathcal{W}$, we say that Q implies $X \twoheadrightarrow Y$—denoted $Q(\mathcal{W}) \vDash X \twoheadrightarrow Y$—if for every database $\mathbb{D}$ the relation $(Q)^{\mathbb{D}}$ satisfies $X \twoheadrightarrow Y$.

$$(Q)^{\mathbb{D}} = \Pi_{XY}\left((Q)^{\mathbb{D}}\right) \bowtie_X \Pi_{XZ}\left((Q)^{\mathbb{D}}\right) \tag{3.2}$$

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 $\mathbb{D}$. The following theorem formalizes this.

Theorem 3.1.3 $Q(\overline{\mathcal{W}}) \vDash X \twoheadrightarrow Y \iff Q \equiv \Pi_{XY}(Q) \bowtie_X \Pi_{XZ}(Q)$

Corollary 3.1.4 *$Q(\overline{\mathcal{W}}) \vDash X \twoheadrightarrow Y$ iff there exists a homomorphism $h : Q \to Q$ (thereby satisfying $h(\overline{\mathcal{W}}) = \overline{\mathcal{W}}$) such that $h(\mathtt{atoms}_Q)$ 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 $Q(\overline{\mathcal{W}})$ is equivalent to a constant selection operation over conjunctive query $Q'(\overline{\mathcal{W}})$*

$$Q \equiv \sigma_{\overline{A=a}}(Q')$$

then $Q \vDash X \twoheadrightarrow Y$ iff $Q' \vDash X \cup A \twoheadrightarrow Y \setminus A$.

Prof

school	dept	name
UW	CS	Frank
UW	CS	David
UW	CS	Ashraf
UW	ENG	Linda
UW	ENG	John

Student

school	dept	name
UW	CS	Dave
UW	CS	Xin
UW	ENG	Cara
UW	ENG	Julia

Figure 3.1: Database $\mathbb{D}_5 = \{\texttt{Prof}, \texttt{Student}\}$

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

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

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

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 $Q(\overline{\mathcal{W}})$ be a new conjunctive query whose output tuple satisfies $\mathcal{W} = \mathcal{B}_a \cup \{A, Z\}$, and whose body is defined as follows.

$$\texttt{body}_Q = \texttt{body}_{Q_a} \cup \texttt{body}_{Q_b} \cup \bigcup_{x \in \mathcal{B}_a \cup \mathcal{B}_b} \{R(A, x), R(x, Z)\}$$

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

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

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

Figure 3.1 shows a sample database instance $\mathbb{D}_5$, while Figure 3.2 shows the relation R_{10} resulting from evaluating either query Q_{25} or Q_{26} over $\mathbb{D}_5$. The reader can verify that relation R_{10} satisfies $A \twoheadrightarrow C$.

R_{10}

A	B	C
CS	Ashraf	Dave
CS	Ashraf	Xin
CS	David	Dave
CS	David	Xin
CS	Frank	Dave
CS	Frank	Xin
ENG	Linda	Cara
ENG	Linda	Julia
ENG	John	Cara
ENG	John	Julia

Figure 3.2: Query result $R_{10} = (Q_{25})^{\mathbb{D}_5} = (Q_{26})^{\mathbb{D}_5}$

Query Q_{27} embodies the appropriate query for testing directly—per Theorem 3.1.3—whether Q_{25} satisfies the MVD $A \twoheadrightarrow C$.

$$\begin{aligned} Q_{27}(A,B,C) \coloneq\ & \texttt{Prof}(\text{'UW'}, A, B), \texttt{Student}(\text{'UW'}, A, C') \\ & \texttt{Prof}(\text{'UW'}, A, B''), \texttt{Student}(\text{'UW'}, A, C) \end{aligned}$$

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

$$h \coloneq \{A/A, B/B, C/C\} \qquad h' \coloneq \{A/A, B/B, C/C, C'/C, B''/B\}$$

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

$$\begin{aligned} Q_{28}(A,B,C) \coloneq\ & \texttt{Prof}(\text{'UW'}, A, B), \texttt{Student}(\text{'UW'}, A, C'), \\ & \texttt{Prof}(F', D', B), \texttt{Student}(F', E', C') \\ & \texttt{Prof}(\text{'UW'}, A, B''), \texttt{Student}(\text{'UW'}, A, C), \\ & \texttt{Prof}(F'', D'', B''), \texttt{Student}(F'', E'', C) \end{aligned}$$

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

$$h'' \coloneq \{A/A, B/B, C/C, D/A, E/A, F/\text{'UW'}\}$$
$$h''' \coloneq \{A/A, B/B, B''/B, C/C, C'/C, D'/D, D''/D, E'/E, E''/E, F'/F, F''/F\}$$

Alternatively, we could have deduced that Q_{26} satisfies $A \twoheadrightarrow C$ 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 for 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 $A \twoheadrightarrow C$ because variables C and B are "separated" by variable A within $\texttt{body}_{Q_{25}}$. 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)-*path* in G is a sequence of $k \geq 1$ adjacent edges starting at s and ending at t. Given $X \subset U$ and $s, t \in (U \setminus X)$, we call X an (s,t)-*articulation set of* G if the subgraph $G|_{U \setminus X}$ does not contain any (s,t)-paths.[3] Given disjoint sets $X, S, T \subset U$, we call X a *weak* (S,T)-*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)-*articulation set* if it is an (s,t)-articulation set for every pair $s \in S$ and $t \in T$.

A (undirected) *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 *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 $f_j \cap f_{j+1} \neq \varnothing$ for $j \in [1, k-1]$ [6, Sect. 6.4]. 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.

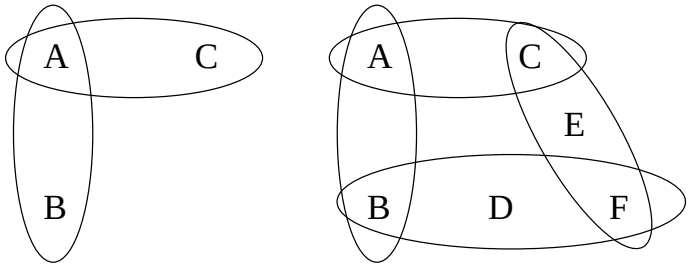
Example 26 Consider the two hypergraphs depicted in Figure 3.3. The set $\{A\}$ is a (B,C)-articulation set in H_{25} but not in H_{26}. Within H_{26}, the set $\{B,F\}$ is a weak (but not strong) (E, AD)-articulation set, while $\{C,F\}$ is a strong (E, AD)-articulation set.

Associated with each conjunctive query Q we define the *query hypergraph* H^Q as follows. The query variables $\mathcal{B}$ from Q form the vertex set of H^Q. The hyperedges correspond to atoms of $\texttt{body}_Q$ 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 $\mathcal{V}$ 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.



(a) H_{25} (b) H_{26}

Figure 3.3: Two hypergraphs

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

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

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

Theorem 3.1.10 *Given a* minimal *conjunctive query $Q(\overline{\mathcal{W}})$ and disjoint attribute sets $X, Y \subset \mathcal{W}$ query Q implies MVD $X \twoheadrightarrow Y$ 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 $h : Q \to Q$ is necessarily isomorphic over $\mathcal{B}$ 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 $A \twoheadrightarrow C$ 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 $\mathcal{I}_i^{\overline{\S}} \subseteq \mathcal{I}_i$ at each level i.

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

$$Q_{(i)}(\mathcal{I}_{[1,i]}\mathcal{I}_{[i+1,d]}^{\overline{\S}}) \text{ :- } \mathtt{body}_Q \tag{3.3}$$

Set $\mathcal{I}_i^{\overline{\S}}$ is a minimal *subset of $\mathcal{I}_i$ satisfying both $\mathcal{V} \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{I}_{[i,d]}^{\overline{\S}} \cup \mathcal{C}$ and the following signature-specific conditions.*

$\S_i$	**Condition**
b	$\mathcal{I}_i \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^{\overline{\S}} \cup \mathcal{C}$
s	$Q_{(i)} \vDash (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^{\overline{\S}}) \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$
n	$Q_{(i)} \vDash \mathcal{I}_{[1,i-1]} \twoheadrightarrow \mathcal{I}_{[i,d]}^{\overline{\S}}$

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

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

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

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

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

Proof. First, let $Q'_{(i)}(\mathcal{I}_{[1,i]}\mathcal{I}_{[i+1,d]}^{\overline{\S}})$ be a minimal CQ equivalent to $Q_{(i)}$ from equation 3.3. Next, let a "candidate for $\mathcal{I}_i^{\overline{\S}}$" denote any set $X \subseteq \mathcal{I}_i$ such that if we choose $\mathcal{I}_i^{\overline{\S}} := X$ 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 $\mathcal{I}_i^{\overline{\S}}$, then $X_1 \cap X_2$ is also a candidate for $\mathcal{I}_i^{\overline{\S}}$. 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 $\S_i$ = 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 $\S_i$ = s**:**
The MVD $Q_{(i)} \vDash \mathcal{I}_{[1,i-1]} \cup (X_1 \cap X_2) \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$ 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, $Q_{(i)} \vDash (\mathcal{I}_{[1,i-1]} \cup X_1) \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$. By Theorem 3.1.10, $\mathcal{I}_{[1,i-1]} \cup X_1$ is a strong $(\mathcal{I}_{[i+1,d]}^{\overline{\S}}, (\mathcal{I}_i \setminus X_1))$-articulation set of $H^{Q'_{(i)}}$.
2. By candidacy of X_2, $Q_{(i)} \vDash \mathcal{I}_{[1,i-1]} \cup X_2 \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$. By Theorem 3.1.10, $\mathcal{I}_{[1,i-1]} \cup X_2$ is a strong $(\mathcal{I}_{[i+1,d]}^{\overline{\S}}, (\mathcal{I}_i \setminus X_2))$-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 $\mathcal{I}_{[i+1,d]}^{\overline{\S}}$ 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 $\S_i = \mathtt{n}$**:**

Similar to case $\S_i = \mathtt{s}$. The candidacies 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 $\mathcal{I}_{[1,i-1]}$ from $H^{Q'_{(i)}}$ causes the four sets $X_1 \smallsetminus X_2$, $X_2 \smallsetminus X_1$, $\mathcal{I}_i \smallsetminus (X_1 \cup X_2)$, and $(X_1 \cap X_2) \cup \mathcal{I}^{\overline{\S}}_{[i+1,d]}$ to occur in separate partitions of the remaining hypergraph. The MVD $Q_{(i)} \vDash \mathcal{I}_{[1,i-1]} \twoheadrightarrow (X_1 \cap X_2) \cup \mathcal{I}^{\overline{\S}}_{[i+1,d]}$ then follows from Theorem 3.1.10.

□

We say that CEQ $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$ is in $\overline{\S}$*-normal form (*$\overline{\S}$*-NF)*if $\mathcal{I}_{[1,d]} = \mathcal{I}^{\overline{\S}}_{[1,d]}$, and so an arbitrary CEQ is converted to $\overline{\S}$-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 $(\mathcal{V} \cap \mathcal{B}) \subseteq \mathcal{I}^{\overline{\S}}_{[1,d]}$ and so the new query is guaranteed to be valid.

Theorem 3.1.16 *Performing* $\overline{\S}$*-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 $\S_i = \mathtt{b}$**:** Trivial.

Case $\S_i = \mathtt{s}$**:** 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 $\mathcal{I}_{[1,i-1]} \cup (\mathcal{I}_i \cap \mathcal{V})$. Identify any non-output members of $\mathcal{I}^{\overline{\S}}_i$ incrementally by traversing the connected components containing $\mathcal{I}_{[i+1,d]}$ and deleting the "nearest" member of $\mathcal{I}_i$.

Case $\S_i = \mathtt{n}$**:** 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 $\mathcal{I}_{[1,i-1]}$. Identify $\mathcal{I}^{\overline{\S}}_i$ by traversing the connected components containing any variable in $(\mathcal{I}_i \cap \mathcal{V}) \cup \mathcal{I}^{\overline{\S}}_{[i+1,d]}$.

□

Example 29 Consider the following four CEQs.

$$
\begin{array}{lll}
 & \overbrace{}^{\overline{\mathcal{I}}_1} \; \overbrace{}^{\overline{\mathcal{I}}_2} \; \overbrace{}^{\overline{\mathcal{V}}} & \\
Q_{29}(A,B\,;\; C \;;C) & \coloneq & \mathtt{Prof}(\text{`UW'}, A, B), \mathtt{Student}(\text{`UW'}, A, C) \\
Q_{30}(\; A \;;\; C \;;C) & \coloneq & \mathtt{Prof}(\text{`UW'}, A, B), \mathtt{Student}(\text{`UW'}, A, C) \\
Q_{31}(\; A \;;B,C;C) & \coloneq & \mathtt{Prof}(\text{`UW'}, A, B), \mathtt{Student}(\text{`UW'}, A, C), \\
 & & \mathtt{Prof}(F, D, B), \mathtt{Student}(F, E, C) \\
Q_{32}(\; A \;;\; C \;;C) & \coloneq & \mathtt{Prof}(\text{`UW'}, A, B), \mathtt{Student}(\text{`UW'}, A, C), \\
 & & \mathtt{Prof}(F, D, B), \mathtt{Student}(F, E, C)
\end{array}
$$

R_{11}

A	B	C	C
CS	Ashraf	Dave	Dave
CS	Ashraf	Xin	Xin
CS	David	Dave	Dave
CS	David	Xin	Xin
CS	Frank	Dave	Dave
CS	Frank	Xin	Xin
ENG	Linda	Cara	Cara
ENG	Linda	Julia	Julia
ENG	John	Cara	Cara
ENG	John	Julia	Julia

R_{12}

A	B	C	C
CS	Ashraf	Dave	Dave
CS	Ashraf	Xin	Xin
CS	David	Dave	Dave
CS	David	Xin	Xin
CS	Frank	Dave	Dave
CS	Frank	Xin	Xin
ENG	Linda	Cara	Cara
ENG	Linda	Julia	Julia
ENG	John	Cara	Cara
ENG	John	Julia	Julia

R_{13}

A	C	C
CS	Dave	Dave
CS	Xin	Xin
ENG	Cara	Cara
ENG	Julia	Julia

Figure 3.4: Query results $R_{11} = (Q_{29})^{\mathbb{D}_5}$, $R_{12} = (Q_{31})^{\mathbb{D}_5}$, and $R_{13} = (Q_{30})^{\mathbb{D}_5} = (Q_{32})^{\mathbb{D}_5}$.

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 $A \twoheadrightarrow C$.

Consider signature sb. The dependency $A \twoheadrightarrow C$ 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 $A \twoheadrightarrow C$ implies that B is not a core index.

The next theorem establishes that $\overline{\S}$-normalization does not change the complex object obtained after decoding the CEQ result with signature $\overline{\S}$. 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 $\mathbb{D}_5$ 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_{13} and R_{11} yield the object $\{\{|\,\mathsf{Dave}, \mathsf{Xin}\,|\}, \{|\,\mathsf{Cara}, \mathsf{Julia}\,|\}\}$ 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 $\{\!\{\{\!\{\mathsf{Dave}, \mathsf{Xin}\}\!\}, \{\!\{\mathsf{Cara}, \mathsf{Julia}\}\!\}\}\!\}$ when decoded with signature nn.

Theorem 3.1.17 *$\overline{\S}$-Normalization preserves $\overline{\S}$-equivalence.*

PROOF. Let $Q(\overline{\mathcal{I}}_1;\ldots;\overline{\mathcal{I}}_d;\overline{\mathcal{V}})$ CEQ, and let $Q'(\overline{\mathcal{I}}_1^{\overline{\S}};\ldots;\overline{\mathcal{I}}_d^{\overline{\S}};\overline{\mathcal{V}})$ be its $\overline{\S}$-normal form. For every $i \in [0,d]$, let Q_i denote the following CEQ.

$$Q_i(\overline{\mathcal{I}}_1;\ldots;\overline{\mathcal{I}}_i;\overline{\mathcal{I}}_{i+1}^{\overline{\S}};\ldots;\overline{\mathcal{I}}_d^{\overline{\S}};\overline{\mathcal{V}}) \coloneq \mathtt{body}_Q \tag{3.4}$$

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

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

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

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

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

$$\mathbb{T}((Q_{i-1})^{\mathbb{D}}) = \{\gamma[\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_{[i,d]}^{\overline{\S}}\overline{\mathcal{V}}] \mid \gamma \in \Gamma\} \tag{3.6}$$

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

$$\mathbb{T}((Q_i)^{\mathbb{D}}) = \{\phi[\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}}\overline{\mathcal{J}}_i\overline{\mathcal{I}}_{[i+1,d]}^{\overline{\S}}\overline{\mathcal{V}}] \mid \phi \in \Phi\} \tag{3.8}$$

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

Case $\S_i = \mathtt{b}$**:**

By Definition 3.1.11, $\mathcal{J}_i \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{C}$. Because $\mathcal{I}_{[1,i-1]}$ functionally determines $\mathcal{J}_i$, removing $\mathcal{J}_i$ 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 Γ and Φ that proves $\Gamma \doteq_{\overline{\S}} \Phi$.

Case $\S_i = \mathtt{s}$**:**

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

Let $\{\overline{b}_1,\ldots,\overline{b}_k\}$ be the set of values in $\mathtt{adom}(\overline{\mathcal{I}}_i^{\overline{\S}},\Gamma[\overline{a}]) = \mathtt{adom}(\overline{\mathcal{I}}_i^{\overline{\S}},\Phi[\overline{a}])$, and let $\{\overline{b}_1\overline{c}_1^1,\ldots\overline{b}_1\overline{c}_1^{l_1},\ldots\overline{b}_k\overline{c}_k^1,\ldots\overline{b}_k\overline{c}_k^{l_k}\}$ be the set of values in $\mathtt{adom}(\overline{\mathcal{I}}_i^{\overline{\S}}\overline{\mathcal{J}}_i,\Phi[\overline{a}])$. For each $\overline{b}_j$, the sub-relation $\Gamma[\overline{a}\overline{b}_j]$ encodes an object of sort $(\S_{[i+1,d]},d-i)$ occurring in the set at level i. By Definition 3.1.11, relation Φ satisfies $\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^{\overline{\S}} \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$, which—by Corollaries 3.1.5 and 3.1.6—implies that

$\Phi[\overline{a}]$ satisfies $\mathcal{I}_i^{\overline{\S}} \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$ (and $\mathcal{I}_i^{\overline{\S}} \twoheadrightarrow \mathcal{J}_i$). It follows from that all of the sub-relations $\Phi[\overline{a}\overline{b}_j\overline{c}_j^1], \ldots \Phi[\overline{a}\overline{b}_j\overline{c}_j^{l_j}]$ are identical to each other and to the sub-relation $\Gamma[\overline{a}\overline{b}_j]$. For each $\overline{c}_j^{l_m}$ add to $C_{\overline{a}}$ the mapping $f(\overline{b}_j\overline{c}_j^{l_m}) := \overline{b}_j$, as well as a child $\overline{\S}_{[i+1,d]}$-certificate proving that $\Phi[\overline{a}\overline{b}_j\overline{c}_j^{l_m}] \doteq_{\overline{\S}_{[i+1,d]}} \Gamma[\overline{a}\overline{b}_j]$ (which is trivial, because relations $\Phi[\overline{a}\overline{b}_j\overline{c}_j^{l_m}] = \Gamma[\overline{a}\overline{b}_j]$). Then, add the mapping $f'(\overline{b}_j) := \overline{b}_j \cdot \overline{c}_j^1$ (we could choose any $\overline{c}_j^{l_m}$). Certificate $C_{\overline{a}}$ is complete when this has been performed for all values of $\overline{b}_j$.

Case $\S_i = \mathtt{n}$**:**

The proof is almost identical to case $\S_i = \mathtt{s}$. Because $\mathcal{I}_{[1,i-1]} \twoheadrightarrow \mathcal{I}_{[i,d]}^{\overline{\S}}$ implies $\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^{\overline{\S}} \twoheadrightarrow \mathcal{I}_{[i+1,d]}^{\overline{\S}}$, all of the relations $\Phi[\overline{a}\overline{b}_j\overline{c}_j^1], \ldots \Phi[\overline{a}\overline{b}_j\overline{c}_j^{l_j}]$ are identical to each other and to the sub-relation $\Gamma[\overline{a}\overline{b}_j]$. Furthermore, $\mathcal{I}_{[1,i-1]} \twoheadrightarrow \mathcal{I}_{[i,d]}^{\overline{\S}}$ also implies that across all values of j, the sets $\{\overline{c}_j^1, \ldots \overline{c}_j^{l_j}\}$ are identical. Therefore, the degree of replication remains constant across all values of $\overline{b}_j \in \mathtt{adom}(\mathcal{I}_i^{\overline{\S}}, \Gamma[\overline{a}])$.

□

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* $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$ *and* $Q'(\overline{\mathcal{I}}'_1; \ldots; \overline{\mathcal{I}}'_d; \overline{\mathcal{V}}')$ *of equal depth, an* index-covering homomorphism *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. $h(\mathtt{body}_{Q'}) \subseteq \mathtt{body}_Q$,
2. $h(\overline{\mathcal{V}}') = \overline{\mathcal{V}}$, *and*
3. $\forall i \in [1, d]$: $\mathcal{I}_i \subseteq h(\mathcal{I}'_i)$.

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 $h : Q'(\overline{\mathcal{V}}') \to Q(\overline{\mathcal{V}})$, where $Q'(\overline{\mathcal{V}}')$ and $Q(\overline{\mathcal{V}})$ denote the CQs formed from Q' and Q by restricting the head tuples to $\overline{\mathcal{V}}'$ and $\overline{\mathcal{V}}$, respectively. Conversely, if h' is a CQ homomorphism from CQ $Q'''(\overline{\mathcal{W}}''')$ to CQ $Q''(\overline{\mathcal{W}}'')$, then h' is also an index-covering homomorphism from the depth-one CEQ $Q'''(\overline{\mathcal{W}}'''; \overline{\mathcal{W}}''')$ to the depth-one CEQ $Q''(\overline{\mathcal{W}}''; \overline{\mathcal{W}}'')$. For this reason, we use the same notation $h : Q' \to Q$ 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 $\overline{\S}$-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 $\overline{\S}$-equivalent iff there exists index-covering homomorphisms in both directions between their $\overline{\S}$-normal forms.*

Proof (sketch). Assume without loss of generality that Q and Q' are already in $\overline{\S}$-normal form (justified by Theorem 3.1.17).

If index-covering homomorphisms exist in both directions, then for any database $\mathbb{D}$ the encoding relations $(Q)^{\mathbb{D}}$ and $(Q')^{\mathbb{D}}$ differ at most by permutation of attributes within each index level. A $\overline{\S}$-certificate proving $(Q)^{\mathbb{D}} \doteq_{\overline{\S}} (Q')^{\mathbb{D}}$ is therefore straightforward to construct, since each node is simply an isomorphism between sub-relations; $Q \doteq\!\!\equiv_{\overline{\S}} Q'$ 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" $\overline{\S}$-certificate nodes that imply certain regularities between the encoding relations that they relate. The overall goal is then to construct a canonical database $\mathbb{D}$ out of the bodies of both Q and Q' such that if we can find a $\overline{\S}$-certificate between $(Q)^{\mathbb{D}}$ and $(Q')^{\mathbb{D}}$ 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* $\overline{\S}$-certificate between $(Q)^{\mathbb{D}}$ and $(Q')^{\mathbb{D}}$ 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 $\mathbb{D}$, 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 $\overline{\S}$-equivalence of CEQs is NP-complete.*

Corollary 3.2.6 *Testing equivalence of $\mathit{COCQL}^{\{f_b,f_s,f_n\}-}$ 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 $h : Q_{30} \to Q_{32}$ and $h' : Q_{32} \to Q_{30}$ prove $Q_{30} \doteq_{\mathtt{sb}} Q_{32}$ and $Q_{30} \doteq_{\mathtt{nn}} Q_{32}$.

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

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

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 $\overline{\S}$-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 $\texttt{COCQL}^{\mathcal{F}}$ 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 Π_2^p-hard [19].[4] 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].

[4] Π_2^p is a class within the second tier of the polynomial hierarchy. A decision problem is in Π_2^p if its complement problem is in Σ_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 model the input to certain normalizing aggregation functions such as `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 `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 $Q(\overline{\mathcal{V}})$ and $Q'(\overline{\mathcal{V}}')$, testing $Q \equiv Q'$ under

- *set semantics* reduces to $Q(\overline{\mathcal{V}};\overline{\mathcal{V}}) \doteq_{\mathtt{s}} Q'(\overline{\mathcal{V}}';\overline{\mathcal{V}}')$;
- *bag-set semantics* reduces to $Q(\mathcal{B};\overline{\mathcal{V}}) \doteq_{\mathtt{b}} Q'(\mathcal{B}';\overline{\mathcal{V}}')$ where $\mathcal{B}$ and $\mathcal{B}'$ are the query body variables;
- *bag-set semantics modulo a product* reduces to $Q(\mathcal{B};\overline{\mathcal{V}}) \doteq_{\mathtt{n}} Q'(\mathcal{B}';\overline{\mathcal{V}}')$; and
- *combined semantics* reduces to $Q(\mathcal{V} \cup \mathcal{M};\overline{\mathcal{V}}) \doteq_{\mathtt{b}} Q'(\mathcal{V}' \cup \mathcal{M}';\overline{\mathcal{V}}')$ where $\mathcal{M}$ and $\mathcal{M}'$ 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 `SUM`, `COUNT`, `COUNT-DISTINCT`, `MAX`, or `MIN` [25]. When the query bodies do not contain comparison predicates, the equivalence problem for SUM and `COUNT` queries reduces to equivalence under bag semantics, while the equivalence problem for `MAX`, `MIN`, and `COUNT-DISTINCT` reduces to equivalence under set semantics. When the query bodies contain comparison predicates, they show the problem to be Π_2^p-complete for `MAX` queries, `MIN` queries, and `COUNT-DISTINCT` queries under a slight syntactic restriction that limits interaction between the aggregated variables and the variables involved in comparison predicates. For `SUM` and `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—*shiftable*, *order-decidable*, *decomposable*, *singleton-determining*, and *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 `AVG` or `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 `SUM`, `COUNT`, and `PRODUCT` aggregation functions.

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

$$\Pi_{\overline{x}}^{z=f(y)}(E) \tag{3.9}$$

where E is a relational algebra query not containing aggregation functions. Observe that when generalized projection only occurs as the top-most query operator, *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.

4.1.1 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 $R \vec{\bowtie}_p S$—is equivalent to the *outer union* of the inner- and anti-joins of R and S,

$$R \vec{\bowtie}_p S \equiv (R \bowtie_p S) \uplus (R \rhd_p S) \equiv (R \bowtie_p S) \uplus (R - (R \ltimes_p S)) \tag{4.1}$$

where the outer union operator $\uplus$ pads the tuples originating from the anti-join with a special null constant[1] - 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_1 subsumes t_2 if t_2 contains strictly more - values than t_1, and t_1 and t_2 coincide on all non-- 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 $R \vec{\bowtie}_p S$ 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].

$$R \vec{\bowtie}_p S \equiv ((R \bowtie_p S) \uplus R)\downarrow \quad \equiv (R \bowtie_p S) \oplus R \tag{4.2}$$

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 - 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 - within the query output are introduced by LOJ operators.

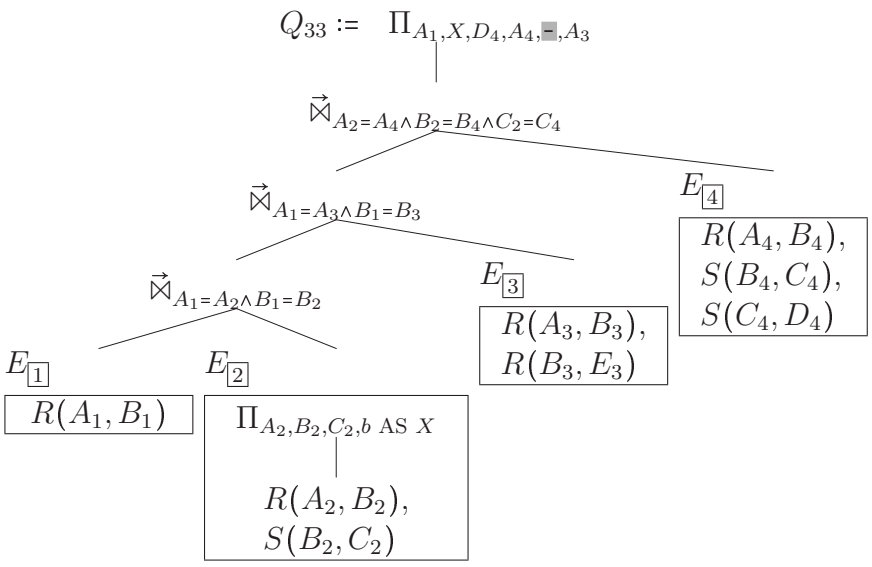


Figure 4.1: SPCL query Q_{33} equivalent to HCQ Q_{17}

that constants are assigned fresh attribute names; for this we adopt the common "c AS A_{i_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, $E_{\boxed{1}}$ corresponds to the tree T_6 in Figure 2.12; similarly, $E_{\boxed{2}}$ corresponds to T_4, $E_{\boxed{3}}$ to T_5, and $E_{\boxed{4}}$ to T_2. The reader can verify that when Q_{33} is evaluated over database instance $\mathbb{D}_2$ 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 $((\Gamma^6 \vec{\bowtie} \Gamma^4) \vec{\bowtie} \Gamma^5) \vec{\bowtie} \Gamma^2$.

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 body_Q, create an SPC expression $E_{\boxed{v}}$ 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 $E_{\boxed{v}}$ 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 $E_{\boxed{1}}$, $E_{\boxed{2}}$, $E_{\boxed{3}}$, and $E_{\boxed{4}}$ represent CQ bodies that are easily translated into pure SPC expressions.

R_{14}

A_1	X	D_4	A_4	-	A_3
1	b	5	1	-	1
2	-	-	-	-	2
2	b	6	2	-	-
3	b	6	3	-	-
3	b	-	-	-	3
5	b	-	-	-	-
6	-	-	-	-	-

R_{15}

A_1	X	D_4	A_4	-	A_3
1	b	-	-	-	1
1	b	5	1	-	1
2	-	-	-	-	2
2	b	-	-	-	-
3	b	-	-	-	-
3	b	-	-	-	3
5	b	-	-	-	-
6	-	-	-	-	-

Figure 4.2: SPCL query results $R_{14} = (Q_{33})^{\mathbb{D}_2}$ and $R_{15} = (Q_{34})^{\mathbb{D}_2}$

nodes of $\texttt{body}_Q$. For each non-root node v whose parent node in $\texttt{body}_Q$ is u, the LOJ operator immediately above $E_{\boxed{v}}$ has a join predicate that equates all of the attributes from $E_{\boxed{u}}$ with their corresponding attributes in $E_{\boxed{v}}$.

3. Create a top-most projection operator, and for each component W_i labelled with v in the head $Q(\overline{\mathcal{W}})$, output the associated attribute from $E_{\boxed{v}}$.

Consider any tree $T_i \in \texttt{trees}(\texttt{body}_Q)$, and let $u_1, \ldots, u_k$ denote the nodes of T_i in breadth-first order. Given any database instance $\mathbb{D}$, there is a one-to-one correspondence between embeddings of T_i into $\mathbb{D}$ and tuples in the *inner* join $(E_{\boxed{u_1}} \bowtie \ldots \bowtie E_{\boxed{u_k}})^{\mathbb{D}}$. An embedding $\gamma : T_i \to \mathbb{D}$ is terminal precisely when the corresponding tuple in $(E_{\boxed{u_1}} \bowtie \ldots \bowtie E_{\boxed{u_k}})^{\mathbb{D}}$ 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 $E_{\boxed{4}}$. Evaluating Q_{34} over database instance $\mathbb{D}_2$ 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 $\mathcal{P}_v \cap \mathcal{L}_u \neq \emptyset$; 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.

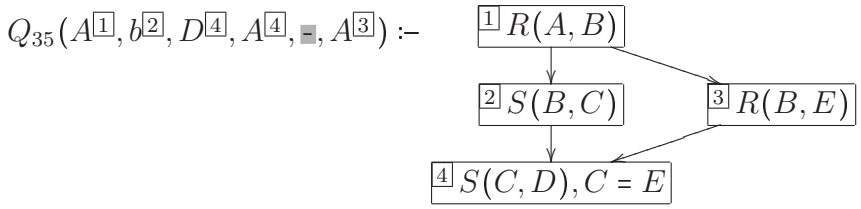


Figure 4.3: Directed acyclic conjunctive query Q_{35}

Γ	A	B	C	D	E	b	$A^{[1]}$	$b^{[2]}$	$D^{[4]}$	$A^{[4]}$	-	$A^{[3]}$
	1	2	4		3	b	1	b	-	-	-	1
	1	2	4	5	4	b	1	b	5	1	-	1
	2	3			4	b	2	-	-	-	-	2
	2	3			5	b	2	-	-	-	-	2
	2	4	5			b	2	b	-	-	-	-
	3	4	5			b	3	b	-	-	-	-
	3	5	6		7	b	3	b	-	-	-	3
	5	7	8			b	5	b	-	-	-	-
	6	8				b	6	-	-	-	-	-

Figure 4.4: Terminal embeddings of Q_{35} into $\mathbb{D}_2$

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 $\texttt{dags}(G)$ 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 $\gamma : G \to \mathbb{D}$ and terminal embeddings $\gamma : G \rightsquigarrow \mathbb{D}$ for $G \in \texttt{dags}(\texttt{body}_Q)$. The CQ $\texttt{body}^+_G$ is defined analogously to $\texttt{body}^+_T$.

Example 34 Figure 4.3 depicts DACQ Q_{35}, and Figure 4.4 shows the terminal embeddings of Q_{35} into database instance $\mathbb{D}_2$.

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 $\texttt{body}_{Q_{36}}$ and the SPC blocks within Q_{34}.

Theorem 4.1.3 *DACQs and SPCL expressions have the same expressive power.*

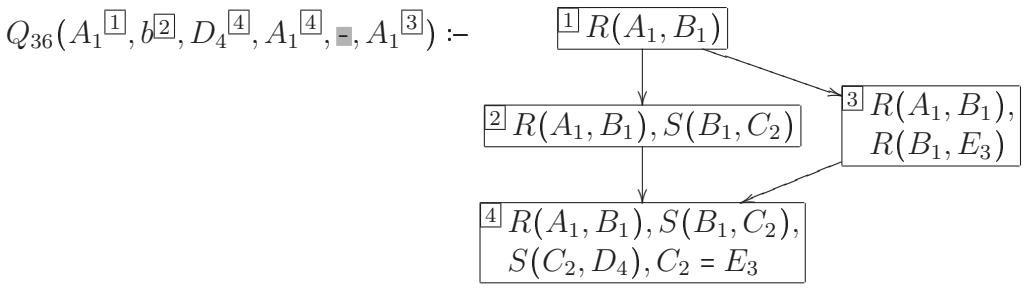


Figure 4.5: DACQ Q_{36} equivalent to SPCL query Q_{34}

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 $E_{\boxed{u_1}}, \ldots, E_{\boxed{u_k}}$ 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 $\boxed{u_1}$ containing all of the base relations and join/selection predicates occurring within SPC subquery $E_{\boxed{u_1}}$.
2. Enumerate SPC queries $E_{\boxed{u_2}}, \ldots, E_{\boxed{u_k}}$. For each query $E_{\boxed{u_i}}$, let p_i denote the join predicate of the LOJ operator that has $E_{\boxed{u_i}}$ as its right input.
   (a) Create a node labelled $\boxed{u_i}$ containing all of the base relations and join/selection predicates occurring within SPC subquery $E_{\boxed{u_i}}$, as well as the predicates in p_i.
   (b) For each $j \neq i$ such that p_i references an attribute originating from $E_{\boxed{u_j}}$, 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.

□

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 $u \in \mathtt{body}_Q$, CQ body $\mathtt{body}_u^+$ is satisfiable.*

[P2] *For each non-root leaf vertex $u \in \mathtt{body}_Q$, the head tuple $\overline{\mathcal{W}}$ of Q contains a component W_i that is labelled with vertex u.*

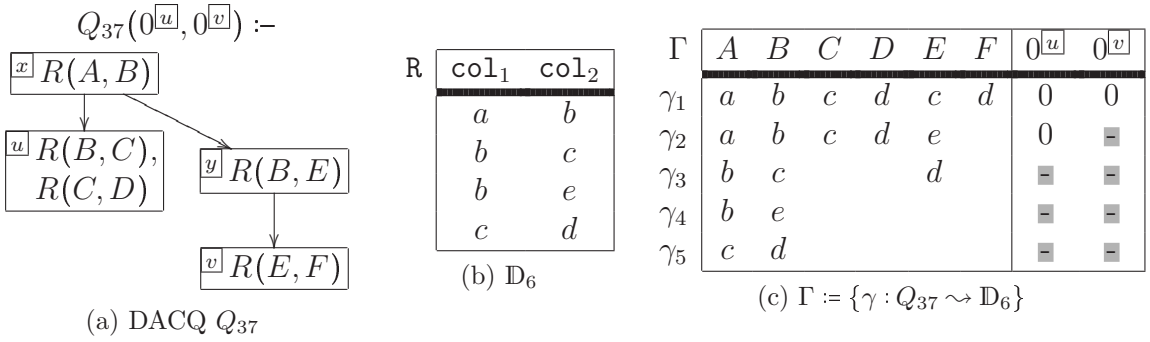
[P3] *For each vertex $u \in \mathtt{body}_Q$ and each parameter variable $X \in \mathcal{P}_u$:*
- *(a) if $\mathtt{pred}_u^+$ implies that X is equal to a constant, then X does not occur within $\mathtt{atoms}_u$; and*
- *(b) if $\mathtt{pred}_u^+$ implies that X is equal to some other parameter $Y \in \mathcal{P}_u$, then $\mathtt{atoms}_u$ does not contain both X and Y (it may contain one).*

(This presumes that predicates in $\mathtt{pred}_u$ 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 $\mathtt{atoms}_u$ is redundant if $\mathtt{pred}_u^{\mathtt{anc}}$ implies that it is identical to an atom within $\mathtt{atoms}_u^{\mathtt{anc}}$. A predicate within $\mathtt{pred}_u$ is redundant if it is implied by $\mathtt{pred}_u^{\mathtt{anc}}$. 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 $u, v \in \mathtt{body}_Q$, if v is* not *an ancestor of u, then there exists some* terminal *embedding of Q into some database $\mathbb{D}$ that embeds $\mathtt{body}_u$ into $\mathbb{D}$ but not $\mathtt{body}_v$ into $\mathbb{D}$.*

The purpose of properties **[P1]** and **[P2]** is to avoid having vertices in $\mathtt{body}_Q$ 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 $\mathtt{body}_Q$—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.



(a) DACQ Q_{37} (b) $\mathbb{D}_6$ (c) $\Gamma \coloneqq \{\gamma : Q_{37} \rightsquigarrow \mathbb{D}_6\}$

Figure 4.6: Query Q_{37} and its terminal embeddings into $\mathbb{D}_6$

Given a query Q, properties [**P1**]–[**P4**] can all be tested in polynomial time by examining the syntax of Q and using transitivity to reason about implication of equality predicates. However, it is not obvious how to test property [**P5**]. On first glance, it may appear that this is simply a CQ containment problem—specifically, that $\mathtt{body}_u^+$ is not contained under $\mathtt{body}_v^+$. However, the following example demonstrates that the problem is more nuanced than that.

Example 36 Consider the DACQ Q_{37} shown in Figure 4.6a. The variable mapping $h \coloneqq \{A/A, B/B, E/C, F/D\}$ is a homomorphism $h : \mathtt{body}_v^+ \to \mathtt{body}_u^+$ proving that any embedding of vertex x that extends to vertex u will also extend to vertex v. Nevertheless, database instance $\mathbb{D}_6$ in Figure 4.6b proves that we cannot safely add an edge from v to u. Doing so would eliminate the terminal embedding γ_2 shown in Figure 4.6c, thereby removing the output tuple $\langle\, 0, \text{-} \,\rangle$ from $(Q_{37})^{\mathbb{D}_6}$.

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 $v \in \mathtt{body}_Q$, and a set of variables $\mathcal{S} \subseteq \mathcal{B}_v^{\mathtt{anc}}$, we say that $\mathcal{S}$* null-determines *v within the embeddings of Q—denoted $Q \vDash \mathcal{S} \twoheadrightarrow v$—if for every database instance $\mathbb{D}$ and every pair of embeddings $\gamma, \phi : \mathtt{body}_v^{\mathtt{anc}} \to \mathbb{D}$ that coincide over $\mathcal{S}$, embedding γ extends to an embedding of v iff embedding ϕ 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 $\mathcal{S}$ to contain constants.)*

Example 37 Continuing Example 36, terminal embeddings γ_1 and γ_2 in Figure 4.6c prove that $Q_{37} \nvDash \{A, B\} \twoheadrightarrow v$. However, $Q_{37} \vDash \{E\} \twoheadrightarrow v$ holds (by the next lemma).

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

PROOF. For any embedding $\gamma : \mathtt{body}_v^{\mathtt{anc}} \to \mathbb{D}$, whether or not γ extends to v is determined only by the tuple of parameter values $\gamma(\overline{\mathcal{P}_v})$. □

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*

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

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

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

PROOF. Suppose that h exists. Choose any database instance $\mathbb{D}$, and any two embeddings $\gamma, \phi : \mathtt{body}_u \to \mathbb{D}$ that coincide over $\mathcal{S}$ and for which there exists an embedding $\phi' : \mathtt{body}_v^+ \to \mathbb{D}$ that extends ϕ. Define the variable mapping $\gamma' : \mathcal{B}_v^+ \to \mathbb{D}$ 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 $\mathtt{atoms}_u$, γ' is consistent with γ over $\mathtt{body}_u$. For each atom $R(\overline{X}) \in \mathtt{atoms}_v$, if $h(R(\overline{X})) \in \mathtt{atoms}_u$ then $\gamma'(R(\overline{X})) = \gamma \circ h(R(\overline{X})) \in \mathbb{D}$. If $h(R(\overline{X})) \in \mathtt{atoms}_v$, then because $h(\overline{X})$ does not contain any variables from $\mathcal{B}_u \setminus \mathcal{S}$ and because γ, ϕ are consistent over $\mathcal{S}$, $\gamma'(R(\overline{X})) = \phi'(R(\overline{X})) \in \mathbb{D}$. Therefore γ' is an embedding of $\mathtt{body}_v^+$ into $\mathbb{D}$ that extends γ, which implies $Q \vDash \mathcal{S} \twoheadrightarrow v$.

Now suppose $Q \vDash \mathcal{S} \twoheadrightarrow v$. 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 $\mathcal{S}$ and satisfy $f_1(\mathcal{B}_u \setminus \mathcal{S}) \cap f_2(\mathcal{B}_v^+ \setminus \mathcal{S}) = \varnothing$. Define database instances $\mathbb{D}_1 := f_1(\mathtt{atoms}_u)$, $\mathbb{D}_2 := f_2(\mathtt{atoms}_v^+)$, and $\mathbb{D} := \mathbb{D}_1 \cup \mathbb{D}_2$. 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 $f^{-1}(\mathbb{D}) = \mathtt{atoms}_v^+$. Choose embedding $\gamma : \mathtt{body}_u \to \mathbb{D}$ as $\gamma := f_1$. Choose embedding $\phi : \mathtt{body}_v^+ \to \mathbb{D}$ as $\phi := f_2$. Because $Q \vDash \mathcal{S} \twoheadrightarrow v$ and γ, ϕ are consistent over $\mathcal{S}$, there must exist some embedding $\gamma' : \mathtt{body}_v^+ \to \mathbb{D}$ that extends γ.

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 $\mathtt{body}_v^+$ to $\mathtt{body}_v^+$ that is the identity over $\mathcal{B}_u$ (and therefore over $\mathtt{body}_u$). 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. $\gamma'(X) \in f_1(\mathcal{S}) = f_2(\mathcal{S})$ and so $h_1(X) \in \mathcal{S}$;
3. $\gamma'(X) \in f_1(\mathcal{B}_u \setminus \mathcal{S})$ and so $h_1(X) \in \mathcal{B}_u \setminus \mathcal{S}$; or
4. $\gamma'(X) \in f_2(\mathcal{B}_v^+ \setminus \mathcal{S})$ and so $h_1(X) \in \mathcal{B}_v^+ \setminus \mathcal{S}$.

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 $h_i(X) \in \mathcal{C}_Q \cup \mathcal{B}_u$, which implies that $\forall j \geq i.[h_j(X) = h_i(X)]$, and so $\gamma' \circ h_k(X) \in f_1(\mathcal{C}_Q \cup \mathcal{B}_u)$; 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 $R(\overline{X}) \in \mathtt{atoms}_v$ such that $h_{k+1}(R(\overline{X})) \in \mathtt{atoms}_v$. By property **[P4]**, $h_{k+1}(R(\overline{X})) \notin \mathtt{atoms}_u$ and therefore $\gamma' \circ h_k(R(\overline{X})) \notin \mathbb{D}_1$. Suppose that there exists some variable $Y \in \overline{X}$ such that $h_{k+1}(Y) \in \mathcal{B}_u \setminus \mathcal{S}$; then, $h_{k+1}(Y)$ must belong to the first case above, implying that $h_k(Y) = h_{k+1}(Y)$ and $\gamma' \circ h_k(Y) = f_1(h_k(Y)) \notin \mathtt{adom}(\mathbb{D}_2)$, which is a contradiction. Therefore, variable Y cannot exist, and so tuple $h_{k+1}(\overline{X})$ does not contain variables from $\mathcal{B}_u \setminus \mathcal{S}$. □

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

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

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

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

Suppose that $Q \models \mathcal{S}_1 \twoheadrightarrow v$, and let h_1 be the corresponding homomorphism that satisfies Theorem 4.2.4. Suppose that $Q \models \mathcal{S}_2 \twoheadrightarrow v$, with h_2 the corresponding homomorphism. Consider the composite function $h_2 \circ h_1$. For any atom $R(\overline{X}) \in \mathtt{atoms}_v$, if $h_1(R(\overline{X})) \in \mathtt{atoms}_u$ then $h_2 \circ h_1(R(\overline{X})) \in \mathtt{atoms}_u$, while if $h_1(R(\overline{X})) \in \mathtt{atoms}_v$ and $h_1(\overline{X})$ contains a variable from $\mathcal{B}_u \setminus \mathcal{S}_2$ then $h_2 \circ h_1(R(\overline{X})) \in \mathtt{atoms}_u$ as well (because h_2 is the identity over $\mathcal{B}_u$ and any atom that h_2 maps into $\mathtt{atoms}_v$ can not contain $\mathcal{B}_u \setminus \mathcal{S}_2$). 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 $|\mathtt{atoms}_v|$ alternations of h_1 and h_2; h is the identity over $\mathtt{atoms}_u$, and for any atom $R(\overline{X}) \in \mathtt{atoms}_v$ that h maps into $\mathtt{atoms}_v$, tuple $h(\overline{X})$ does not contain any variable from $\mathcal{B}_u \setminus (\mathcal{S}_1 \cap \mathcal{S}_2)$. Therefore, h proves $Q \models (\mathcal{S}_1 \cap \mathcal{S}_2) \twoheadrightarrow v$. □

Example 38 Consider the following two DACQs.

$$Q_{38}(A^{\boxed{b}}) \text{ :- } \boxed{a}\, R(A,B), R(X,Y) \longrightarrow \boxed{b}\, R(A,A), R(X,Z)$$

$$Q_{39}(A^{\boxed{d}}) \text{ :- } \boxed{c}\, R(A,B), R(X,Y) \longrightarrow \boxed{d}\, R(A,Z), R(X,Z)$$

The homomorphism $h_b = \{A/A, B/B, X/X, Y/Y, Z/Y\}$ proves $Q_{38} \models \{A\} \twoheadrightarrow b$, and $\mathcal{S}_b = \{A\}$ is minimal because no homomorphism from $\texttt{body}_b^+$ to itself can map atom $R(A, A)$ into $\texttt{body}_a$. For Q_{39}, the only homomorphism from $\texttt{body}_d^+$ to itself that is the identity over $\texttt{body}_c$ is $h_d = \{A/A, B/B, X/X, Y/Y, Z/Z\}$; therefore, $\mathcal{S}_d = \{A, X\} = \mathcal{P}_d$ is the minimal set satisfying $Q_{39} \models \mathcal{S}_d \twoheadrightarrow d$.

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 $\mathcal{S} \subseteq \mathcal{B}_u$;*

assume a total ordering over $\mathcal{B} \cup \mathcal{C}$ that places all constants before any variables. Let $\kappa : \mathcal{B} \to \mathcal{B} \cup \mathcal{C}$ be the mapping of each variable to the minimal element $Y \in \mathcal{B} \cup \mathcal{C}$ such that $\texttt{pred}_v^+ \models X = Y$. Then, $Q \models \mathcal{S} \twoheadrightarrow v$ iff

1. *for each variable X in $\texttt{pred}_v$ there exists a variable $Y \in \mathcal{S}$ such that $\texttt{pred}_u \models X = Y$; and*
2. *$\kappa(Q) \models \kappa(\mathcal{S}) \twoheadrightarrow v$.*

PROOF. Suppose that $Q \models \mathcal{S} \twoheadrightarrow v$.

1. Suppose that $\texttt{pred}_v$ contains some predicate $X = Z$ such that there does not exist a variable $Y \in \mathcal{S}$ satisfying $\texttt{pred}_u \models X = Y$. Let γ be any embedding of $\texttt{body}_v^+$ into any database $\mathbb{D}$. Let c denote some fresh constant not in $\texttt{adom}(D)$. Initialize variable mapping ϕ as identical to γ; then, for each variable Y such that $\texttt{pred}_u \models X = Y$, set $\phi(Y) := c$. Define database $\mathbb{D}' := \mathbb{D} \cup \phi(\texttt{atoms}_u)$. Both γ and ϕ embed $\texttt{body}_u$ into $\mathbb{D}'$ and they coincide on $\mathcal{S}$, yet γ extends to v while ϕ does not because $\phi(X) = c \neq \phi(Z)$ (property **[P4]** requires that $\texttt{pred}_u \not\models X = Z$). Hence, $Q \not\models \mathcal{S} \twoheadrightarrow v$, which is a contradiction, and so $\texttt{pred}_v$ cannot contain predicate $X = Z$.
2. By construction, mapping κ enacts the equality predicates within $\texttt{pred}_v^+$; therefore, each embedding of $\kappa(\texttt{body}_u)$ is also an embedding of $\texttt{body}_u$, and each embedding of $\kappa(\texttt{body}_v^+)$ is also an embedding of $\texttt{body}_v^+$. Therefore, any two embeddings of $\kappa(\texttt{body}_u)$ proving $\kappa(Q) \not\models \kappa(\mathcal{S}) \twoheadrightarrow v$ would also prove $Q \not\models \mathcal{S} \twoheadrightarrow v$, which is a contradiction. Therefore, $\kappa(Q) \models \kappa(\mathcal{S}) \twoheadrightarrow v$.

Now suppose that for each variable X in $\texttt{pred}_v$ there exists a variable $Y \in \mathcal{S}$ satisfying $\texttt{pred}_u \models X = Y$, and suppose that $\kappa(Q) \models \kappa(\mathcal{S}) \twoheadrightarrow v$. Given any database $\mathbb{D}$ and any two embeddings $\gamma, \phi : \texttt{body}_u \to \mathbb{D}$ that coincide over $\mathcal{S}$, embeddings γ, ϕ are also embeddings of $\kappa(\texttt{body}_u) \to \mathbb{D}$ that coincide over $\kappa(\mathcal{S})$. Furthermore, γ and ϕ coincide over $\mathcal{P}_v$, and so γ satisfies $\texttt{pred}_v$ iff ϕ satisfies $\texttt{pred}_v$. Suppose that γ' extends γ to an embedding of $\texttt{body}_v^+$; then γ' is also an embedding of $\kappa(\texttt{body}_v^+)$. By $\kappa(Q) \models \kappa(\mathcal{S}) \twoheadrightarrow v$, there exists an embedding ϕ' that extends ϕ to $\kappa(\texttt{body}_v^+)$, and ϕ' is also an embedding of $\texttt{body}_v^+$. Therefore, $Q \models \mathcal{S} \twoheadrightarrow v$. □

Each predicate in $\kappa(\texttt{pred}_Q)$ is a tautology and can be deleted. Hence, $\kappa(Q)$ does not contain equality predicates and so $\kappa(Q) \models \kappa(\mathcal{S}) \twoheadrightarrow v$ 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 $v \in \texttt{body}_Q$, and some set $\mathcal{S} \subseteq \mathcal{B}_v^{\texttt{anc}}$, create query Q' from Q by first deleting all nodes not in $\{v\} \cup \texttt{ancestors}(v)$ and then merging together all of the nodes in $\texttt{ancestors}(v)$. It follows from Definition 4.2.2 that $Q \models \mathcal{S} \twoheadrightarrow v$ if and only if $Q' \models \mathcal{S} \twoheadrightarrow v$. 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 $v \in \texttt{body}_Q$ and every set $\mathcal{S} \subseteq \mathcal{B}_v^{\texttt{anc}}$, if $Q \models \mathcal{S} \twoheadrightarrow v$ then $Q \models (\mathcal{S}_Q \cap \mathcal{S}) \twoheadrightarrow v$. Furthermore, $\mathcal{S}_Q$ contains all variables occurring within $\texttt{pred}_Q$.*

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 $u, v \in \texttt{body}_Q$ 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 ϑ be a mapping from the vertex labels of Q to the vertex labels of Q_u. Then, the following property holds

$$\forall \mathbb{D}. \forall G \in \texttt{dags}(Q). \left[(u \in G \wedge v \notin G) \Longrightarrow (\{\gamma : G \rightsquigarrow \mathbb{D}\} = \varnothing) \right] \tag{4.3}$$

if and only if

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

Proof. Suppose that property 4.3 holds.

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

$$Q'(\overline{\mathcal{B}_u^+}) \coloneq \texttt{body}_u^+$$
$$Q''(\overline{\mathcal{B}_u^+}) \coloneq \texttt{body}_u^+ \cup \texttt{body}_v^+$$

   which, by the Homomorphism Theorem for CQ containment [6], implies that there exists a homomorphism $h' : Q'' \to Q'$ satisfying $h'(\overline{\mathcal{B}_u^+}) = \overline{\mathcal{B}_u^+}$. This mapping h' therefore satisfies the criteria for mapping h given above.
2. For any $\mathbb{D}$, let γ, ϕ be any two embeddings of Q_u that are consistent over $\mathcal{B}_u^+ \cap \mathcal{B}_v^+$. For any $t \in \texttt{ancestors}(v) \cup v$, property 4.3 guarantees that both γ and ϕ can be extended into embeddings of $\vartheta(t)$, and so $Q_u \vDash \mathcal{B}_u^+ \cap \mathcal{B}_v^+ \twoheadrightarrow \vartheta(t)$ 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 $t \in \texttt{ancestors}(v) \cup \{v\}$. Suppose further than property 4.3 is violated; that is, there exists some database $\mathbb{D}$ and $G \in \texttt{dags}(Q)$ with $u \in G$ and $v \notin G$ that allows some terminal embedding $\gamma : G \rightsquigarrow \mathbb{D}$. Choose any vertex $t \in \texttt{ancestors}(v) \cup \{v\}$ 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. γ is an embedding of $\texttt{body}^{\texttt{anc}}_{\vartheta(t)}$ into $\mathbb{D}$;
2. $\gamma \circ h$ is an embedding of $\texttt{body}^{\texttt{anc}}_{\vartheta(t)}$ into $\mathbb{D}$;
3. γ and $\gamma \circ h$ are consistent over $\mathcal{B}_u^+ \cap \mathcal{B}_v^+$ (by definition of h and $\mathcal{B}_u^+ \cap \mathcal{B}_v^+ \subseteq \mathcal{B}_t^{\texttt{anc}}$);
4. $\gamma \circ h$ extends to $\vartheta(v)$ and therefore also to $\vartheta(t)$; and
5. $Q_u \vDash \mathcal{B}_u^+ \cap \mathcal{B}_v^+ \twoheadrightarrow \vartheta(t)$ implies that γ must extend to $\vartheta(t)$ within $\texttt{body}_{Q_u}$, which further implies that γ extends to t within $\texttt{body}_Q$.

The above reasoning contradicts the choice of γ as a *terminal* embedding of G. Therefore, property 4.3 must hold. □

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 $u, v \in \texttt{body}_Q$ that cause the violation (i.e., that satisfy property 4.3), along with the homomorphism h from Theorem 4.2.4 and, for each $t \in \texttt{ancestors}(v) \cup \{v\}$ the homomorphisms needed to verify $Q_u \vDash \mathcal{B}_u^+ \cap \mathcal{B}_v^+ \twoheadrightarrow \vartheta(t)$ (cf. Theorems 4.2.4 and 4.2.6). □

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

1. *there exists a homomorphism $h : \texttt{body}_v^+ \to \texttt{body}_u^+$ such that h is the identity over $\mathcal{B}_u^+ \cap \mathcal{B}_v^+$, and*

Algorithm 4 Converting DACQs into canonical form

CANONICAL(Q)

▷ **Input:** satisfiable DACQ $Q(\overline{\mathcal{W}})$
▷ **Output:** an equivalent DACQ in canonical form
while $\exists u \in \mathtt{body}_Q$ such that $\mathtt{body}_u^+$ is unsatisfiable
  do delete vertex u and all its descendants from $\mathtt{body}_Q$
    substitute - for each component of $\overline{\mathcal{W}}$ labelled with a deleted vertex
while $\exists u \in \mathtt{body}_Q$ such that u is a leaf, u is not the root,
    and u does not occur as a label in $\overline{\mathcal{W}}$
  do delete vertex u from $\mathtt{body}_Q$
foreach $u \in \mathtt{body}_Q, X \in \mathcal{P}_u$
  do if $\mathtt{pred}_u^+$ equates X to $Y \in \mathcal{P}_u \cup \mathcal{C}$
    then replace any occurrence of X with Y within $\mathtt{atoms}_u$
      replace any occurrence of $X^{\boxed{u}}$ with $Y^{\boxed{u}}$ within $\overline{\mathcal{W}}$
delete redundant atoms, predicates, and edges from $\mathtt{body}_Q$
foreach pair $u, v \in \mathtt{body}_Q$ such that $u \neq v$ and $v \notin \mathtt{ancestors}(u)$
  do if TESTPROPERTY4.3(u, v) = `true`
    then add an edge from v to u
      while $\mathtt{body}_Q$ contains a cycle
        do merge all participating vertices
      delete redundant atoms and predicates from $\mathtt{body}_Q$
return Q

2. for all $t \in \mathtt{ancestors}(v) \cup \{v\}$, $\mathcal{P}_t \subseteq (\mathcal{B}_u^+ \cap \mathcal{B}_v^+)$.

PROOF. By Lemma 4.2.3, $\mathcal{P}_t \subseteq (\mathcal{B}_u^+ \cap \mathcal{B}_v^+)$ implies that $Q \models \mathcal{B}_u^+ \cap \mathcal{B}_v^+ \rightarrow t$. By construction of Q_u, this further implies that $Q_u \models \mathcal{B}_u^+ \cap \mathcal{B}_v^+ \rightarrow \vartheta(t)$. □

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).

$$\{(\boxed{1},\boxed{2}), (\boxed{1},\boxed{3}), (\boxed{1},\boxed{4}), (\boxed{2},\boxed{3}), (\boxed{2},\boxed{4}), (\boxed{3},\boxed{2}), (\boxed{3},\boxed{4})\}$$

For each pair, the reader can verify that no homomorphism exists from $\mathtt{body}_v^+$ to $\mathtt{body}_u^+$; 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 CANONICAL(Q) shown in Algorithm 4 transforms an arbitrary satisfiable DACQ into an equivalent query in canonical form. It relies upon invoking

the function 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 $\texttt{body}^+_u$ is unsatisfiable, neither u nor any of its descendants can participate in any embedding. Deleting u and its descendants from $\texttt{body}_Q$ 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 -. Hence, when the loop terminates the modified query remains equivalent and obeys property [**P1**].
- **Lines 4–5:** Let G be any DAG in $\texttt{dags}(\texttt{body}_Q)$ such that $u \in G$, and let $G' \in \texttt{dags}(\texttt{body}_Q)$ be the DAG formed by deleting u from G. Let $\gamma : G \rightsquigarrow \mathbb{D}$ be any terminal embedding of G into some database $\mathbb{D}$, and let $\gamma' : G' \to \mathbb{D}$ be the restriction of γ to G'. Deleting u from $\texttt{body}_Q$ deletes terminal embedding γ but causes γ' to become a terminal embedding; furthermore, the partial applications of γ and γ' 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 $G \in \texttt{dags}(\texttt{body}_Q)$ containing vertex u, any embedding γ 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 $G \in \texttt{dags}(\texttt{body}_Q)$ 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 $\texttt{dags}(\texttt{body}_Q)$, 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 testProperty4.3(u, v) evaluates to `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 $\texttt{body}_Q$ 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 $\texttt{body}_Q$, the query also satisfies property [**P5**].

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

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

1. $\texttt{body}^+_5$ is unsatisfiable, so delete nodes 5 and 8.

$$\texttt{body}^+_5 := R(A, B, C), R(A, D, 2), S(G, H), A = 2, A = 3$$

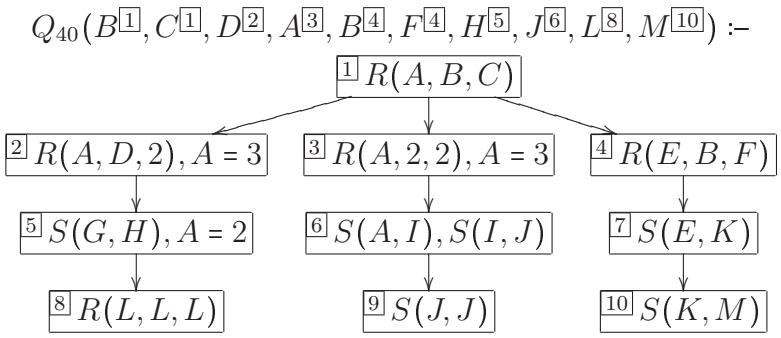


Figure 4.7: HCQ Q_{40} prior to conversion to canonical form

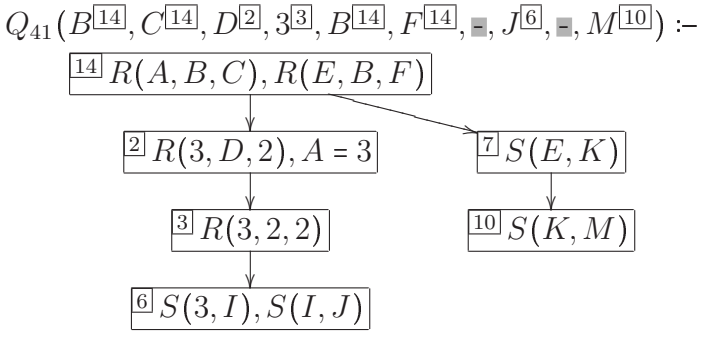


Figure 4.8: Query Q_{41} = Canonical(Q_{40})

2. Delete non-root leaf node $\boxed{9}$ because label $\boxed{9}$ does not occur in the query head.
3. Replace variable A with the constant 3 within the atoms of nodes $\boxed{2}$, $\boxed{3}$, and $\boxed{6}$, and replace $A^{\boxed{3}}$ with $3^{\boxed{3}}$ within the query head.
4. Add an edge from $\boxed{2}$ to $\boxed{3}$ because $\text{TESTPROPERTY}4.3(\boxed{3},\boxed{2})$ returns `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 $h : \texttt{body}_2^+ \to \texttt{body}_3^+$ is the identity over $\mathcal{B}_3^+ \cap \mathcal{B}_2^+$.

$$\texttt{body}_3^+ := R(A, B, C), R(A, 2, 2), A = 3$$
$$\texttt{body}_2^+ := R(A, B, C), R(A, D, 2), A = 3$$
$$h := \{A/A, B/B, C/C, D/2\}$$

   Next, $\texttt{ancestors}(\boxed{2}) \cup \{\boxed{2}\} = \{\boxed{1}, \boxed{2}\}$, and so $\mathcal{P}_1 = \varnothing$ 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 $\boxed{3}$ redundant, and makes the edge $(\boxed{1}, \boxed{3})$ redundant, so delete them.
5. Add an edge from $\boxed{4}$ to $\boxed{1}$ because $\text{TESTPROPERTY}4.3(\boxed{1},\boxed{4})$ returns `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 $h : \texttt{body}_4^+ \to \texttt{body}_1^+$ is the identity over $\mathcal{B}_1^+ \cap \mathcal{B}_4^+$.

$$\texttt{body}_1^+ := R(A, B, C)$$
$$\texttt{body}_4^+ := R(A, B, C), R(E, B, F)$$
$$h := \{A/A, B/B, C/C, E/A, F/C\}$$

   Next, $\texttt{ancestors}(\boxed{4}) \cup \{\boxed{4}\} = \{\boxed{1}, \boxed{4}\}$, and so $\mathcal{P}_1 = \varnothing$ 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 $\boxed{1}$ and $\boxed{4}$ into a single node $\boxed{14}$.

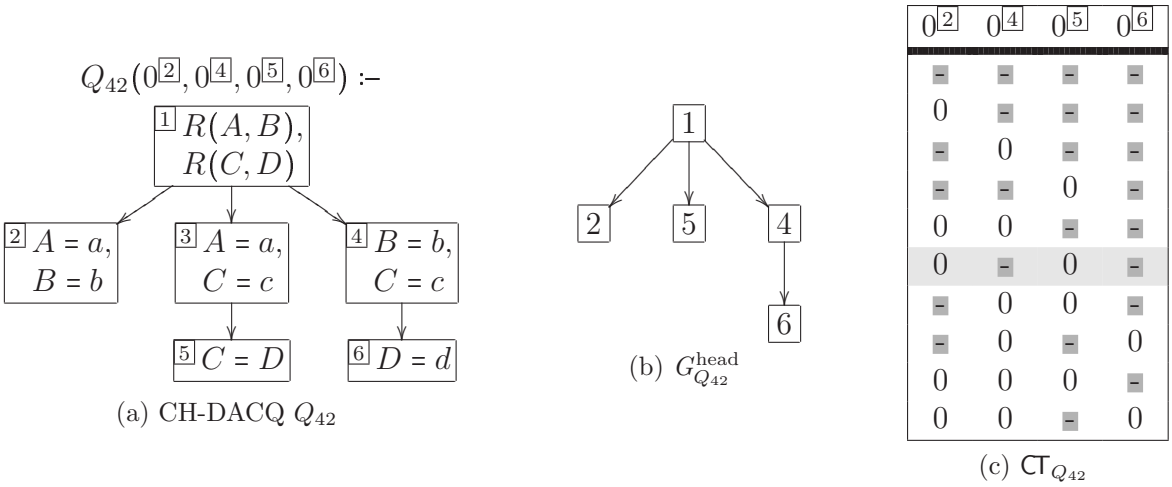
4.3 Equivalence of Constant-Headed Queries

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*.



(a) CH-DACQ Q_{42} (b) $G^{\text{head}}_{Q_{42}}$ (c) $\mathsf{CT}_{Q_{42}}$

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 $Q(\overline{\mathcal{W}})$ with head label set L, the* head graph *of Q is a rooted DAG $G^{\text{head}}_Q = (U, E)$ defined as follows.*

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

Definition 4.3.2 (Tuple Shapes) *Given DACQ $Q(\overline{\mathcal{W}})$ with head label set L, we define a* tuple shape ts *of Q to be a subset of L corresponding to the vertices of some DAG within $\mathtt{dags}(G^{\text{head}}_Q)$. We use TS_Q to denote the set of all tuple shapes of Q.*

$$\mathsf{TS}_Q := \{\mathsf{ts} \mid \exists (U, E) \in \mathtt{dags}(G^{\text{head}}_Q) \wedge \mathsf{ts} = L \cap U\} \tag{4.4}$$

We call shape $\mathsf{ts} = L$ the terminal shape for Q.

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

$$\mathsf{TS}_{42} = \{\mathsf{ts}_\varnothing, \mathsf{ts}_2, \mathsf{ts}_4, \mathsf{ts}_5, \mathsf{ts}_{24}, \mathsf{ts}_{25}, \mathsf{ts}_{45}, \mathsf{ts}_{46}, \mathsf{ts}_{245}, \mathsf{ts}_{246}, \mathsf{ts}_{2456}\}$$

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 TS_{42} *except* for the terminal shape ts_{2456}.

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 $Q(\overline{\mathcal{W}})$ and $Q'(\overline{\mathcal{W}}')$ with head graphs $G_Q^{\text{head}} = (U, E)$ and $G_{Q'}^{\text{head}} = (U', E')$, a* head graph isomorphism θ *is a bijective mapping $\theta : U \to U'$ satisfying the two properties*

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

We say that Q and Q' are head graph isomorphic *if θ exists.*

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

Theorem 4.3.4 *Let $Q(\overline{\mathcal{W}})$ and $Q'(\overline{\mathcal{W}}')$ 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 θ does not exist to establish that if θ does not exist, then there exists a tuple that only one of the queries is able to output. □

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

$$\begin{aligned}\mathtt{dags}(\mathtt{body}_Q, \mathsf{ts}) := \{G \mid G = (U, E) \in \mathtt{dags}(\mathtt{body}_Q) \\ \text{and } U \text{ contains all of the labels in } \mathsf{ts} \\ \text{and } U \text{ contains none of the labels in } L \setminus \mathsf{ts}\}\end{aligned} \tag{4.5}$$

$$\mathtt{dags}(\mathtt{body}_Q) = \bigcup_{\mathsf{ts} \in \mathsf{TS}_Q} \mathtt{dags}(\mathtt{body}_Q, \mathsf{ts}) \tag{4.6}$$

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

$$G_{\mathsf{ts}}^{\min} := G \in \mathtt{dags}(\mathtt{body}_Q, \mathsf{ts}).\ \text{all leaves of } G \text{ are in } \mathsf{ts} \tag{4.7}$$

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

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

$$\hat{Q}|_{\mathsf{ts}}(\overline{\mathcal{W}}|_{\mathsf{ts}}) :- \mathtt{body}^{+}_{G_{\mathsf{ts}}^{\min}} \tag{4.8}$$

$$\hat{Q} := \bigcup_{\mathsf{ts} \in \mathsf{TS}_Q} \hat{Q}|_{\mathsf{ts}} \tag{4.9}$$

Example 42 Continuing Example 41, consider shape $\mathsf{ts}_\varnothing \in \mathsf{TS}_{Q_{42}}$, which corresponds to the following subset of $\mathtt{dags}(\mathtt{body}_{Q_{42}})$ (subscripts denote labels within each DAG).

$$\mathtt{dags}(\mathtt{body}_{Q_{42}}, \mathsf{ts}_\varnothing) = \{G_1, G_{13}\}$$

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

$$\hat{Q}_{42}|_{\mathsf{ts}_\varnothing}(\_, \_, \_, \_) :- R(A, B), R(C, D)$$

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

$$\mathbb{T}((\hat{Q})^{\mathbb{D}}) = \{\hat{\gamma}(\overline{\mathcal{W}}) \mid \exists G \in \mathtt{dags}(\mathtt{body}_Q), \gamma : G \to \mathbb{D}\} \tag{4.10}$$

Proof. By equations 4.5 and 4.6, every $G \in \mathtt{dags}(\mathtt{body}_Q)$ corresponds to precisely one shape $\mathsf{ts} \in \mathsf{TS}_Q$, and so G is an extension of the minimal DAG $G^{\min}_{\mathsf{ts}}$. Any embedding $\gamma : G \to \mathbb{D}$ is therefore also an embedding of $G^{\min}_{\mathsf{ts}}$ into $\mathbb{D}$, and head tuple $\overline{\mathcal{W}}|_{\mathsf{ts}}$ (Definition 4.3.5) is designed so that $\hat{\gamma}(\overline{\mathcal{W}}) = \gamma(\overline{\mathcal{W}}|_{\mathsf{ts}})$ (see equation 2.22). □

Corollary 4.3.7 $Q \sqsubseteq \hat{Q}$

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

$$\forall \mathsf{ts} \in \mathsf{TS}_Q.[\hat{Q}|_{\mathsf{ts}} \equiv \hat{Q}'|_{\theta(\mathsf{ts})}]$$

where θ 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 $\mathsf{ts} \in \mathsf{TS}_Q$. 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 $\mathrm{CQ}^=$). 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 $\mathsf{ts} \in \mathsf{TS}_Q$ the set of parameter variables $\mathcal{P}_{\mathsf{ts}} \subseteq \mathcal{P}^+_Q$, as follows.

$$\mathcal{P}_{\mathsf{ts}} := \bigcup_{v \in G^{\min}_{\mathsf{ts}}} \mathcal{P}_v$$

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

- *there exists a head graph isomorphism θ between Q and Q',*
- *there exists a bijection $\nu : \mathcal{P}^+_Q \to \mathcal{P}^+_{Q'}$ satisfying $\forall \mathsf{ts} \in \mathsf{TS}_Q.[\nu(\mathcal{P}_{\mathsf{ts}}) = \mathcal{P}_{\theta(\mathsf{ts})}]$, and*

- *for each shape* $\mathsf{ts} \in \mathsf{TS}_Q$*, the CQ equivalence* $\hat{Q}|_{\mathsf{ts}} \equiv \hat{Q'}|_{\theta(\mathsf{ts})}$ *can be proved by homomorphisms* $h_{\mathsf{ts}} : \hat{Q'}|_{\theta(\mathsf{ts})} \to \hat{Q}|_{\mathsf{ts}}$ *and* $h'_{\mathsf{ts}} : \hat{Q}|_{\mathsf{ts}} \to \hat{Q'}|_{\theta(\mathsf{ts})}$ *that are consistent with* ν*,*

then $Q \equiv Q'$*.*

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

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

$$\mathsf{CT}_Q := \{\overline{\mathcal{W}}|_{\mathsf{ts}} \mid \mathsf{ts} \in \mathsf{TS}_Q \wedge \ \hat{Q}|_{\mathsf{ts}} \textit{ is satisfiable}\} \tag{4.11}$$

For a given canonical tuple $t \in \mathsf{CT}_Q$*, we use* ts^Q_t *to denote the corresponding tuple shape from* TS_Q*.*

Lemma 4.3.11 *Given CH-DACQ* Q *and any database* $\mathbb{D}$*,* $\mathbb{T}((Q)^{\mathbb{D}}) \subseteq \mathsf{CT}_Q$*.*

PROOF. By Definition 4.3.10 and Corollary 4.3.7, each $t \in \mathbb{T}((Q)^{\mathbb{D}})$ corresponds to some satisfiable query $\hat{Q}|_{\mathsf{ts}}$ with $t = \overline{\mathcal{W}}|_{\mathsf{ts}}$. □

Lemma 4.3.12 *Given CH-DACQs* Q *and* Q' *in canonical form, if* $Q \equiv Q'$ *then* $\mathsf{CT}_Q = \mathsf{CT}_{Q'}$*.*

PROOF. Follows from Definition 4.3.10 because $\theta(\mathsf{TS}_Q) = \mathsf{TS}_{Q'}$ (Theorem 4.3.4) and $\hat{Q}|_{\mathsf{ts}} \equiv \hat{Q'}|_{\theta(\mathsf{ts})}$ (Corollary 4.3.8). □

Even though the set of canonical tuples CT_Q is defined in terms of satisfiable shape queries, CT_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 $\hat{Q}_{42}|_{\mathsf{ts}_{25}}$ and $\hat{Q}_{42}|_{\mathsf{ts}_{2456}}$, whose bodies correspond to DAGs $G_{125}, G_{12456} \in \mathtt{dags}(\mathtt{body}_{Q_{42}})$.

$$\hat{Q}_{42}|_{\mathsf{ts}_{25}}(0, \blacksquare, 0, \blacksquare) \coloneq R(A,B), R(C,D), A = a, B = b, C = c, C = D$$
$$\hat{Q}_{42}|_{\mathsf{ts}_{2456}}(0,0,0,0) \coloneq R(A,B), R(C,D), A = a, B = b, C = c, C = D, D = d$$

The shaded row in Figure 4.9c is canonical tuple $t_{25} = \mathtt{head}_{\hat{Q}_{42}|_{\mathsf{ts}_{25}}}$. Shape query $\hat{Q}_{42}|_{\mathsf{ts}_{25}}$ is satisfiable, yet Q_{42} will never output t_{25} because any embedding γ of $\hat{Q}_{42}|_{\mathsf{ts}_{25}}$ is an embedding of G_{125} that can be extended to an embedding of G_{1245}, and therefore γ is not terminal. The tuple $t_{2456} = \langle\, 0,0,0,0 \,\rangle = \mathtt{head}_{\hat{Q}_{42}|_{\mathsf{ts}_{2456}}}$ is not a canonical tuple—even though $\mathsf{ts}_{2456} \in \mathsf{TS}_{Q_{42}}$—because shape query $\hat{Q}_{42}|_{\mathsf{ts}_{2456}}$ 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 $Q \overset{t}{\equiv} Q'$ to mean $\forall \mathbb{D}.[t \in (Q)^{\mathbb{D}} \iff t \in (Q')^{\mathbb{D}}]$.*

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

1. $\mathsf{CT}_Q = \mathsf{CT}_{Q'}$, *and*
2. $\forall t \in \mathsf{CT}_Q$, $Q \overset{t}{\equiv} Q'$.

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 $\boxed{u}$. This implies $\mathtt{dags}(G_Q^{\text{head}}) = \{G_u\}$, and so $\mathsf{CT}_Q = \{t\}$ for some (possibly empty) tuple t. By Definition 4.3.1, vertex u must be the root vertex of $\mathtt{body}_Q$. Because Q is in canonical form, property **[P2]** of Definition 4.2.1 implies that $\mathtt{body}_Q$ is also only a single vertex—hence, Q is simply a boolean CQ. Given query Q' with

$\mathsf{CT}_{Q'} = \mathsf{CT}_Q$, testing tuple-specific equivalence $Q \overset{t}{\equiv} Q'$ 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 ξ can be expressed as a rule of the form

$$P^\xi \to C^\xi$$

where P^ξ—called the *premise*—and C^ξ—called the *conclusion*—are both $\mathrm{CQ}^=$ 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 ξ is satisfied by database instance $\mathbb{D}$—denoted $\mathbb{D} \vDash \xi$—if each embedding of the premise into $\mathbb{D}$ can be extended to an embedding of the conclusion. Without loss of generality we can assume that $P^\xi \cap C^\xi = \varnothing$, because any atom or predicate occurring on both sides of the dependency is trivially implied and can therefore be deleted from C^ξ. Given a set of dependencies Σ and a database instance $\mathbb{D}$, we write $\mathbb{D} \vDash \Sigma$ to mean $\forall \xi \in \Sigma.(\mathbb{D} \vDash \xi)$. Given two dependency sets Σ_1 and Σ_2, we write $\Sigma_1 \vDash_{\text{fin}} \Sigma_2$ and $\Sigma_1 \equiv_{\text{fin}} \Sigma_2$ to mean finite implication $\forall \mathbb{D}.(\mathbb{D} \vDash \Sigma_1 \implies \mathbb{D} \vDash \Sigma_2)$ and finite equivalence $\forall \mathbb{D}.(\mathbb{D} \vDash \Sigma_1 \iff \mathbb{D} \vDash \Sigma_2)$, respectively ($\vDash_{\text{unr}}$ and $\equiv_{\text{unr}}$ are analogous for when the quantification over $\mathbb{D}$ allows infinite database instances). Deciding $\vDash_{\text{fin}}$ and $\equiv_{\text{fin}}$ are mutually-reducible problems,

$$\Sigma_1 \vDash_{\text{fin}} \Sigma_2 \iff \Sigma_1 \equiv_{\text{fin}} (\Sigma_1 \cup \Sigma_2) \tag{4.12}$$

$$\Sigma_1 \equiv_{\text{fin}} \Sigma_2 \iff (\Sigma_1 \vDash_{\text{fin}} \Sigma_2) \wedge (\Sigma_2 \vDash_{\text{fin}} \Sigma_1) \tag{4.13}$$

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 $\Sigma^{(Q,t)}$.

$$\xi^{(u,t)} \coloneqq \mathtt{body}^+_u \to \bot \tag{4.14}$$

$$\Sigma^{(Q,t)} \coloneqq \{\xi^{(u,t)}\} \tag{4.15}$$

Lemma 4.3.15 *For any database* $\mathbb{D}$, $t \in (Q)^{\mathbb{D}}$ *iff* $\mathbb{D} \nvDash \Sigma^{(Q,t)}$.

Proof. By equation 4.14, $\mathbb{D} \vDash \xi^{(u,t)}$ iff $\mathtt{body}^+_u$ does not embed into $\mathbb{D}$. □

Corollary 4.3.16 $Q \overset{t}{\equiv} Q'$ *iff* $\Sigma^{(Q,t)} \equiv_{\text{fin}} \Sigma^{(Q',t)}$.

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.

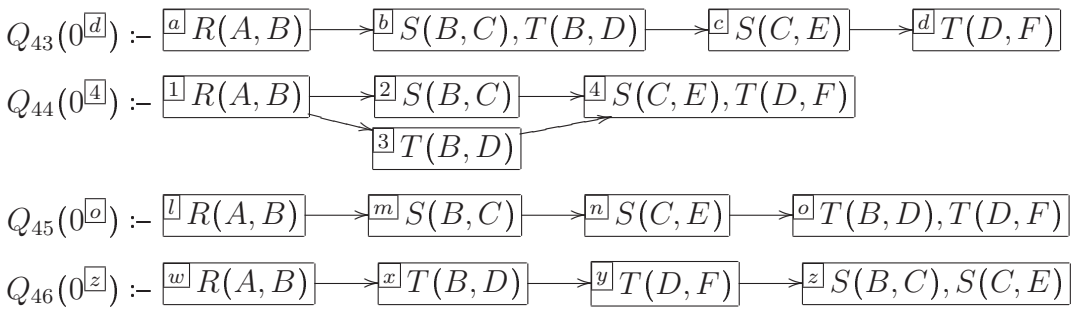


Figure 4.10: Four CH-DACQs with dual-node head graphs

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 $\boxed{u}$ to child $\boxed{v}$. This implies $\mathtt{dags}(G_Q^{\text{head}}) = \{G_{uv}, G_u\}$, and so $\mathsf{CT}_Q = \{t_1, t_2\}$ for some tuple t_1 whose corresponding shape $\mathfrak{ts}_{t_1}^Q \in \mathsf{TS}_Q$ is terminal, and a second tuple t_2 that is subsumed by t_1 (and whose corresponding shape $\mathfrak{ts}_{t_2}^Q \in \mathsf{TS}_Q$ does not contain $\boxed{v}$). Because Q is in canonical form, property **[P2]** of Definition 4.2.1 implies that $\mathtt{body}_Q$ is a rooted DAG with root u and the single leaf v, but with an arbitrary number of intermediate vertices. Given a second query Q' with $\mathsf{CT}_{Q'} = \mathsf{CT}_Q$, we can characterize the relationships $\overset{t_1}{\equiv}$ and $\overset{t_2}{\equiv}$ 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 $\{\langle\, 0\,\rangle, \langle\, \blacksquare\,\rangle\}$. 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.

$$\xi^{(v,t_1)} := \mathtt{body}_v^+ \to \bot \tag{4.16}$$

$$\Sigma^{(Q,t_1)} := \{\xi^{(v,t_1)}\} \tag{4.17}$$

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* $\mathbb{D}$, $t_1 \in (Q)^{\mathbb{D}}$ *iff* $\mathbb{D} \not\models \Sigma^{(Q,t_1)}$.

Corollary 4.3.18 $Q \overset{t_1}{\equiv} Q'$ *iff* $\Sigma^{(Q,t_1)} \equiv_{\text{fin}} \Sigma^{(Q',t_1)}$.

Example 45 Continuing Example 44, for canonical tuple $\langle\, 0\,\rangle$ all four queries in Figure 4.10 yield the same $\langle\, 0\,\rangle$-dependency set

$$\Sigma^{(Q_{43},\langle 0\rangle)} = \Sigma^{(Q_{44},\langle 0\rangle)} = \Sigma^{(Q_{45},\langle 0\rangle)} = \Sigma^{(Q_{46},\langle 0\rangle)}$$
$$= \{[R(A,B), S(B,C), T(B,D), S(C,E), T(D,F)] \to \bot\}$$

and so $Q_{43} \overset{\langle 0 \rangle}{\equiv} Q_{44} \overset{\langle 0 \rangle}{\equiv} Q_{45} \overset{\langle 0 \rangle}{\equiv} Q_{46}$ 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 $\text{CQ}^{=}$ 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 $\Sigma \vDash_{\text{fin}} \xi$ using the chase we construct a new dependency $\hat{\xi}$ by isomorphically "freezing" both the free variables and the non-free premise variables of ξ into fresh constants. Because premise $P^{\hat{\xi}}$ contains only constants, it can be treated as a database instance to be chased. Let database $\mathbb{D} := \texttt{chase}_{\Sigma}(P^{\hat{\xi}})$ denote the (possibly infinite) database resulting from any sequence of chase steps applying dependencies from Σ to the initial database $P^{\hat{\xi}}$. If $\mathbb{D}$ is finite then $\mathbb{D} \vDash \Sigma$; if $\mathbb{D}$ is infinite then the truth of $\mathbb{D} \vDash \Sigma$ is independent of the order of chase steps[3] [6, Ch. 10]. In either case, the truth of $\mathbb{D} \vDash \hat{\xi}$ is independent of the order of chase steps.[3]

The chase procedure can return the conclusion $\Sigma \vDash_{\text{fin}} \xi$ in finite time by demonstrating any finite database $\hat{\mathbb{D}} \subseteq \mathbb{D}$ corresponding to a prefix of the chase sequence yielding $\mathbb{D}$, such that either

1. there exists an embedding $\gamma : C^{\hat{\xi}} \to \hat{\mathbb{D}}$, or
2. $\hat{\mathbb{D}}$ contains a contradiction.

This proves $\hat{\mathbb{D}} \vDash \hat{\xi}$ and hence $\mathbb{D} \vDash \hat{\xi}$ (because premise $P^{\hat{\xi}}$ does not contain variables) which—by the isomorphic freezing—implies $\Sigma \vDash_{\text{unr}} \xi$ and hence $\Sigma \vDash_{\text{fin}} \xi$.

The chase procedure only returns the conclusion $\Sigma \nvDash_{\text{fin}} \xi$ when $\mathbb{D}$ is finite and $\mathbb{D} \nvDash \hat{\xi}$ (i.e., $\mathbb{D}$ does not contain a contradiction and there does not exist an embedding of $C^{\hat{\xi}}$ into $\mathbb{D}$). Hence, $\Sigma \vDash_{\text{fin}} \xi$ can only be decided by the chase algorithm in cases where there exists a finite chase sequence for $\texttt{chase}_{\Sigma}(P^{\hat{\xi}})$—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 $\texttt{dags}(\texttt{body}_Q, \texttt{ts}^Q_{t_2})$ contains all of the DAGs in $\texttt{dags}(\texttt{body}_Q)$ containing node u but not node v. For each $G \in \texttt{dags}(\texttt{body}_Q, \texttt{ts}^Q_{t_2})$ we define the following embedded dependency $\xi^{(G,t_2)}$, and we collect all of these dependencies into the dependency set $\Sigma^{(Q,t_2)}$.

$$\xi^{(G,t_2)} := \texttt{body}^+_G \to \texttt{body}^+_v \tag{4.18}$$

$$\Sigma^{(Q,t_2)} := \{\xi^{(G,t_2)} \mid G \in \texttt{dags}(\texttt{body}_Q, t_2)\} \tag{4.19}$$

[3] As long as the infinite chase sequence obeys certain basic properties—primarily, that no dependency is "starved."

(When constructing $\xi^{(G,t_2)}$ from equation 4.18, any trivially-implied atoms are deleted from the conclusion.)

Example 46 Continuing Examples 44 and 45, for canonical tuple $\langle \blacksquare \rangle$ the queries in Figure 4.10 yield the following $\langle \blacksquare \rangle$-dependency sets.

$$\begin{aligned}
\Sigma^{(Q_{43},\langle\blacksquare\rangle)} = \{\ & \xi^{(a,\langle\blacksquare\rangle)} : [R(A,B)] \to [S(B,C), T(B,D), S(C,E), T(D,F)], \\
& \xi^{(ab,\langle\blacksquare\rangle)} : [R(A,B), S(B,C), T(B,D)] \to [S(C,E), T(D,F)], \\
& \xi^{(abc,\langle\blacksquare\rangle)} : [R(A,B), S(B,C), T(B,D), S(C,E)] \to [T(D,F)]\ \} \\
\Sigma^{(Q_{44},\langle\blacksquare\rangle)} = \{\ & \xi^{(1,\langle\blacksquare\rangle)} : [R(A,B)] \to [S(B,C), T(B,D), S(C,E), T(D,F)], \\
& \xi^{(12,\langle\blacksquare\rangle)} : [R(A,B), S(B,C)] \to [T(B,D), S(C,E), T(D,F)], \\
& \xi^{(13,\langle\blacksquare\rangle)} : [R(A,B), T(B,D)] \to [S(B,C), S(C,E), T(D,F)], \\
& \xi^{(123,\langle\blacksquare\rangle)} : [R(A,B), S(B,C), T(B,D)] \to [S(C,E), T(D,F)]\ \} \\
\Sigma^{(Q_{45},\langle\blacksquare\rangle)} = \{\ & \xi^{(l,\langle\blacksquare\rangle)} : [R(A,B)] \to [S(B,C), S(C,E), T(B,D), T(D,F)], \\
& \xi^{(lm,\langle\blacksquare\rangle)} : [R(A,B), S(B,C)] \to [S(C,E), T(B,D), T(D,F)], \\
& \xi^{(lmn,\langle\blacksquare\rangle)} : [R(A,B), S(B,C), S(C,E)] \to [T(B,D), T(D,F)]\ \} \\
\Sigma^{(Q_{46},\langle\blacksquare\rangle)} = \{\ & \xi^{(w,\langle\blacksquare\rangle)} : [R(A,B)] \to [T(B,D), T(D,F), S(B,C), S(C,E)], \\
& \xi^{(wx,\langle\blacksquare\rangle)} : [R(A,B), T(B,D)] \to [T(D,F), S(B,C), S(C,E)], \\
& \xi^{(wxy,\langle\blacksquare\rangle)} : [R(A,B), T(B,D), T(D,F)] \to [S(B,C), S(C,E)]\ \}
\end{aligned}$$

Lemma 4.3.19 *For any database* $\mathbb{D}$, $t_2 \in (Q)^{\mathbb{D}}$ *iff* $\mathbb{D} \not\models \Sigma^{(Q,t_2)}$.

PROOF. Suppose that $t_2 \in (Q)^{\mathbb{D}}$. Then, there exists some terminal embedding $\gamma : G \to \mathbb{D}$ of some DAG $G \in \mathtt{dags}(\mathtt{body}_Q, \mathtt{ts}_{t_2}^Q)$ which doesn't contain v; however, γ is also an embedding of $P^{\xi^{(G,t_2)}}$ that does not extend to $C^{\xi^{(G,t_2)}}$ and so $\mathbb{D} \not\models \xi^{(G,t_2)}$. Conversely, let G be a maximal DAG in $\mathtt{dags}(\mathtt{body}_Q, \mathtt{ts}_{t_2}^Q)$ such that $\mathbb{D} \not\models \xi^{(G,t_2)}$. Then, there exists embedding $\gamma : P^{\xi^{(G,t_2)}} \to \mathbb{D}$ that does not extend to $C^{\xi^{(G,t_2)}}$; however, γ is a terminal embedding of G (by choice of G) and so generates tuple $t_2 \in (Q)^{\mathbb{D}}$. □

Corollary 4.3.20 $Q \overset{t_2}{\equiv} Q'$ *iff* $\Sigma^{(Q,t_2)} \equiv_{\text{fin}} \Sigma^{(Q',t_2)}$.

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.

$Q_{43} \overset{\langle\blacksquare\rangle}{\equiv} Q_{44}$:

$\Sigma^{(Q_{43},\langle\blacksquare\rangle)} \models_{\text{fin}} \xi^{(1,\langle\blacksquare\rangle)}$: Trivial

$\Sigma^{(Q_{43},\langle\blacksquare\rangle)} \models_{\text{fin}} \xi^{(12,\langle\blacksquare\rangle)}$:

$$\begin{aligned}
\mathbb{D} &:= \mathtt{chase}_{\Sigma^{(Q_{43},\langle\blacksquare\rangle)}}([R(a,b), S(b,c)]) \\
&= \{R(a,b), S(b,c), S(b,c_1), T(b,d_1), S(c_1,e_1), T(d_1,f_1), S(c,e_2)\} \\
&\models [T(b,D), S(c,E), T(D,F)]
\end{aligned}$$

$\Sigma^{(Q_{43},\langle\blacksquare\rangle)} \models_{\text{fin}} \xi^{(13,\langle\blacksquare\rangle)}$:

$$\begin{aligned}
\mathbb{D} &:= \mathtt{chase}_{\Sigma^{(Q_{43},\langle\blacksquare\rangle)}}([R(a,b), T(b,d)]) \\
&= \{R(a,b), T(b,d), S(b,c_1), T(b,d_1), S(c_1,e_1), T(d_1,f_1), T(d,f_2)\} \\
&\models [S(b,C), S(C,E), T(d,F)]
\end{aligned}$$

$\Sigma^{(Q_{43},\langle\blacksquare\rangle)} \models_{\text{fin}} \xi^{(123,\langle\blacksquare\rangle)}$: Trivial

$\Sigma^{(Q_{44},\langle - \rangle)} \models_{\text{fin}} \xi^{(a,\langle - \rangle)}$: Trivial
$\Sigma^{(Q_{44},\langle - \rangle)} \models_{\text{fin}} \xi^{(ab,\langle - \rangle)}$: Trivial
$\Sigma^{(Q_{44},\langle - \rangle)} \models_{\text{fin}} \xi^{(abc,\langle - \rangle)}$:
$\mathbb{D} \coloneqq \mathtt{chase}_{\Sigma^{(Q_{44},\langle - \rangle)}}([R(a,b), S(b,c), T(b,d), S(c,e)])$
$= \{R(a,b), S(b,c), T(b,d), S(c,e), S(c,e_1), T(d,f_1)\}$
$\models [T(d,F)]$

$Q_{43} \overset{\langle - \rangle}{\not\equiv} Q_{45}$: $\Sigma^{(Q_{45},\langle - \rangle)} \not\models_{\text{fin}} \xi^{(ab,\langle - \rangle)}$:
$\mathbb{D} \coloneqq \mathtt{chase}_{\Sigma^{(Q_{45},\langle - \rangle)}}([R(a,b), S(b,c), T(b,d)])$
$= \{R(a,b), S(b,c), T(b,d), S(c,e_1), T(b,d_1), T(d_1,f_1)\}$
$\not\models [S(c,E), T(d,F)]$

$Q_{43} \overset{\langle - \rangle}{\not\equiv} Q_{46}$: $\Sigma^{(Q_{46},\langle - \rangle)} \not\models_{\text{fin}} \xi^{(ab,\langle - \rangle)}$:
$\mathbb{D} \coloneqq \mathtt{chase}_{\Sigma^{(Q_{46},\langle - \rangle)}}([R(a,b), S(b,c), T(b,d)])$
$= \{R(a,b), S(b,c), T(b,d), T(d,f_1), S(b,c_1)S(c_1,e_1)\}$
$\not\models [S(c,E), T(d,F)]$

$Q_{45} \overset{\langle - \rangle}{\not\equiv} Q_{46}$: $\Sigma^{(Q_{46},\langle - \rangle)} \not\models_{\text{fin}} \xi^{(lm,\langle - \rangle)}$:
$\mathbb{D} \coloneqq \mathtt{chase}_{\Sigma^{(Q_{46},\langle - \rangle)}}([R(a,b), S(b,c)])$
$= \{R(a,b), S(b,c), T(b,d_1), T(d_1,f_1), S(b,c_1)S(c_1,e_1)\}$
$\not\models [S(c,E), T(b,D), T(D,F)]$

The equivalence $Q_{43} \equiv Q_{44}$ also follows from Theorem 4.3.9. Queries Q_{43} and Q_{44} have parameter sets $\mathcal{P}^+_{Q_{43}} = \mathcal{P}^+_{Q_{44}} = \{B, C, D\}$ with bijection ν being the obvious identity mapping. The obvious identity mapping also forms the homomorphism between the shape queries for shapes $\mathsf{ts}^{Q_{43}}_{\langle - \rangle} = \varnothing = \mathsf{ts}^{Q_{44}}_{\langle - \rangle}$

$$\hat{Q}_{43}|_{\varnothing}(-) \coloneq R(A,B)$$
$$\hat{Q}_{44}|_{\varnothing}(-) \coloneq R(A,B)$$

and shape queries for shapes $\mathsf{ts}^{Q_{43}}_{\langle 0 \rangle} = \{\boxed{d}\} \in \mathsf{TS}_{Q_{43}}$ and $\mathsf{ts}^{Q_{44}}_{\langle 0 \rangle} = \{\boxed{4}\} \in \mathsf{TS}_{Q_{46}}$.

$$\hat{Q}_{43}|_{d}(0) \coloneq R(A,B), S(B,C), T(B,D), S(C,E), T(D,F)$$
$$\hat{Q}_{44}|_{4}(0) \coloneq R(A,B), S(B,C), T(B,D), S(C,E), T(D,F)$$

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 $\hat{Q}_{45}|_o$ and $\hat{Q}_{46}|_z$ 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 ξ can be expressed as a rule of the form

$$P^{\xi} \to C_1^{\xi} \mid \ldots \mid C_n^{\xi} \tag{4.20}$$

where the premise P^{ξ} and each conclusion C_i^{ξ} are $\mathrm{CQ}^{=}$ bodies; ξ is satisfied by database instance $\mathbb{D}$ if each embedding of the premise into $\mathbb{D}$ 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 $t \in \mathsf{CT}_Q$ corresponding to shape $\mathfrak{ts}_t^Q \in \mathsf{TS}_Q$, let $\{\boxed{v_1}, \ldots \boxed{v_n}\} \subseteq L$ be the set of vertices in head graph G_Q^{head} that do not occur in shape $\mathfrak{ts}_t^Q$ but are children of vertices in $\mathfrak{ts}_t^Q$. For each $G \in \mathtt{dags}(\mathtt{body}_Q, \mathfrak{ts}_t^Q)$ we define a single embedded disjunctive dependency $\xi^{(G,t)}$

$$\xi^{(G,t)} := \begin{cases} \mathtt{body}_G^+ \to \bot & \text{if } n = 0 \\ \mathtt{body}_G^+ \to \mathtt{body}_{v_1}^+ \mid \ldots \mid \mathtt{body}_{v_n}^+ & \text{otherwise} \end{cases} \tag{4.21}$$

and we define dependency set $\Sigma^{(Q,t)}$ as follows.

$$\Sigma^{(Q,t)} := \{\xi^{(G,t)} \mid G \in \mathtt{dags}(\mathtt{body}_Q, \mathfrak{ts}_t^Q)\} \tag{4.22}$$

Note that $n = 0$ if and only if $\mathfrak{ts}_t^Q$ is terminal if and only if $\mathtt{dags}(\mathtt{body}_Q, \mathfrak{ts}_t^Q) = \{\mathtt{body}_Q\}$ (by property [**P2**] of Definition 4.2.1).

Lemma 4.3.21 *For any database $\mathbb{D}$ and canonical tuple t, $t \in (Q)^{\mathbb{D}}$ iff $\mathbb{D} \not\models \Sigma^{(Q,t)}$.*

Proof. Suppose that $t \in (Q)^{\mathbb{D}}$. Then, there exists some terminal embedding $\gamma : G \rightsquigarrow \mathbb{D}$ of some DAG $G \in \mathtt{dags}(\mathtt{body}_Q, \mathfrak{ts}_t^Q)$. However, γ is also an embedding of premise query $P^{\xi^{(G,t)}}$. If $\mathfrak{ts}_t^Q$ is not the terminal shape then γ does not extend to any conclusion query $C_i^{\xi^{(G,t)}}$, whereas if $\mathfrak{ts}_t^Q$ is terminal then the conclusion of $\xi^{(G,t)}$ is a contradiction to which no embedding can extend; in either case, $\mathbb{D} \not\models \xi^{(G,t)}$.

Conversely, let G be a maximal DAG in $\mathtt{dags}(\mathtt{body}_Q, \mathfrak{ts}_t^Q)$ such that $\mathbb{D} \not\models \xi^{(G,t)}$. There exists embedding $\gamma : P^{\xi^{(G,t)}} \to \mathbb{D}$ that does not extend to any conclusion of $\xi^{(G,t)}$; however, γ is a terminal embedding of G that generates tuple $t \in (Q)^{\mathbb{D}}$. □

Corollary 4.3.22 *For any canonical tuple $t \in \mathsf{CT}_Q$, $Q \overset{t}{\equiv} Q'$ 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 $\overline{\mathcal{W}}$ 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* $\mathcal{C}_S$. Choose $\mathcal{C}_S$ as any strict subset of $\mathcal{C}$ such that both $\mathcal{C}_S$ and $\mathcal{C} \setminus \mathcal{C}_S$ are infinite. Without loss of generality, we will assume that $\mathcal{C}_S$ is chosen so that for any given set of queries, all constants appearing in the queries are from the set $\mathcal{C} \setminus \mathcal{C}_S$. Because the constants in $\mathcal{C}_S$ 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 $(\mathcal{C}_Q \cup \mathcal{C}_{Q'}) \cap \mathcal{C}_S = \varnothing$), if $Q \not\equiv Q'$ then there exists a database instance $\mathbb{D}$ with $\texttt{adom}(\mathbb{D}) \subset \mathcal{C}_S \cup \mathcal{C}_Q \cup \mathcal{C}_{Q'}$ such that $(Q)^{\mathbb{D}} \neq (Q')^{\mathbb{D}}$.*

Given any two tuples t and t' over domain $\mathcal{C} \cup \{\text{-}\}$, we say that t *is $\mathcal{C}_S$-homomorphic to t'*—denoted $t \gtrsim t'$—if there exists a mapping $h : \mathcal{C}_S \to \mathcal{C}$, extended with identity over $(\mathcal{C} \setminus \mathcal{C}_S) \cup \{\text{-}\}$, such that $h(t) = t'$. We say that t *is $\mathcal{C}_S$-isomorphic to t'*—denoted $t \approx t'$—if $t \gtrsim t'$ and $t' \gtrsim t$. We say that t *is more general than t'*—denoted $t \succ t'$—if $t \gtrsim t'$ and $t' \not\gtrsim t$.

Given two sets of tuples S_1, S_2 over $\mathcal{C}$, we say that S_1 is $\mathcal{C}_S$-contained in S_2—denoted $S_1 \lesssim S_2$—if $\forall t_1 \in S_1 \exists t_2 \in S_2.(t_1 \approx t_2)$. We say that S_1 is $\mathcal{C}_S$-equivalent to S_2—denoted $S_1 \approx S_2$—if $S_1 \lesssim S_2$ and $S_2 \lesssim S_1$.

In order to define the canonical tuples for a DACQ, we first define the *canonical freezings* of a satisfiable CQ$^{=}$. Any tuple that can be output by the CQ$^{=}$ 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$^=$ $Q(\overline{\mathcal{W}})$ with $\mathcal{C}_B \cap \mathcal{C}_S = \varnothing$, let $\mathcal{H}_Q$ denote the set of all homomorphisms $h : \mathcal{W} \cap \mathcal{B} \to \mathcal{C}_S \cup \mathcal{C}_Q$ that have been extended with identity over all symbols not in $\mathcal{W} \cap \mathcal{B}$. The (infinite) set of* valid freezings *of Q—denoted $\mathbb{CF}^+_Q$—contains the homomorphic images of head tuple $\overline{\mathcal{W}}$ under mappings from $\mathcal{H}_Q$ that are consistent with the equality predicates in the query body.*

$$\mathbb{CF}^+_Q \coloneqq \{h(\overline{\mathcal{W}}) \mid h \in \mathcal{H}_Q \wedge h(\mathtt{pred}_Q) \not\models \bot\} \tag{4.23}$$

The set of canonical freezings *of Q—denoted $\mathbb{CF}_Q$—is a* minimal *subset of $\mathbb{CF}^+_Q$ satisfying $\mathbb{CF}_Q \approxeq \mathbb{CF}^+_Q$ (if many such sets exist, we arbitrarily choose one).*

Set $\mathbb{CF}_Q$ is finite, with size bounded by a function exponential in both $|\mathcal{C}_Q|$ and $|\mathcal{W} \cap \mathcal{B}|$.

Lemma 4.4.3 *Given CQ$^=$ Q and any database instance $\mathbb{D}$ over $\mathcal{C}_S \cup \mathcal{C}_Q$, $\mathbb{T}((Q)^{\mathbb{D}}) \subseteq \mathbb{CF}_Q$.*

Proof. Any embedding $\gamma : Q \to \mathbb{D}$ yields $\gamma(\overline{\mathcal{W}}) \in \mathbb{CF}^+_Q$ (by Definition 4.4.2), and so $\mathbb{T}((Q)^{\mathbb{D}}) \subseteq \mathbb{CF}^+_Q \subseteq \mathbb{CF}_Q$. □

Definition 4.4.4 (Canonical Tuples) *Given DACQ Q in canonical form, the set $\mathbb{CT}_Q$ of* canonical tuples for *Q corresponds to the canonical freezings of all satisfiable shape queries.*

$$\mathbb{CT}_Q \coloneqq \{t \mid \exists \mathtt{ts} \in \mathbb{TS}_Q \wedge \hat{Q}|_{\mathtt{ts}} \text{ is satisfiable} \wedge t \in \mathbb{CF}_{\hat{Q}|_{\mathtt{ts}}}\}$$

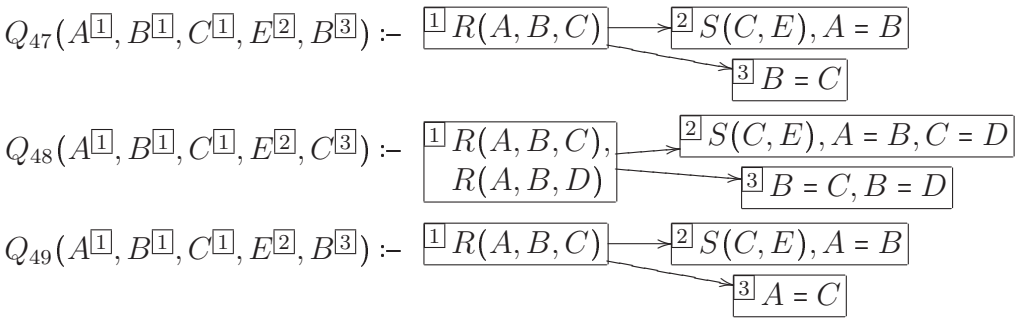
As defined previously for CH-DACQs, we use $\mathtt{ts}^Q_t$ to denote the tuple shape from $\mathbb{TS}_Q$ corresponding to a given canonical tuple $t \in \mathbb{CT}_Q$.

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,

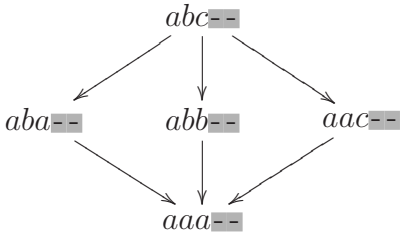
$$\mathbb{TS}_{Q_{47}} = \mathbb{TS}_{Q_{48}} = \mathbb{TS}_{Q_{49}} = \{\mathtt{ts}_1, \mathtt{ts}_{12}, \mathtt{ts}_{13}, \mathtt{ts}_{123}\}$$

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 $\mathbb{CT}_{Q_{47}} = \mathbb{CT}_{Q_{48}}$ even though the head tuples of Q_{47} and Q_{48} are not isomorphic. In contrast, $\mathbb{CT}_{Q_{47}} \neq \mathbb{CT}_{Q_{49}}$ (due to shape $\mathtt{ts}_{13}$) even though the head tuples of Q_{47} and Q_{49} are isomorphic.

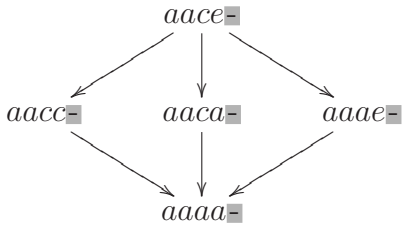
Lemma 4.4.5 *Given DACQ Q in canonical form and any database instance $\mathbb{D}$ over $\mathcal{C}_S \cup \mathcal{C}_Q$, $\mathbb{T}((Q)^{\mathbb{D}}) \subseteq \mathbb{CT}_Q$.*



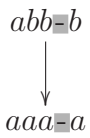
(a) DACQs Q_{47}, Q_{48}, and Q_{49}



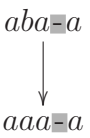
(b) $\mathbb{CF}_{\hat{Q}_{47}|_{\mathbb{s}_1}} = \mathbb{CF}_{\hat{Q}_{48}|_{\mathbb{s}_1}} = \mathbb{CF}_{\hat{Q}_{49}|_{\mathbb{s}_1}}$



(c) $\mathbb{CF}_{\hat{Q}_{47}|_{\mathbb{s}_{12}}} = \mathbb{CF}_{\hat{Q}_{48}|_{\mathbb{s}_{12}}} = \mathbb{CF}_{\hat{Q}_{49}|_{\mathbb{s}_{12}}}$

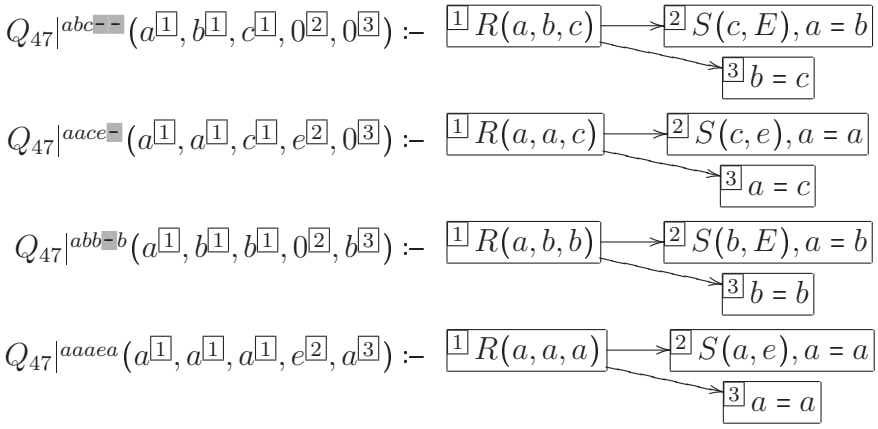


(d) $\mathbb{CF}_{\hat{Q}_{47}|_{\mathbb{s}_{13}}} = \mathbb{CF}_{\hat{Q}_{48}|_{\mathbb{s}_{13}}}$



(e) $\mathbb{CF}_{\hat{Q}_{49}|_{\mathbb{s}_{13}}}$

(f) $\mathbb{CF}_{\hat{Q}_{47}|_{\mathbb{s}_{123}}} = \mathbb{CF}_{\hat{Q}_{48}|_{\mathbb{s}_{123}}} = \mathbb{CF}_{\hat{Q}_{49}|_{\mathbb{s}_{123}}}$



(g) Frozen DACQs obtained from Q_{47}

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 $\mathsf{CT}_Q \approx \mathsf{CT}_{Q'}$.*

PROOF. Follows from Definition 4.4.4 because $\theta(\mathsf{TS}_Q) = \mathsf{TS}_{Q'}$ (Theorem 4.3.4), and $\hat{Q}|_{\mathtt{ts}} \equiv \hat{Q}'|_{\theta(\mathtt{ts})}$ (Corollary 4.3.8) implies $\mathsf{CF}_{\hat{Q}|_{\mathtt{ts}}} \approx \mathsf{CF}_{\hat{Q}'|_{\theta(\mathtt{ts})}}$ (by Definition 4.4.2). □

Corollary 4.4.7 *If $Q \equiv Q'$ then CT_Q and $\mathsf{CT}_{Q'}$ can be chosen so that $\mathsf{CT}_Q = \mathsf{CT}_{Q'}$.*

Definition 4.4.8 (Frozen DACQ) *Given DACQ $Q(\overline{\mathcal{W}})$ in canonical form and any canonical tuple $t \in \mathsf{CT}_Q$ with corresponding shape $\mathtt{ts}_t^Q \in \mathsf{TS}_Q$,*

- *recall that t is a canonical freezing of shape query $\hat{Q}|_{\mathtt{ts}_t^Q}$ (cf. Definition 4.4.4),*
- *recall that $\hat{Q}|_{\mathtt{ts}_t^Q}$ is a $CQ^=$ with head tuple $\overline{\mathcal{W}}|_{\mathtt{ts}_t^Q}$ (cf. Definition 4.3.5), and*
- *recall that there exists some homomorphism $h_t : \mathcal{W}|_{\mathtt{ts}_t^Q} \cap \mathcal{B} \to \mathcal{C}_S \cup \mathcal{C}_Q$ extended with identity over all symbols not in $\mathcal{W}|_{\mathtt{ts}_t^Q} \cap \mathcal{B}$ such that $t = h_t(\overline{\mathcal{W}}|_{\mathtt{ts}_t^Q})$ (cf. Definition 4.4.2).*

We define the freezing of Q with t*—denoted $Q|^t$—as the following CH-DACQ.*

- $\mathtt{head}_{Q|^t} := \overline{\mathcal{W}}|^t$, *where $\overline{\mathcal{W}}|^t$ is constructed from $\overline{\mathcal{W}}$ by locating each component $W_i^{\boxed{v}}$ in which W_i is a variable, and if $v \in \mathtt{ts}_t^Q$ replacing W_i with $h_t(W_i)$, whereas if $v \notin \mathtt{ts}_t^Q$ replacing W_i with an arbitrary constant from $\mathcal{C} \setminus \mathcal{C}_S$ (for simplicity, we always use the constant 0).*
- $\mathtt{body}_{Q|^t} := h_t(\mathtt{body}_Q)$

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, $\forall t \in \mathsf{CT}_Q.[Q \overset{t}{\equiv} Q|^t]$.*

PROOF. Consider any database $\mathbb{D}$. Suppose that $t \in (Q)^{\mathbb{D}}$; then there exists some terminal embedding $\gamma : Q \rightsquigarrow \mathbb{D}$ satisfying $\hat{\gamma}(\overline{\mathcal{W}}) = t$. This implies that there exists some $G \in \mathtt{dags}(\mathtt{body}_Q, \mathtt{ts}_t^Q)$ such that γ is a terminal embedding of G (see equation 4.5). By Theorem 4.3.6, γ is also an embedding of $\hat{Q}|_{\mathtt{ts}_t^Q}$ into $\mathbb{D}$ satisfying $\hat{\gamma}(\overline{\mathcal{W}}|_{\mathtt{ts}_t^Q}) = t$, and so, by definition of $\overline{w}|^t$, $\hat{\gamma}(\overline{\mathcal{W}}|^t = \hat{\gamma}(\overline{\mathcal{W}}|_{\mathtt{ts}_t^Q}) = t = h_t(\overline{\mathcal{W}}|_{\mathtt{ts}_t^Q})$. Because mappings γ and h_t agree over $\overline{\mathcal{W}}|_{\mathtt{ts}_t^Q}$ and h_t is the identity mapping over all other symbols, we can decompose embedding γ as $\gamma = \phi \circ h_t$ for some mapping $\phi : \mathcal{B} \to \mathtt{adom}(\mathbb{D})$ extended with identity over constants $\mathcal{C}$. Because $Q|^t$ has body $h_t(\mathtt{body}_Q)$, mapping ϕ is an embedding of $h_t(G)$ into $\mathbb{D}$ satisfying $\hat{\phi}(\overline{\mathcal{W}}|^t) = t$. This implies $t \in (Q|^t)^{\mathbb{D}}$.

Now suppose that $t \in (Q|^t)^{\mathbb{D}}$; then there exists some $G' \in \mathtt{dags}(h_t(\mathtt{body}_Q))$ and some terminal embedding $\phi : G' \rightsquigarrow \mathbb{D}$ satisfying $\hat{\phi}(\overline{\mathcal{W}}|^t) = t$, implying[4] $G' \in \mathtt{dags}(h_t(\mathtt{body}_Q), \mathtt{ts}_t^Q)$. Then $\gamma := \phi \circ h_t$ is a terminal embedding of the DAG $G \in \mathtt{dags}(\mathtt{body}_Q)$ satisfying $G' = h_t(G)$, and $\hat{\gamma}(\overline{\mathcal{W}}) = t$ because $G \in \mathtt{dags}(\mathtt{body}_Q, \mathtt{ts}_t^Q)$. □

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. $\mathsf{CT}_Q \approx \mathsf{CT}_{Q'}$, *and*
2. $\forall t \in \mathsf{CT}_Q$, $Q|^t \overset{t}{\equiv} Q'|^t$.

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 $aace$-.

$$Q_{48}|^{abc\text{--}}(a^{\boxed{1}}, b^{\boxed{1}}, c^{\boxed{1}}, 0^{\boxed{2}}, 0^{\boxed{3}}) \text{ :- } \boxed{1}\,R(a,b,c),\ R(a,b,D) \rightarrow \boxed{2}\,S(c,E), a = b, c = D;\ \boxed{3}\,b = c, b = D$$

$$Q_{48}|^{aace\text{-}}(a^{\boxed{1}}, a^{\boxed{1}}, c^{\boxed{1}}, e^{\boxed{2}}, 0^{\boxed{3}}) \text{ :- } \boxed{1}\,R(a,a,c),\ R(a,a,D) \rightarrow \boxed{2}\,S(c,e), a = a, c = D;\ \boxed{3}\,a = c, a = D$$

[4]We needed to insert arbitrary constants into head $\overline{\mathcal{W}}|^t$ of query $Q|^t$ so that $\overline{\mathcal{W}}|^t$ retains the same labels and tuple shapes as $\overline{\mathcal{W}}$. If we instead defined $\overline{\mathcal{W}}|^t := t$—which effectively removes any output labels not in $\mathtt{ts}_t^Q$—then $Q|^t$ would output the tuple t too often (in particular, $Q|^t$ would incorrectly output t for any terminal embedding $\phi' : G' \rightsquigarrow \mathbb{D}$ of any $G' \in \mathtt{dags}(h_t(\mathtt{body}_Q), \mathtt{ts}')$ for which $\mathtt{ts}_t^Q \subset \mathtt{ts}'$).

From the queries in Figure 4.11g it is straightforward to verify the tuple-specific equivalences. $Q_{47}|^{abc--} \overset{\langle abc-- \rangle}{\equiv} Q_{48}|^{abc--}$ and $Q_{47}|^{aace-} \overset{\langle aace- \rangle}{\equiv} Q_{48}|^{aace-}$. In fact, for any canonical tuple t with shape ts_{12}, ts_{13}, or ts_{123} we can show that the t-specific equivalence $Q_{47}|^t \overset{t}{\equiv} Q_{48}|^t$ holds (even though the generic equivalence $Q_{47}|^t \equiv Q_{48}|^t$ may not hold). In contrast, for shape ts_1, tuple-specific equivalence holds for the most general freezing $abc--$, but the database $\mathbb{D}_7 = \{R(a,a,c), R(a,a,d), S(c,e)\}$ proves $Q_{47}|^{aac--} \overset{\langle aac-- \rangle}{\not\equiv} Q_{48}|^{aac--}$ and therefore $Q_{47} \overset{\langle aac-- \rangle}{\not\equiv} Q_{48}$, as demonstrated by the following relations $(Q_{47})^{\mathbb{D}_7}$ and $(Q_{48})^{\mathbb{D}_7}$.

$(Q_{47})^{\mathbb{D}_7}$

$A^{\boxed{1}}$	$B^{\boxed{1}}$	$C^{\boxed{1}}$	$E^{\boxed{2}}$	$B^{\boxed{3}}$
a	a	c	e	-
a	a	d	-	-

$(Q_{48})^{\mathbb{D}_7}$

$A^{\boxed{1}}$	$B^{\boxed{1}}$	$C^{\boxed{1}}$	$E^{\boxed{2}}$	$C^{\boxed{3}}$
a	a	c	e	-
a	a	c	-	-
a	a	d	-	-

The "problem" illustrated by Example 50 is that the tuple-specific equivalence $Q_{47}|^{abc--} \overset{\langle abc-- \rangle}{\equiv} Q_{48}|^{abc--}$ does not imply $Q_{47}|^{aac--} \overset{\langle aac-- \rangle}{\equiv} Q_{48}|^{aac--}$, even though canonical tuple $\langle\, abc-- \,\rangle$ is more general than $\langle\, aac-- \,\rangle$. 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 $Q|^{t_1} \overset{t_1}{\equiv} Q|^{t_1}$ also implies $Q|^{t_2} \overset{t_2}{\equiv} Q|^{t_2}$, for two canonical tuples satisfying $t_1 \succ t_2$. 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 ξ (see equation 4.20) and a set of disjunctive embedded dependencies Σ, the chase $\mathtt{chase}_\Sigma(P^{\hat{\xi}})$ 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 $\mathbb{D}_p$. The chase procedure returns the conclusion $\Sigma \models_{\text{fin}} \xi$ if and only if for each path p it can prove $\mathbb{D}_p \models \hat{\xi}$, which it does by exhibiting a finite database $\hat{\mathbb{D}}_p \subseteq \mathbb{D}_p$—corresponding to some finite prefix of path p—that satisfies $\hat{\mathbb{D}}_p \models \hat{\xi}$. Given any mapping $h : \mathcal{C} \to \mathcal{C}$, and a specific chase invocation that yielded the conclusion $\Sigma \models_{\text{fin}} \xi$, we say that the conclusion is *sound w.r.t. h* if for each path p, whenever the database $\hat{\mathbb{D}}_p$ contains a contradiction, so does the database $h(\hat{\mathbb{D}}_p)$.

Theorem 4.4.11 *Given DACQs Q and Q' in canonical form satisfying $\mathsf{CT}_Q = \mathsf{CT}_{Q'}$, and given two canonical tuples $t_1, t_2 \in \mathsf{CT}_Q$ satisfying $t_1 \succ t_2$, let h be the $\mathcal{C}_S$-homomorphism (extended with identity) satisfying $h(t_1) = t_2$. If $Q|^{t_1} \overset{t_1}{\equiv} Q'|^{t_1}$ (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 $Q|^{t_2} \overset{t_2}{\equiv} Q'|^{t_2}$ (i.e., $\Sigma^{(Q|^{t_2}, t_2)} \equiv_{\text{fin}} \Sigma^{(Q'|^{t_2}, t_2)}$).*

PROOF. Suppose that $Q|^{t_1} \overset{t_1}{\equiv} Q|^{t_1}$. Due to Definition 4.4.8 and equations 4.21 and 4.22, we observe that $\Sigma^{(Q|^{t_2},t_2)} = h(\Sigma^{(Q|^{t_1},t_1)})$ and $\Sigma^{(Q'|^{t_2},t_2)} = h(\Sigma^{(Q'|^{t_1},t_1)})$.

Consider any chase invocation that proves some implication $\Sigma \vDash_{\text{fin}} \xi$ as part of the proof that $\Sigma^{(Q|^{t_1},t_1)} \equiv_{\text{fin}} \Sigma^{(Q'|^{t_1},t_1)}$. Let T denote a (possibly infinite) chase tree for $\mathtt{chase}_\Sigma(P^{\hat{\xi}})$, with $\mathbb{D}_p$ 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 $\hat{\mathbb{D}}_p \subseteq \mathbb{D}_p$ the finite database that terminated the search of the chase sequence along path p.

Define $\mathbb{D}_1 := P^{\hat{\xi}}$, and let $\mathbb{D}_1 \xrightarrow{(\xi_i,\gamma)} \{\mathbb{D}_{1.1},\ldots,\mathbb{D}_{1.k}\}$ be the first chase step in chase tree T enacted by some variable mapping $\gamma : X \to \mathtt{adom}(\mathbb{D}_1)$, where X is the set of premise and free variables of some dependency $\xi_i \in \Sigma$ having k conclusions, and γ embeds premise P^{ξ_i} into $\mathbb{D}_1$. Let $\mathbb{D}_{1.j}$ denote the database obtained from $\mathbb{D}_1$ by chasing with the j^{th} conclusion of ξ_i.

Now consider the dependency set $h(\Sigma)$ and the dependency $h(\xi)$ which has premise $P^{h(\xi)} = h(P^{\xi}) = h(\mathbb{D}_1)$. Because ξ_i and $h(\xi_i)$ have the same variables (h only remaps certain constants), it follows that $h \circ \gamma : X \to \mathtt{adom}(h(\mathbb{D}_1))$ is an embedding of $h(P^{\xi_i}) = P^{h(\xi_i)}$ into $h(\mathbb{D}_1)$. Furthermore, $h(\mathbb{D}_{1.j})$ is the database obtained from $h(\mathbb{D}_1)$ by chasing with the the j^{th} conclusion of $h(\xi_i)$. It follows that $h(\mathbb{D}_1) \xrightarrow{(h(\xi_i),h\circ\gamma)} \{h(\mathbb{D}_{1.1}),\ldots,h(\mathbb{D}_{1.k})\}$ is a valid—although possibly spurious[5]—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 $\mathtt{chase}_{h(\Sigma)}(P^{h(\hat{\xi})})$.

Now consider any database $\hat{\mathbb{D}}_p$ at the leaves of $\hat{T}$. We know that $\hat{\mathbb{D}}_p \vDash \hat{\xi}$, which implies that either there exists an embedding $\gamma : C_j^{\xi} \to \hat{\mathbb{D}}_p$ of one of the conclusions of ξ, or $\hat{\mathbb{D}}_p$ contains a contradiction (see page 115). If γ exists, then, by the argument above, $h \circ \gamma$ is an embedding of $h(C_j^{\xi}) = C_j^{h(\xi)}$ into $h(\hat{\mathbb{D}}_p)$. Alternatively, the final chase step leading into $\hat{\mathbb{D}}_p$ may have derived a contradiction, either explicitly (i.e., a dependency of the form $P^{\xi_i} \to \bot$) or implicitly by equating two distinct constants $a, b \in \mathcal{C}$; as long as the chase invocation is sound w.r.t. h, $h(\hat{\mathbb{D}}_p)$ will also contain a contradiction. In either case, $h(\hat{\mathbb{D}}_p) \vDash h(\xi)$. Hence, the chase tree $h(\hat{T})$ proves $h(\Sigma) \vDash_{\text{fin}} h(\xi)$. □

Corollary 4.4.12 *For any shape* $\mathsf{ts} \in \mathsf{TS}_Q$ *and corresponding shape* $\mathsf{ts}' \in \mathsf{TS}_{Q'}$*, if*

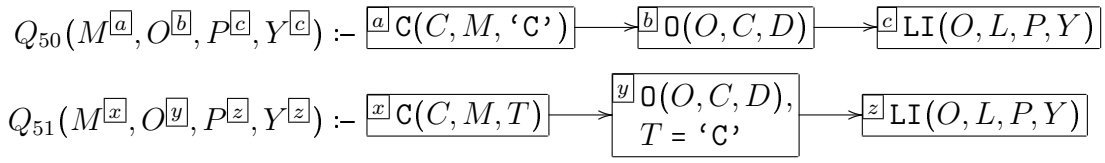
- *all of the equality predicates in* $\mathtt{body}_Q$ *occur within* $G^{\text{min}}_{\mathsf{ts}} \in \mathtt{dags}(\mathtt{body}_Q)$*, and*
- *all of the equality predicates in* $\mathtt{body}_{Q'}$ *occur within* $G^{\text{min}}_{\mathsf{ts}'} \in \mathtt{dags}(\mathtt{body}_{Q'})$*,*

then only the most general canonical tuple of shape ts *needs to be considered when deciding* $Q \equiv Q'$*.*

[5]The chase step $\mathbb{D}_1 \xrightarrow{(\xi_i,\gamma)} \{\mathbb{D}_{1.1},\ldots,\mathbb{D}_{1.k}\}$ is spurious if $\mathbb{D}_1 \vDash \gamma(\xi_i)$. 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).



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 (Corollary 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 problem, our algorithm might construct an instance of an implication problem for which we cannot guarantee chase termination (which does not prove that DACQ equivalence 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 $\mathbb{D} := \mathtt{chase}_\Sigma(P^{\hat{\xi}})$ is infinite, the chase algorithm may be able to conclude $\Sigma \vDash_{\text{fin}} \xi$ based upon a finite sequence of

chase steps yielding some $\hat{\mathbb{D}} \subset \mathbb{D}$. 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 $\Sigma \models_{\text{fin}} \xi$ nor $\Sigma \not\models_{\text{fin}} \xi$ 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 Σ that does not guarantee a terminating chase procedure, the main idea behind constraint relaxation is to replace Σ with a set $\hat{\Sigma}$ that both satisfies $\Sigma \models_{\text{fin}} \hat{\Sigma}$ and guarantees a terminating chase procedure.

Proposition 4.5.1 *Given dependency ξ and set Σ, let $\hat{\Sigma}$ be any dependency set satisfying $\Sigma \models_{\text{fin}} \hat{\Sigma}$. If $\hat{\Sigma} \models_{\text{fin}} \xi$ then $\Sigma \models_{\text{fin}} \xi$.*

The trick is in constructing a useful set $\hat{\Sigma}$ without requiring a chase procedure to explicitly prove $\Sigma \models_{\text{fin}} \hat{\Sigma}$. One technique is to construct $\hat{\Sigma}$ from Σ by identifying dependencies that cause violations of the termination guarantee, and for each such dependency either delete it from Σ or relax it by adding atoms/predicates to the premise and/or deleting atoms/predicates from the conclusion.

Example 52 Consider the following dependency ξ and dependency set $\Sigma = \{\xi_1, \xi_2\}$.

$$\begin{aligned} \xi &: [R(X,Y)] \rightarrow [S(Y,Z)] \\ \xi_1 &: [R(A,B), R(A,C)] \rightarrow [S(B,D), S(C,D)] \\ \xi_2 &: [R(A,B), R(A,C)] \rightarrow [R(B,E)] \end{aligned}$$

Set Σ does not guarantee chase termination, and indeed the database $\texttt{chase}_{\Sigma}([R(x,y)])$ is not finite. If we choose $\hat{\Sigma} = \{\xi_1\}$ then $\Sigma \models_{\text{fin}} \hat{\Sigma}$ is trivial, and $\hat{\Sigma}$ guarantees chase termination (due to *weak acyclicity*—see Section 4.6.4). Then, $\texttt{chase}_{\hat{\Sigma}}([R(x,y)]) = \{R(x,y), S(y,d_1)\}$ proves $\hat{\Sigma} \models_{\text{fin}} \xi$ and therefore $\Sigma \models_{\text{fin}} \xi$.

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 Galido-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 Π_2^p-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 Π_2^p-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 Π_2^p [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 Π_2^p-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 Urpí 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 Π_2^p-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*[6] [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 $\hat{\Sigma}$ be the corresponding set of "flattened"

[6]The condition was derived by 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 Σ 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 Σ, 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 $ABC \twoheadrightarrow v$ with $v \in \mathtt{body}_Q$ and $\{A, B, C\} \subseteq \mathcal{B}_v^{\mathtt{anc}}$, 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, $Q \vDash ABC \twoheadrightarrow v$ if and only if over every database $\mathbb{D}$, the relation $(\mathtt{body}_Q)^{\mathbb{D}}$ (i.e., the result of Q prior to projection onto the head attributes) satisfies the FD $A\boxed{u_A}B\boxed{u_B}C\boxed{u_C}c\boxed{u_1}\cdots c\boxed{u_k} \rightarrow c\boxed{v}$ (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 $X \mapsto Y$ 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 outerjoin 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 that the query result—which is a relation containing NULL (i.e., -) values—always satisfies the strict FD $\overline{\mathcal{I}}_{[1,d]} \rightarrow \overline{\mathcal{V}}$. 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 $\texttt{COCQL}^{\mathcal{F}}$ equivalence, when considering HCEQ encoding-equivalence we will restrict our attention to the subclass of HCEQs that can be output by the 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 $\overline{\S}$-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 $\overline{\mathcal{I}}_{[1,d]} \rightarrow \overline{\mathcal{V}}$. 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 $\texttt{dags}(\texttt{body}_{Q_{52}})$.

$$\begin{aligned} Q_{52}(A^{\boxed{x}}, B^{\boxed{x}}, 0^{\boxed{y}}) \coloneq & \boxed{{}^{\boxed{x}} R(A,B)} \longrightarrow \boxed{{}^{\boxed{y}} R(B,C)} \\ Q_{53}(A,B,0) \coloneq & \ R(A,B) \\ Q_{54}(A,B,0) \coloneq & \ R(A,B), R(B,C) \end{aligned}$$

Over attribute set $\mathcal{W} = \{A, B, 0\}$, the FD $A \rightarrow 0$ and the MVD $A \twoheadrightarrow 0$ are both trivial (because 0 is a constant), and so $Q_{53} \vDash A \rightarrow 0$, $Q_{53} \vDash A \twoheadrightarrow 0$, $Q_{54} \vDash A \rightarrow 0$, and $Q_{54} \vDash A \twoheadrightarrow 0$. In contrast, over labelled attribute set $\mathcal{W} = \{A^{\boxed{x}}, B^{\boxed{x}}, 0^{\boxed{y}}\}$, neither the FD $A^{\boxed{x}} \rightarrow 0^{\boxed{y}}$ nor the MVD $A^{\boxed{x}} \twoheadrightarrow 0^{\boxed{y}}$ is trivial, because the database instance $\mathbb{D}_8 \coloneq \{R(a,a), R(a,b)\}$ proves $Q_{52} \nvDash A^{\boxed{x}} \rightarrow 0^{\boxed{y}}$ and $Q_{52} \nvDash A^{\boxed{x}} \twoheadrightarrow 0^{\boxed{y}}$.

$(Q_{52})^{\mathbb{D}_8}$

$A^{\boxed{x}}$	$B^{\boxed{x}}$	$0^{\boxed{y}}$
a	a	0
a	b	-

$(Q_{53})^{\mathbb{D}_8}$

A	B	0
a	a	0
a	b	0

$(Q_{54})^{\mathbb{D}_8}$

A	B	0
a	a	0

The violation of the FD $A^{\boxed{x}} \rightarrow 0^{\boxed{y}}$ within relation $(Q_{52})^{\mathbb{D}_8}$ 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 $Q_{52} \nvDash A \rightarrow y$. Similarly, the violation of MVD $A^{\boxed{x}} \twoheadrightarrow 0^{\boxed{y}}$ 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 $\overline{\S}$-NF from Chapter 3 to handle HCEQs originating from the ENCODE() procedure. Our definition of core indexes relies upon testing whether a CQ$^{=}$ satisfies a particular MVD. In Section 3.1.1 we considered the comparable problem for CQs, and those results can be extended to CQ$^{=}$s 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 $\overline{\mathcal{W}} \coloneq \overline{\mathcal{I}}_{[1,d]}\overline{\mathcal{V}}$ for the

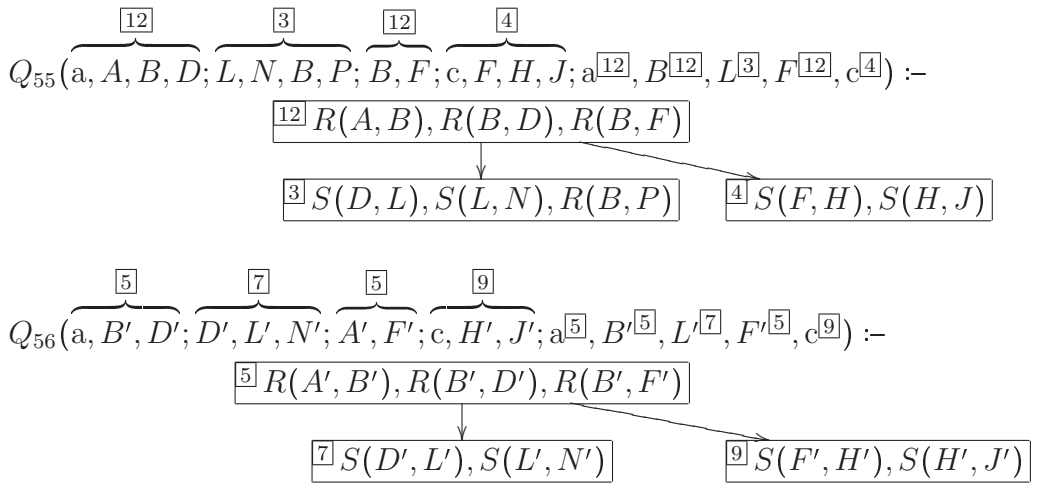


Figure 5.1: Encoding queries Q_{55} = CANONICAL(Q_{20}) and Q_{56} = CANONICAL(Q_{21})

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$^{\{f_b,f_s,f_n\}}$ *query Q' such that Q = CANONICAL(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 ENCODE() (see Section 2.3.2) and CANONICAL() (see Algorithm 4 in Section 4.2.2). Recall from the definition of encoding queries (Definition 2.3.3) that $\overline{\mathcal{I}}_i$ is a sequence of labelled indexes, and that we use $\hat{\mathcal{I}}_i$ and $\mathcal{I}_i$ to denote the corresponding sets of *labelled* and *unlabelled* indexes, respectively.

Observation 5.1.2 *Given any derived DACEQ $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$, 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 $\hat{\mathcal{I}}_i$ is either*

- *(a) a subset of $\{\blacksquare\}$ (i.e., $\{\}$ or $\{\blacksquare\}$), or*
- *(b) a non-empty set of variables and constants* all labelled with the same node $v_i \in \mathtt{body}_Q$. *It is possible that $v_i = v_j$ for $i \neq j$.*

[**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,* $X \in \mathcal{L}_{v_i}$*.*

[**P9**] *For any* $i \in [1,d]$ *such that* $\hat{\mathcal{I}}_i \notin \{\text{-}\}$*, the parameters of* $\mathtt{body}_{v_i}$ *satisfy* $\mathcal{P}_{v_i} \subseteq \mathcal{I}_{[1,i-1]}$*, implying* $Q \vDash \mathcal{I}_{[1,i-1]} \twoheadrightarrow v_i$ *by Lemma 4.2.3.*

[**P10**] *For every node* $v \in \mathtt{body}_Q$*, 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 $\overline{\mathcal{I}}_i$ is either entirely - values or does not include any - value. Property **[P9]** guarantees that for any valuation of index tuples $\overline{\mathcal{I}}_{[1,i-1]}$, the possible valuations of index tuple $\overline{\mathcal{I}}_i$ cannot include both the entirely - tuple and a tuple not containing -. For a concrete example, the reader is invited to refer back to encoding relations $R_8 = (Q_{55})^{\mathbb{D}_3}$ and $R_9 = (Q_{56})^{\mathbb{D}_3}$ 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* $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$ *with nullability set* $\mathcal{S}_Q$*, a length-*d *signature* $\overline{\S}$*, and an integer* $i \in [1,d]$*, for each* $G \in \mathtt{dags}(\mathtt{body}_Q)$ *satisfying* $v_i \in G$*, define set* $\mathcal{W}_{(G,i)} := \mathcal{B}^+_G \cap (\mathcal{I}_{[1,i]} \cup \mathcal{I}^{\overline{\S}}_{[i+1,d]})$*, and let* $Q_{(G,i)}$ *be the following* $CQ^=$*.*

$$Q_{(G,i)}(\mathcal{W}_{(G,i)}) \text{ :- } \mathtt{body}^+_G \tag{5.1}$$

Define the set of core indexes at level i relative to $\overline{\S}$*—denoted* $\hat{\mathcal{I}}^{\overline{\S}}_i$*—as a* minimal *subset of* $\hat{\mathcal{I}}_i$ *such that* $\mathcal{I}^{\overline{\S}}_i$ *(the* unlabelled *version the indexes in* $\hat{\mathcal{I}}^{\overline{\S}}_i$*—see Section 2.3.1) satisfies both* $(\mathcal{V} \cup \mathcal{S}_Q) \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{I}^{\overline{\S}}_{[i,d]} \cup \mathcal{C}$ *and the following signature-specific conditions.*

$\S_\mathtt{i}$	**Condition**
b	$\mathcal{I}_i \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{I}^{\overline{\S}}_i \cup \mathcal{C}$
s	$\forall G$ *containing* v_i : $\left[Q_{(G,i)} \vDash \mathcal{B}^+_G \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}^{\overline{\S}}_i) \twoheadrightarrow \mathcal{B}^+_G \cap \mathcal{I}^{\overline{\S}}_{[i+1,d]}\right]$
n	$\forall G$ *containing* v_i : $\left[Q_{(G,i)} \vDash \mathcal{B}^+_G \cap \mathcal{I}_{[1,i-1]} \twoheadrightarrow \mathcal{B}^+_G \cap \mathcal{I}^{\overline{\S}}_{[i,d]}\right]$

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

Lemma 5.1.4 *Definition 5.1.3 determines a unique minimal set* $\hat{\mathcal{I}}^{\overline{\S}}_i$*.*

Proof (sketch). Identical to the proof of Lemma 3.1.15, except that the MVD arguments for cases $\S_i = \mathtt{s}$ and $\S_i = \mathtt{n}$ need to be repeated for each $Q_{(G,i)}$ (equation 5.1). □

Example 56 Continuing Examples 54 and 55, Query Q_{55} satisfies $\{F\} \twoheadrightarrow 4$ and $\{D\} \twoheadrightarrow 3$, and so $\mathcal{S}_{Q_{55}} = \{D, F\}$. The core indexes of Q_{55} with respect to encoding signature $\overline{\S} = \mathtt{snsb}$ are calculated as follows.

$i = 4$: $\S_4 = \mathtt{b}$, $\hat{\mathcal{I}}_4^{\overline{\S}} = \{H^{\boxed{4}}, J^{\boxed{4}}\}$
$\mathsf{c}^{\boxed{4}} \notin \hat{\mathcal{I}}_4^{\overline{\S}}$ because $\mathsf{c} \in \mathcal{C}$; $F^{\boxed{4}} \notin \hat{\mathcal{I}}_4^{\overline{\S}}$ because $F \in \mathcal{I}_3$; $\{H, J\} \subseteq \mathcal{I}_4^{\overline{\S}}$ in order to satisfy condition $\mathcal{I}_4 = \{\mathsf{c}, F, H, J\} \subseteq \mathcal{I}_{[1,3]} \cup \mathcal{I}_4^{\overline{\S}} \cup \mathcal{C}$.

$i = 3$: $\S_3 = \mathtt{s}$, $\hat{\mathcal{I}}_3^{\overline{\S}} = \{F^{\boxed{12}}\}$
$B^{\boxed{12}} \notin \hat{\mathcal{I}}_3^{\overline{\S}}$ because $B \in \mathcal{I}_2$; $F^{\boxed{12}} \in \hat{\mathcal{I}}_3^{\overline{\S}}$ because $F \in \mathcal{S}_{Q_{55}} \setminus (\mathcal{I}_{[1,2]} \cup \mathcal{I}_4^{\overline{\S}} \cup \mathcal{C})$.

$i = 2$: $\S_2 = \mathtt{n}$, $\mathcal{I}_2^{\overline{\S}} = \{L^{\boxed{3}}, N^{\boxed{3}}\}$
$L^{\boxed{3}} \in \hat{\mathcal{I}}_2^{\overline{\S}}$ because $L \in \mathcal{V} \setminus (\mathcal{I}_1 \cup \mathcal{I}_{[3,4]}^{\overline{\S}} \cup \mathcal{C})$; $B^{\boxed{3}} \notin \hat{\mathcal{I}}_2^{\overline{\S}}$ because $B \in \mathcal{I}_1$; $P^{\boxed{3}} \notin \hat{\mathcal{I}}_2^{\overline{\S}}$ due to the MVDs $Q_{(123,2)} \vDash \mathsf{a}ABD \twoheadrightarrow LNF$ and $Q_{(1234,2)} \vDash \mathsf{a}ABD \twoheadrightarrow LNFHJ$, while $Q_{(123,2)}$ requires $N^{\boxed{3}} \in \hat{\mathcal{I}}_2^{\overline{\S}}$ in order to satisfy the necessary MVD.

$$\begin{aligned} Q_{(123,2)}(\mathsf{a}, A, B, D, L, N, P, F) \coloneq\ & R(A,B), R(B,D), R(B,F), \\ & S(D,L), S(L,N), R(B,P) \\ Q_{(1234,2)}(\mathsf{a}, A, B, D, L, N, P, F, H, J) \coloneq\ & R(A,B), R(B,D), R(B,F), S(D,L), \\ & S(L,N), R(B,P), S(F,H), S(H,J) \end{aligned}$$

$i = 1$: $\S_1 = \mathtt{s}$, $\mathcal{I}_1^{\overline{\S}} = \{B^{\boxed{12}}, D^{\boxed{12}}\}$
$\mathsf{a}^{\boxed{12}} \notin \hat{\mathcal{I}}_1^{\overline{\S}}$ because $\mathsf{a} \in \mathcal{C}$; $B^{\boxed{12}} \in \hat{\mathcal{I}}_1^{\overline{\S}}$ because $B \in \mathcal{V} \setminus (\mathcal{I}_{[2,4]}^{\overline{\S}} \cup \mathcal{C})$; $D^{\boxed{12}} \in \hat{\mathcal{I}}_1^{\overline{\S}}$ because $D \in \mathcal{S}_{Q_{55}} \setminus (\mathcal{I}_{[2,4]}^{\overline{\S}} \cup \mathcal{C})$; $A^{\boxed{12}} \notin \hat{\mathcal{I}}_1^{\overline{\S}}$ due to the MVDs $Q_{(12,1)} \vDash BD \twoheadrightarrow F$, $Q_{(123,1)} \vDash BD \twoheadrightarrow LNF$, $Q_{(124,1)} \vDash BD \twoheadrightarrow FHJ$, and $Q_{(1234,1)} \vDash BD \twoheadrightarrow LNFHJ$.

$$\begin{aligned} Q_{(12,1)}(\mathsf{a}, A, B, D, F) \coloneq\ & R(A,B), R(B,D), R(B,F) \\ Q_{(123,1)}(\mathsf{a}, A, B, D, L, N, F) \coloneq\ & R(A,B), R(B,D), R(B,F), \\ & S(D,L), S(L,N), R(B,P) \\ Q_{(124,1)}(\mathsf{a}, A, B, D, F, H, J) \coloneq\ & R(A,B), R(B,D), R(B,F), \\ & S(F,H), S(H,J) \\ Q_{(1234,1)}(\mathsf{a}, A, B, D, L, N, F, H, J) \coloneq\ & R(A,B), R(B,D), R(B,F), S(D,L), \\ & S(L,N), R(B,P), S(F,H), S(H,J) \end{aligned}$$

Given derived DACEQ $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$, we convert Q to $\overline{\S}$-*normal form ($\overline{\S}$-NF)* by deleting all non-core indexes from the query head. Performing $\overline{\S}$-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 $\overline{\S}$-normalization of derived DACEQs is in Π_2^p.*

Proof. For each level i and each dag $G \in \mathtt{dags}(\mathtt{body}_Q)$, 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 $I \in \hat{\mathcal{I}}_{[1,d]}^{\overline{\S}}$ is in Σ_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

requires I to be core). Hence, verifying the non-core attributes $\mathcal{I}_i \setminus \hat{\mathcal{I}}^{\overline{\S}}_{[1,d]}$ is in co-$\Sigma^p_2 = \Pi^p_2$. □

Conjecture 5.1.6 *Performing $\overline{\S}$-normalization of derived DACEQs is in NP.*

COMMENT. It would suffice to show

$$\begin{array}{c}\left\{\forall v \in \mathtt{body}_Q.\left[Q_{(v,i)} \vDash \mathcal{B}^+_v \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}^{\overline{\S}}_i) \twoheadrightarrow \mathcal{B}^+_v \cap \mathcal{I}^{\overline{\S}}_{[i+1,d]}\right]\right\} \Longrightarrow \\ \left\{\forall G \text{ containing } v_i \ :\left[Q_{(G,i)} \vDash \mathcal{B}^+_G \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}^{\overline{\S}}_i) \twoheadrightarrow \mathcal{B}^+_G \cap \mathcal{I}^{\overline{\S}}_{[i+1,d]}\right]\right\}\end{array} \tag{5.2}$$

and

$$\begin{array}{c}\left\{\forall v \in \mathtt{body}_Q.\left[Q_{(v,i)} \vDash \mathcal{B}^+_v \cap \mathcal{I}_{[1,i-1]} \twoheadrightarrow \mathcal{B}^+_v \cap \mathcal{I}^{\overline{\S}}_{[i,d]}\right]\right\} \Longrightarrow \\ \left\{\forall G \text{ containing } v_i \ :\left[Q_{(G,i)} \vDash \mathcal{B}^+_G \cap \mathcal{I}_{[1,i-1]} \twoheadrightarrow \mathcal{B}^+_G \cap \mathcal{I}^{\overline{\S}}_{[i,d]}\right]\right\}\end{array} \tag{5.3}$$

where $Q_{(v,i)}$ with body $\mathtt{body}^+_v$ 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 3.1.16 would identify a subset of $\hat{\mathcal{I}}_i$ that must be core. $\hat{\mathcal{I}}^{\overline{\S}}_i$ 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 $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$, any index level $i \in [1,d]$, and any node $v \in \mathtt{body}_Q$: if $Q \vDash \mathcal{I}_{[1,i]} \twoheadrightarrow v$ then $Q \vDash \mathcal{I}^{\overline{\S}}_{[1,i]} \twoheadrightarrow v$.*

PROOF. Definition 5.1.3 implies that $(\mathcal{S}_Q \cap \mathcal{I}_{[1,i]}) \subseteq \mathcal{I}^{\overline{\S}}_{[1,i]}$. By Corollary 4.2.7, if $Q \vDash \mathcal{I}_{[1,i]} \twoheadrightarrow v$ then $Q \vDash (\mathcal{S}_Q \cap \mathcal{I}_{[1,i]}) \twoheadrightarrow v$. □

Theorem 5.1.8 *$\overline{\S}$-Normalization preserves $\overline{\S}$-equivalence*

PROOF. For every $i \in [0,d]$, let Q_i denote the following HCEQ,

$$Q_i(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_i; \overline{\mathcal{I}}^{\overline{\S}}_{i+1}; \ldots; \overline{\mathcal{I}}^{\overline{\S}}_d; \overline{\mathcal{V}}) \coloneq \mathtt{body}_Q \tag{5.4}$$

and assume that $\overline{\mathcal{I}}_i = \overline{\mathcal{I}}^{\overline{\S}}_i \cdot \overline{\mathcal{J}}_i$, where $\hat{\mathcal{J}}_i$ is the set of non-core indexes in $\hat{\mathcal{I}}_i$. 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 $\hat{Q}_i$ 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 $\S_i = \mathtt{b}$:

It suffices to show that for any database $\mathbb{D}$, the relation $\Phi \coloneqq (Q_i)^{\mathbb{D}}$ satisfies the FD $\hat{\mathcal{I}}_{[1,i-1]} \to \hat{\mathcal{J}}_i$; the remainder then follows identically to the proof of Theorem 3.1.17. By Definition 3.1.11, either

1. $\mathcal{J}_i \subseteq \{\text{-}\}$, or
2. $\mathcal{J}_i \subseteq \mathcal{I}_{[1,i-1]} \cup \mathcal{C}$.

In the second case, property **[P7]** (Observation 5.1.2) implies that all elements of $\hat{\mathcal{J}}_i$ are labelled with v_i satisfying $Q \vDash \mathcal{I}_{[1,i-1]} \twoheadrightarrow v_i$; therefore, in either case the relation Φ satisfies the FD $\hat{\mathcal{I}}_{[1,i-1]} \rightarrow \hat{\mathcal{J}}_i$.

Case $\S_i = \mathbf{s}$**:**

It suffices to show that relation Φ satisfies the MVD $\hat{\mathcal{I}}_{[1,i-1]} \cup \hat{\mathcal{I}}_i^{\overline{\S}} \twoheadrightarrow \hat{\mathcal{I}}_{[i+1,d]}^{\overline{\S}} \cup \hat{\mathcal{V}}$; the remainder then follows identically to the proof of Theorem 3.1.17.

By property **[P7]**, either $\hat{\mathcal{I}}_i \subseteq \{\text{-}\}$ or all elements of $\hat{\mathcal{I}}_i$ are labelled with v_i. If $\hat{\mathcal{I}}_i \subseteq \{\text{-}\}$ then attributes $\overline{\mathcal{I}}_i$ are constant within relation Φ and the MVD holds trivially; therefore, assume that some v_i labels all elements of $\hat{\mathcal{I}}_i$.

Let $\phi_1 : G_1 \rightsquigarrow \mathbb{D}$ and $\phi_2 : G_2 \rightsquigarrow \mathbb{D}$ be any two terminal embeddings in Φ—of $G_1, G_2 \in \mathtt{dags}(\mathtt{body}_Q)$, respectively—that satisfy $\hat{\phi}_1(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}}) = \hat{\phi}_2(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}})$. (Recall from equation 2.22 that $\hat{\phi}$ denotes the partial application of ϕ.) For each $j \in [1,i]$ such that $\hat{\mathcal{I}}_j \notin \{\text{-}\}$, $v_j \in G_1$ iff $v_j \in G_2$. If $v_i \notin G_1$ then $\hat{\phi}_1(\overline{\mathcal{J}}_i) = \hat{\phi}_2(\overline{\mathcal{J}}_i)$ is a constant tuple composed only of the - 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

$$Q_{(G_3,i)} \vDash \mathcal{B}^+_{G_3} \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^{\overline{\S}}) \twoheadrightarrow \mathcal{B}^+_{G_3} \cap \mathcal{I}^{\overline{\S}}_{[i+1,d]}$$

and ϕ_1, ϕ_2 are embeddings of $\mathtt{body}^+_{G_3}$ into $\mathbb{D}$ that agree on $\mathcal{B}^+_{G_3} \cap (\mathcal{I}_{[1,i-1]} \cap \mathcal{I}_i^{\overline{\S}})$, there exists an embedding $\phi_3 : G_3 \rightarrow \mathbb{D}$ satisfying

- $\hat{\phi}_3(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}}) = \hat{\phi}_2(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}}) = \hat{\phi}_1(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}})$,
- $\hat{\phi}_3(\mathcal{J}_i) = \hat{\phi}_2(\mathcal{J}_i)$ which, by $\mathcal{I}_{[1,i-1]} \twoheadrightarrow v_i$, implies $\hat{\phi}_3(\hat{\mathcal{J}}_i) = \hat{\phi}_2(\hat{\mathcal{J}}_i)$, and
- $\forall j \in [i+1, d]$, if $v_j \notin (G_1 \setminus G_3)$ then $\hat{\phi}_3(\overline{\mathcal{I}}_j^{\overline{\S}}) = \hat{\phi}_1(\overline{\mathcal{I}}_j^{\overline{\S}})$.

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, $Q \vDash \mathcal{I}_{[1,i-1]} \cup \mathcal{I}^{\overline{\S}}_{[i,j-1]} \twoheadrightarrow v_j$. By $\hat{\phi}_3(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}^{\overline{\S}}_{[i,j-1]}) = \hat{\phi}_1(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}^{\overline{\S}}_{[i,j-1]})$ and the definition of nullability dependencies (Definition 4.2.2), ϕ_3 must extend to G_4. We now iterate the above argument, appealing to the following MVD

$$Q_{(G_4,i)} \vDash \mathcal{B}^+_{G_4} \cap (\mathcal{I}_{[1,i-1]} \cup \mathcal{I}_i^{\overline{\S}}) \twoheadrightarrow \mathcal{B}^+_{G_4} \cap \mathcal{I}^{\overline{\S}}_{[i+1,d]}$$

and to ϕ_1, ϕ_3 being embeddings of $\mathtt{body}^+_{G_4}$ into $\mathbb{D}$ agreeing on $\mathcal{B}^+_{G_4} \cap (\mathcal{I}_{[1,i-1]} \cap \mathcal{I}_i^{\overline{\S}})$ to deduce the existence of some embedding $\phi_4 : G_4 \rightarrow \mathbb{D}$. By induction, we eventually arrive at some terminal embedding $\phi_k : G_1 \rightsquigarrow \mathbb{D}$ satisfying

[1] We are abusing notation by using G to refer to both a DAG within $\mathtt{dags}(\mathtt{body}_Q)$ and the set of node labels occurring within that DAG.

- $\hat{\phi}_k(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}}) = \hat{\phi}_2(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}}) = \hat{\phi}_1(\overline{\mathcal{I}}_{[1,i-1]}\overline{\mathcal{I}}_i^{\overline{\S}})$,
- $\hat{\phi}_k(\hat{\mathcal{J}}_i) = \hat{\phi}_2(\hat{\mathcal{J}}_i)$, and
- $\forall j \in [i+1, d]$, $\hat{\phi}_k(\overline{\mathcal{I}}_{[i+1,d]}^{\overline{\S}}) = \hat{\phi}_1(\overline{\mathcal{I}}_{[i+1,d]}^{\overline{\S}})$.

Embedding ϕ_k proves that that relation Φ satisfies the desired MVD.

Case $\S_i = \mathtt{n}$**:**

It suffices to show that relation Φ satisfies the MVD $\hat{\mathcal{I}}_{[1,i-1]} \twoheadrightarrow \hat{\mathcal{I}}_{[i,d]}^{\overline{\S}} \cup \hat{\mathcal{V}}$. The proof is identical to case $\S_i = \mathtt{s}$, 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 $Q_{57} \doteq_{\mathtt{snsb}} Q_{55} \doteq_{\mathtt{snsb}} Q_{20}$. Query Q_{58} in Figure 5.2b is the snsb-normal form of Q_{56} and therefore $Q_{58} \doteq_{\mathtt{snsb}} Q_{56} \doteq_{\mathtt{snsb}} Q_{21}$. Figure 5.2c shows the encoding relation $R_{16} = (Q_{57})^{\mathbb{D}_3}$, which can be obtained by projecting away the non-core index attributes from encoding relation $R_8 = (Q_{20})^{\mathbb{D}_3}$ (cf. Figure 2.20). Figure 5.2d shows the encoding relation $R_{17} = (Q_{58})^{\mathbb{D}_3}$, which can be obtained by projecting away the non-core index attributes from encoding relation $R_9 = (Q_{21})^{\mathbb{D}_3}$ (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 $\overline{\S}$-equivalence between derived DACEQs. We start by deriving a necessary condition that relates $\overline{\S}$-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 $\overline{\S}$-normal forms constitutes a sufficient condition for $\overline{\S}$-equivalence.

Definition 5.2.1 (Structural DACQ) *Given derived DACEQ* $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$*, for each* $i \in [1, d]$ *define the attribute* X_i *as follows.*

$$X_i := \begin{cases} \boxed{\text{-}} & \text{if } \hat{\mathcal{I}}_i \subseteq \{\boxed{\text{-}}\} \\ 0^{\boxed{v_i}} & \text{otherwise } (v_i \text{ defined as in property } [\mathbf{P7}] \text{ of Observation 5.1.2)} \end{cases}$$

We define the structural query $\textsc{Structural}(Q)$ *as the DACQ with the same body as* Q *and with the head tuple* $\overline{\mathcal{W}} := \overline{X}_{[1,d]}\overline{\mathcal{V}}$.

$$Q_{57}(\overbrace{B,D}^{\boxed{12}};\overbrace{L,N}^{\boxed{3}};\overbrace{F}^{\boxed{12}};\overbrace{H,J}^{\boxed{4}};\mathrm{a}^{\boxed{12}},B^{\boxed{12}},L^{\boxed{3}},F^{\boxed{12}},\mathrm{c}^{\boxed{4}}) \coloneq$$
$$\boxed{12}\,R(A,B),R(B,D),R(B,F)$$
$$\boxed{3}\,S(D,L),S(L,N),R(B,P) \qquad \boxed{4}\,S(F,H),S(H,J)$$

(a) Q_{57}, the `snsb`-normal form of Q_{55}

$$Q_{58}(\overbrace{B',D'}^{\boxed{5}};\overbrace{L',N'}^{\boxed{7}};\overbrace{F'}^{\boxed{5}};\overbrace{H',J'}^{\boxed{9}};\mathrm{a}^{\boxed{5}},B'^{\boxed{5}},L'^{\boxed{7}},F'^{\boxed{5}},\mathrm{c}^{\boxed{9}}) \coloneq$$
$$\boxed{5}\,R(A',B'),R(B',D'),R(B',F')$$
$$\boxed{7}\,S(D',L'),S(L',N') \qquad \boxed{9}\,S(F',H'),S(H',J')$$

(b) Q_{58}, the `snsb`-normal form of Q_{56}

R_{16}

$B^{\boxed{12}}$	$D^{\boxed{12}}$	$L^{\boxed{3}}$	$N^{\boxed{3}}$	$F^{\boxed{12}}$	$H^{\boxed{4}}$	$J^{\boxed{4}}$	$\mathrm{a}^{\boxed{12}}$	$B^{\boxed{12}}$	$L^{\boxed{3}}$	$F^{\boxed{12}}$	$\mathrm{c}^{\boxed{4}}$
2	3	4	5	3	4	5	a	2	4	3	c
2	3	4	5	3	4	6	a	2	4	3	c
2	3	4	5	7	-	-	a	2	4	7	-
2	3	4	6	3	4	5	a	2	4	3	c
2	3	4	6	3	4	6	a	2	4	3	c
2	3	4	6	7	-	-	a	2	4	7	-
2	7	-	-	3	4	5	a	2	-	3	c
2	7	-	-	3	4	6	a	2	-	3	c
2	7	-	-	7	-	-	a	2	-	7	-

(c) Query result $R_{16} = (Q_{57})^{\mathbb{D}_3}$ satisfying $\textsc{Decode}(R_{16}, \texttt{snsb}) = o_{13}$

R_{17}

$B'^{\boxed{5}}$	$D'^{\boxed{5}}$	$L'^{\boxed{7}}$	$N'^{\boxed{37}}$	$F'^{\boxed{5}}$	$H'^{\boxed{9}}$	$J'^{\boxed{9}}$	$\mathrm{a}^{\boxed{5}}$	$B'^{\boxed{5}}$	$L'^{\boxed{7}}$	$F'^{\boxed{5}}$	$\mathrm{c}^{\boxed{9}}$
2	3	4	5	3	4	5	a	2	4	3	c
2	3	4	5	3	4	6	a	2	4	3	c
2	3	4	5	7	-	-	a	2	4	7	-
2	3	4	6	3	4	5	a	2	4	3	c
2	3	4	6	3	4	6	a	2	4	3	c
2	3	4	6	7	-	-	a	2	4	7	-
2	7	-	-	3	4	5	a	2	-	3	c
2	7	-	-	3	4	6	a	2	-	3	c
2	7	-	-	7	-	-	a	2	-	7	-

(d) Query result $R_{17} = (Q_{57})^{\mathbb{D}_3}$ satisfying $\textsc{Decode}(R_{17}, \texttt{snsb}) = o_{13}$

Figure 5.2: The `snsb`-normal forms of Q_{55} and Q_{56} and their results over $\mathbb{D}_3$

Theorem 5.2.2 *Given any two derived DACEQs Q and Q' and any encoding signature $\overline{\S}$, if $Q \doteq_{\overline{\S}} Q'$ then* $\textsc{Structural}(Q) \equiv \textsc{Structural}(Q')$.

PROOF. Given any database $\mathbb{D}$, let Γ denote the set of terminal embeddings of $\texttt{body}_Q$ into $\mathbb{D}$, and let Φ be the analogous set for $\texttt{body}_{Q'}$. Suppose that $Q \doteq_{\overline{\S}} Q'$; then, there must exist some $\overline{\S}$-certificate C proving $(Q)^{\mathbb{D}} \doteq_{\overline{\S}} (Q')^{\mathbb{D}}$. Consider any $\gamma \in \Gamma$—then, there must exist some $\phi \in \Phi$ such that certificate C contains a tuple node $n^{\mathtt{t}}$ that proves $\hat{\gamma}(\overline{\mathcal{V}}) = \hat{\phi}(\overline{\mathcal{V}}')$. By Definitions 2.2.5–2.2.7, the certificate node at any level $i \in [1, d]$ along the path to $n^{\mathtt{t}}$ must satisfy $\hat{\gamma}(\overline{\mathcal{I}}_i) = \blacksquare^{|\overline{\mathcal{I}}_i|} \iff \hat{\phi}(\overline{\mathcal{I}}'_i) = \blacksquare^{|\overline{\mathcal{I}}'_i|}$. It follows that $\hat{\gamma}(\overline{X}_{[1,d]}\overline{\mathcal{V}}) = \hat{\phi}(\overline{X}'_{[1,d]}\overline{\mathcal{V}}')$, which proves $\textsc{Structural}(Q) \sqsubseteq \textsc{Structural}(Q')$. Repeating the argument in the other direction proves $\textsc{Structural}(Q') \sqsubseteq \textsc{Structural}(Q)$. □

Corollary 5.2.3 *If $Q \doteq_{\overline{\S}} Q'$ then* $\textsc{Structural}(Q)$ *and* $\textsc{Structural}(Q')$ *are head graph isomorphic.*

PROOF. Theorem 4.3.4. □

Corollary 5.2.4 *Given any derived DACEQ Q and any signature $\overline{\S}$, the $\overline{\S}$-normal form of Q contains within its head the label of every node in $\texttt{body}_Q$.*

PROOF. Property **[P10]** of Observation 5.1.2 implies that the label of every node in $\texttt{body}_Q$ appears within the head graph of $\textsc{Structural}(Q)$, and so the claim follows by Theorem 5.1.8 and Corollary 5.2.3. □

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 to handle CQ$^=$s.

Definition 5.2.5 (CQ$^=$ Homomorphism) *Given two satisfiable CQ$^=$s $Q(\overline{\mathcal{W}})$ and $Q'(\overline{\mathcal{W}}')$, a homomorphism h from Q' to Q is a mapping $h : \mathcal{B}' \to \mathcal{B} \cup \mathcal{C}$ satisfying*

1. $\texttt{body}_Q \vDash h(\texttt{body}_{Q'})$*; in other words,*
   (a) $\texttt{pred}_Q \vDash h(\texttt{pred}_{Q'})$*, and*
   (b) $\forall R(\overline{X}') \in \texttt{atoms}_{Q'}.\exists R(\overline{X}) \in \texttt{atoms}_Q.\Big[\texttt{pred}_Q \vDash h(\overline{X}') = \overline{X}\Big]$*; and*
2. *for each attribute $W'_i \in \overline{\mathcal{W}}'$: $\texttt{pred}_Q \vDash h(W'_i) = W_i$, where W_i is the corresponding attribute in $\overline{\mathcal{W}}$.*

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 $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$ and $Q'(\overline{\mathcal{I}}'_1; \ldots; \overline{\mathcal{I}}'_d; \overline{\mathcal{V}}')$ such that* $\textsc{Structural}(Q')$ *and* $\textsc{Structural}(Q)$

are head graph isomorphic, for each node $u \in \mathtt{body}_Q$ *let* u' *denote the corresponding node in* $\mathtt{body}_{Q'}$*, 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* $u' \in \mathtt{body}_{Q'}$, $\mathtt{body}^+_u \vDash \mathtt{body}_{u'}$*; in other words,*
   (a) $\mathtt{pred}^+_u \vDash h(\mathtt{pred}_{u'})$*, and*
   (b) $\forall R(\overline{X}') \in \mathtt{atoms}_{u'}.\exists R(\overline{X}) \in \mathtt{atoms}^+_u.\left[\mathtt{pred}^+_u \vDash h(\overline{X}') = \overline{X}\right]$*;*
2. *for each attribute* $V_i'^{\boxed{u'}} \in \overline{\mathcal{V}}'$*:* $\mathtt{pred}^+_u \vDash h(V_i') = V_i$*, where* $V_i^{\boxed{u}}$ *is the corresponding attribute in* $\overline{\mathcal{V}}$*; and*
3. $\forall i \in [1,d]$*:* $\hat{\mathcal{I}}_i \subseteq h(\hat{\mathcal{I}}'_i)$*.*

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.

$$h = \{A/A', B/B', D/D', F/F', L/L', N/N', P/F', H/H', J/J'\}$$
$$h' = \{A'/A, B'/B, D'/D, F'/F, L'/L, N'/N, H'/H, J'/J\}$$

Theorem 5.2.7 *Two derived DACEQs are* $\overline{\S}$*-equivalent if there exists index-covering homomorphisms in both directions between their* $\overline{\S}$*-normal forms.*

Proof. Let $Q(\overline{\mathcal{I}}_1; \ldots; \overline{\mathcal{I}}_d; \overline{\mathcal{V}})$ and $Q'(\overline{\mathcal{I}}'_1; \ldots; \overline{\mathcal{I}}'_d; \overline{\mathcal{V}}')$ be the $\overline{\S}$-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 $\overline{\S}$-equivalent iff $Q \doteq_{\overline{\S}} Q'$. The existence of h and h' implies by definition that there exists a head graph isomorphism θ from Q to Q', and by Corollary 5.2.4 θ must be a bijection between the nodes in $\mathtt{body}_Q$ and $\mathtt{body}_{Q'}$.

By Definition 5.2.6, composite mapping $h \circ h'$ satisfies $\forall i \in [1,d].\left[h \circ h'(\hat{\mathcal{I}}_i) = \hat{\mathcal{I}}_i\right]$. Therefore, we can assume w.l.o.g. that $h \circ h'$ satisfies the following property

$$\forall i \in [1,d].\left[h \circ h'(\overline{\mathcal{I}}_i) = \overline{\mathcal{I}}_i\right] \tag{5.5}$$

(by the finite period of any permutation of tuple $\overline{\mathcal{I}}_{[1,d]}$).

Consider any database $\mathbb{D}$ and any terminal embedding $\gamma : G \rightsquigarrow \mathbb{D}$ of any $G \in \mathtt{dags}(\mathtt{body}_Q)$; then, $\gamma \circ h$ is an embedding of $\theta(G) \in \mathtt{dags}(\mathtt{body}_{Q'})$ into $\mathbb{D}$. Definition 5.1.3 implies $\mathcal{S}_Q \subseteq \mathcal{I}_{[1,d]}$, and equation 5.5 implies $\gamma \circ h \circ h'(\overline{\mathcal{I}}_{[1,d]}) = \gamma(\overline{\mathcal{I}}_{[1,d]})$; therefore, the definition of $\mathcal{S}_Q$ (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 $(Q)^{\mathbb{D}}$ and $(Q')^{\mathbb{D}}$ differ only by permutation of attributes within each index level, and so it is trivial to construct a $\overline{\S}$-certificate proving $(Q)^{\mathbb{D}} \doteq_{\overline{\S}} (Q')^{\mathbb{D}}$. By our unrestricted choice of $\mathbb{D}$, $Q \doteq_{\overline{\S}} Q'$ follows. □

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 $Q_{57} \doteq_{\texttt{snsb}} Q_{58}$. We conclude that $Q_{20} \doteq_{\texttt{snsb}} Q_{21}$ (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 $\overline{\S}$-normal forms—is not necessary for $\overline{\S}$-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 $\overline{\S}$-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 sss-normalization of Q_{59} (using Definition 5.1.3).

$i = 3$: $\S_3 = \mathtt{s}$, $\hat{\mathcal{I}}_3^{\overline{\S}} = \{\}$
$C^{\boxed{v}} \notin \hat{\mathcal{I}}_3^{\overline{\S}}$ because $C \in \mathcal{I}_2$.

$i = 2$: $\S_2 = \mathtt{s}$, $\hat{\mathcal{I}}_2^{\overline{\S}} = \{A^{\boxed{u}}, C^{\boxed{u}}\}$
$C^{\boxed{u}} \in \hat{\mathcal{I}}_2^{\overline{\S}}$ because $C \in \mathcal{V} \setminus (\mathcal{I}_1 \cup \mathcal{I}_3^{\overline{\S}} \cup \mathcal{C})$; meanwhile, $A^{\boxed{u}} \in \hat{\mathcal{I}}_2^{\overline{\S}}$ because $Q_{(u,2)} \vDash BA \twoheadrightarrow C$ and $Q_{(uv,2)} \vDash BA \twoheadrightarrow C$ (both trivially), but $Q_{(u,2)} \nvDash B \twoheadrightarrow C$.

$$Q_{(u,2)}(B, A, C) \coloneq R(A, B), S(B, C), R(A, D), S(D, C), T(D)$$
$$Q_{(uv,2)}(B, A, C) \coloneq R(A, B), S(B, C), R(A, D), S(D, C), T(D), T(B)$$

$i = 1$: $\S_1 = \mathtt{s}$, $\hat{\mathcal{I}}_1^{\overline{\S}} = \{B^{\boxed{u}}\}$
$B^{\boxed{u}} \in \hat{\mathcal{I}}_1^{\overline{\S}}$ because $\mathcal{S}_{Q_{59}} = \{B\}$.

Therefore, Q_{60} is the sss-NF of Q_{59}. However, we can show that $Q_{60} \doteq_{\texttt{sss}} Q_{61}$, implying that A should not need to be a core index of Q_{59} to preserve ss-equivalence.

[2] In general, the existence of index-covering homomorphisms in both directions only implies only that query results are identical *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.

Suppose that we replace the attribute $C^{\boxed{v}}$ with the constant $0^{\boxed{v}}$ within the head of Q_{59}. Within this new query, variable C would not be an output variable, and so the trivial MVDs $Q_{(u,2)} \vDash B \twoheadrightarrow \varnothing$ and $Q_{(uv,2)} \vDash B \twoheadrightarrow \varnothing$ would justify $\hat{\mathcal{I}}_2^{\overline{\S}} = \{\}$ (i.e., neither $A^{\boxed{u}}$ nor $C^{\boxed{u}}$ would be core indexes). Therefore, the assessment that both $A^{\boxed{u}}$ and $C^{\boxed{u}}$ are core within Q_{59} (as per Definition 5.1.3) is directly attributable to head attribute $C^{\boxed{v}}$.

Now reconsider CQ $Q_{(uv,2)}$ shown above; $Q_{(uv,2)}$ 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 $Q_{(uv,2)}$ satisfies the MVD $B \twoheadrightarrow C$. In contrast, h is not an automorphism for $Q_{(u,2)}$, and $Q_{(u,2)} \nvDash B \twoheadrightarrow C$. As a concrete example, consider the the database instance $\mathbb{D}_9$ shown in Figure 5.3b and the set of terminal embeddings $\Gamma = \{\gamma : Q_{59} \rightsquigarrow \mathbb{D}_9\}$ shown in Figure 5.3c. If we project "relation" Γ onto attributes ABC, it violates the MVD $B \twoheadrightarrow C$; however, if we first restrict Γ to only embeddings that extend to $\boxed{v}$, then MVD $B \twoheadrightarrow C$ is satisfied.

In summary, index $C^{\boxed{u}}$ needs to be core in order to functionally determine the value of the output attribute $C^{\boxed{v}}$, but *only for embeddings that extend to node* $\boxed{v}$. Index $A^{\boxed{u}}$ was deemed core because in general it is not separated from core index $C^{\boxed{u}}$ by an MVD, but the subset of embeddings that extend to node $\boxed{v}$—i.e., the embeddings that require $C^{\boxed{u}}$ to be core—satisfy an MVD separating $A^{\boxed{u}}$ from $C^{\boxed{u}}$ (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 $\{\langle 0 \rangle, \langle \blacksquare \rangle\}$. The tuple-specific equivalence $Q_{62} \overset{\langle 0 \rangle}{\equiv} Q_{63}$ 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 $Q_{62} \overset{\langle \blacksquare \rangle}{\equiv} Q_{63}$ requires proving $\Sigma^{(Q_{62}, \langle \blacksquare \rangle)} \equiv_{\text{fin}} \Sigma^{(Q_{63}, \langle \blacksquare \rangle)}$, where $\Sigma^{(Q_{62}, \langle \blacksquare \rangle)}$ and

$$
\begin{array}{lll}
Q_{59}(& \overbrace{B^{\boxed{u}}}^{\overline{\mathcal{I}}_1}\ ; & \overbrace{A^{\boxed{u}}, C^{\boxed{u}}}^{\overline{\mathcal{I}}_2};\ \overbrace{C^{\boxed{v}}}^{\overline{\mathcal{I}}_3};\ \overbrace{C^{\boxed{v}}}^{\overline{\mathcal{V}}}\ ) :- \\
Q_{60}(& B^{\boxed{u}}\ ; & A^{\boxed{u}}, C^{\boxed{u}};\ ;\ C^{\boxed{v}}\ ) :- \\
Q_{61}(& B^{\boxed{u}}\ ; & C^{\boxed{u}}\ ;\ ;\ C^{\boxed{v}}\ ) :-
\end{array}
\quad \boxed{u}\, R(A,B), S(B,C), R(A,D), S(D,C), T(D) \longrightarrow \boxed{v}\, T(B)
$$

(a) HCEQs Q_{59}, Q_{60}, and Q_{61}

R

$\mathtt{col}_1$	$\mathtt{col}_2$
a_1	b
a_1	d_1
a_2	b
a_2	d_2

S

$\mathtt{col}_1$	$\mathtt{col}_2$
b	c_1
d_1	c_1
b	c_2
d_2	c_2

T

$\mathtt{col}_1$
d_1
d_2

(b) Database $\mathbb{D}_9$

Γ	A	B	C	D	$B^{\boxed{u}}$	$A^{\boxed{u}}$	$C^{\boxed{u}}$	$C^{\boxed{v}}$
γ_1	a_1	b	c_1	d_1	b	a_1	c_1	-
γ_2	a_1	d_1	c_1	d_1	d_1	a_1	c_1	c_1
γ_3	a_2	b	c_2	d_2	b	a_2	c_2	-
γ_4	a_2	d_2	c_2	d_2	d_2	a_2	c_2	c_2

(c) Terminal embeddings of Q_{59} into $\mathbb{D}_9$

Figure 5.3: Encoding queries Q_{59}, Q_{60}, and Q_{61} evaluated over database $\mathbb{D}_9$.

$$
Q_{62}(0^{\boxed{v}}) :- \boxed{u}\, R(W,X), R(Y,Z) \longrightarrow \boxed{v}\, S(X,X), S(X_1,X_1), \ldots, S(X_n,X_n), R(X,X_1), R(X_1,X_2), \ldots, R(X_{n-1},X_n), S(Z,X), S(Z_1,X_1), \ldots, S(Z_n,X_n), R(Z,Z_1), R(Z_1,Z_2), \ldots, R(Z_{n-1},Z_n), S(Z,Z), S(Z_1,Z_1), \ldots, S(Z_n,Z_n)
$$

$$
Q_{63}(0^{\boxed{2}}) :- \boxed{1}\, R(A,B), R(C,D), R(E,F) \longrightarrow \boxed{2}\, S(B,B), S(B_1,B_1), \ldots, S(B_n,B_n), R(B,B_1), R(B_1,B_2), \ldots, R(B_{n-1},B_n), S(D,F)
$$

Figure 5.4: HCQs Q_{62} and Q_{63}

$\Sigma^{(Q_{62},\langle\blacksquare\rangle)}$ are defined as follows.

$$\Sigma^{(Q_{62},\langle\blacksquare\rangle)} = \{\xi^{(u,\langle\blacksquare\rangle)} : \mathtt{body}_u \to \mathtt{body}_v\}$$
$$\Sigma^{(Q_{63},\langle\blacksquare\rangle)} = \{\xi^{(a,\langle\blacksquare\rangle)} : \mathtt{body}_1 \to \mathtt{body}_2\}$$

The implication $\Sigma^{(Q_{62},\langle\blacksquare\rangle)} \models_{\text{fin}} \Sigma^{(Q_{63},\langle\blacksquare\rangle)}$ can be proven by the chase procedure using two chase steps:

1. chase $\xi^{(u,\langle\blacksquare\rangle)}$ 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 $\xi^{(u,\langle\blacksquare\rangle)}$ with binding $X/f, Z/d$ to generate the atom $S(d,f)$.

The implication $\Sigma^{(Q_{63},\langle\blacksquare\rangle)} \models_{\text{fin}} \Sigma^{(Q_{62},\langle\blacksquare\rangle)}$ can be proven by the chase procedure using $n+2$ chase steps:

1. chase $\xi^{(1,\langle\blacksquare\rangle)}$ 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 $\xi^{(1,\langle\blacksquare\rangle)}$ 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 $\xi^{(1,\langle\blacksquare\rangle)}$ 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}.

$$Q_{64}(\overbrace{W^{\boxed{u}}, X^{\boxed{u}}, Y^{\boxed{u}}, Z^{\boxed{u}}}^{\overline{\mathcal{I}}_1}; \overbrace{X_1^{\boxed{v}}, \ldots, X_n^{\boxed{v}}, Z_1^{\boxed{v}}, \ldots, Z_n^{\boxed{v}}}^{\overline{\mathcal{I}}_2}; \overbrace{0^{\boxed{v}}}^{\overline{\mathcal{V}}}) \coloneq \mathtt{body}_{Q_{62}}$$

Theorem 4.2.4 implies that $\mathcal{S}_{Q_{64}} = \{X, Z\}$ is the minimal set satisfying $Q_{64} \vDash \mathcal{S}_{Q_{64}} \twoheadrightarrow v$, and so $\mathtt{ss}$-normalization of Q_{64} yields the following query Q_{65}.

$$Q_{65}(X^{\boxed{u}}, Z^{\boxed{u}}; ; 0^{\boxed{v}}) \coloneq \mathtt{body}_{Q_{62}}$$

It straightforward to verify that over any database $\mathbb{D}$, the $\mathtt{ss}$-decoding of $(Q_{65})^{\mathbb{D}}$ yields one of four possible values:

1. $\{\{0\}, \{\blacksquare\}\}$ if and only if $(Q_{62})^{\mathbb{D}} = \{\langle\, 0\,\rangle, \langle\,\blacksquare\,\rangle\}$,
2. $\{\{0\}\}$ if and only if $(Q_{62})^{\mathbb{D}} = \{\langle\, 0\,\rangle\}$,
3. $\{\{\blacksquare\}\}$ if and only if $(Q_{62})^{\mathbb{D}} = \{\langle\,\blacksquare\,\rangle\}$, or
4. $\{\}$ if and only if $(Q_{62})^{\mathbb{D}} = \{\}$.

Repeating the above argument for Q_{63}, we define Q_{66}

$$Q_{66}(\overbrace{A^{\boxed{1}}, B^{\boxed{1}}, C^{\boxed{1}}, D^{\boxed{1}}, E^{\boxed{1}}, F^{\boxed{1}}}^{\overline{\mathcal{I}}_1}; \overbrace{B_1^{\boxed{2}}, \ldots, B_n^{\boxed{2}}}^{\overline{\mathcal{I}}_2}; \overbrace{0^{\boxed{2}}}^{\overline{\mathcal{V}}}) \coloneq \mathtt{body}_{Q_{63}}$$

which has $\mathcal{S}_{Q_{66}} = \{B, D, F\}$ and $\mathtt{ss}$-normal form Q_{67}.

$$Q_{67}(B^{\boxed{1}}, D^{\boxed{1}}, F^{\boxed{1}}; ; 0^{\boxed{2}}) \coloneq \mathtt{body}_{Q_{63}}$$

Due to correspondence between the outputs of Q_{67} and Q_{63}, we conclude the following.

$$Q_{65} \doteq_{\mathtt{ss}} Q_{67} \iff Q_{62} \equiv Q_{63}$$

We have already shown that $Q_{62} \equiv Q_{63}$, and therefore $Q_{65} \doteq_{\mathtt{ss}} Q_{67}$. 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 ss-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$^{\{f_b, f_s, f_n\}}$ 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 τ_{14} of Q_{68} and Q_{69}, as well as its transformation to the chain sort $(\texttt{bnbnb}, 6)$. Figure 6.4 illustrates the CEQs $Q_{70} := \textsc{Encode}(Q_{68})$ and

$Q_{71} := \text{ENCODE}(Q_{69})$. (Components of the queries have been labelled for the sake of clarity.) Converting query Q_{70} to bnbnb-NF removes the shaded indexes from $\overline{\mathcal{I}}_4$ and $\overline{\mathcal{I}}_2$. Query Q_{71} is already in 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.[1]

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 Σ of schema constraints (denoted $Q \equiv^{\Sigma} Q'$) to CEQ encoding-equivalence. Assuming that Σ contains only functional dependencies (FDs) and acyclic inclusion dependencies (INDs)—which generalize primary keys and acyclic foreign keys—we can decide *encoding-equivalence w.r.t.* Σ as follows:

1. Prior to the conversion to $\overline{\S}$-normal form, we pre-process each CEQ by first applying the standard Σ-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 Σ to expand out the index sets in the query head. For each $i \in [1, d]$ starting with $i = 1$, we expand $\overline{\mathcal{I}}_i$ with all variables functionally determined by $\overline{\mathcal{I}}_{[1,i]}$, deleting variables from inner index sets whenever they are added to outer index sets.
3. The conversion to $\overline{\S}$-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 Σ. (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 Σ denote the set of primary and foreign key constraints shown in Figure 1.1. Chasing the body of Q_{70} with Σ 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 $Q_{72} := \texttt{chase}_{\Sigma}(Q_{70})$ shown in Figure 6.4. Shaded attributes again indicate redundant index columns that get removed by bnbnb-normalization. Query Q_{71} is unchanged after chasing with Σ.

[1] Cohen et al. prove that queries performing top-level application of SUM are equivalent if and only if the relational inputs to the SUM function are always equivalent under bag semantics [25]. Grumbach et al. prove a similar result relating AVG to normalized bags [50].

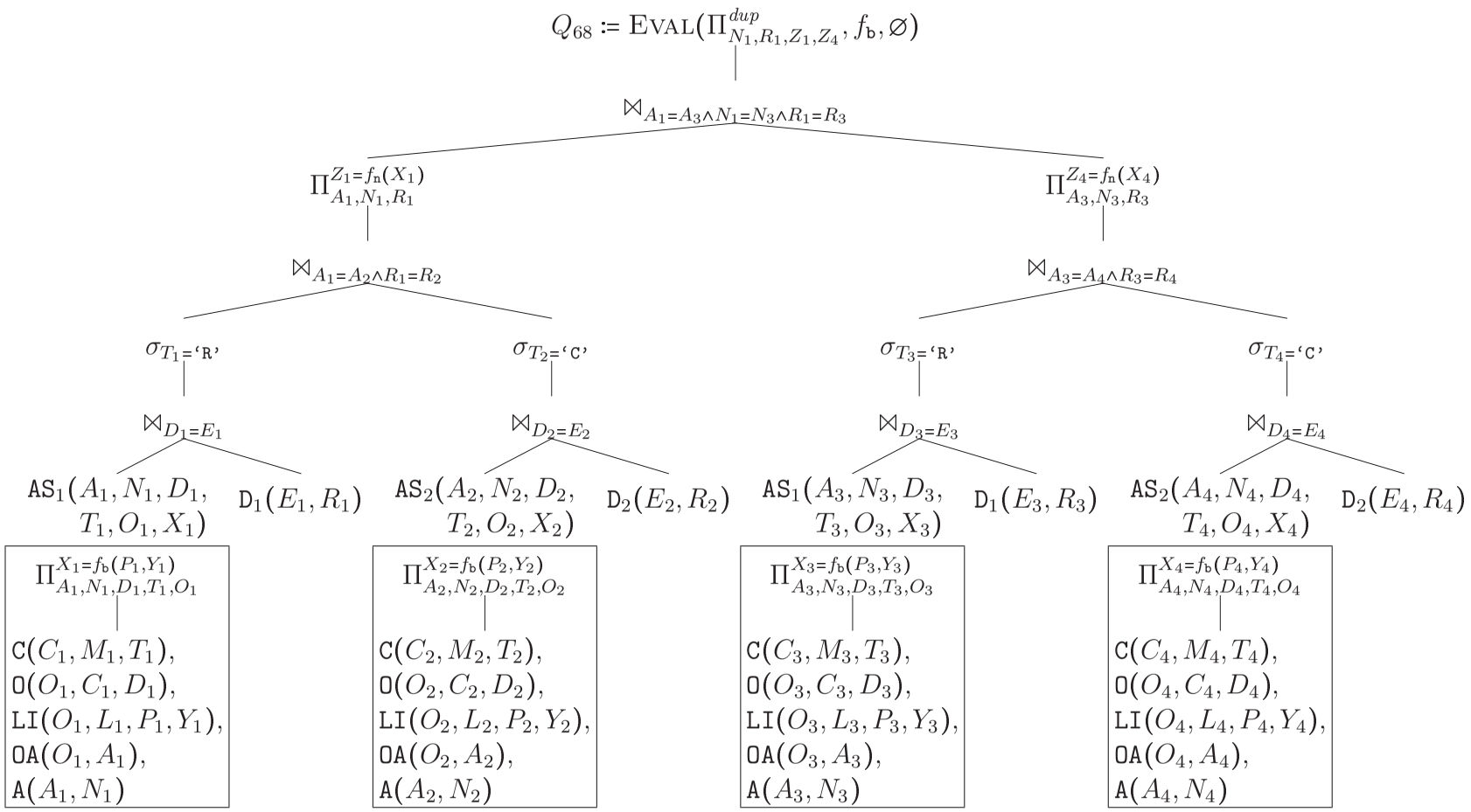


Figure 6.1: Q_{68}, the $\mathtt{COCQL}^{\{f_{\mathtt{b}}, f_{\mathtt{s}}, f_{\mathtt{n}}\}}$ translation of $\mathtt{SQL}$ query Q_1

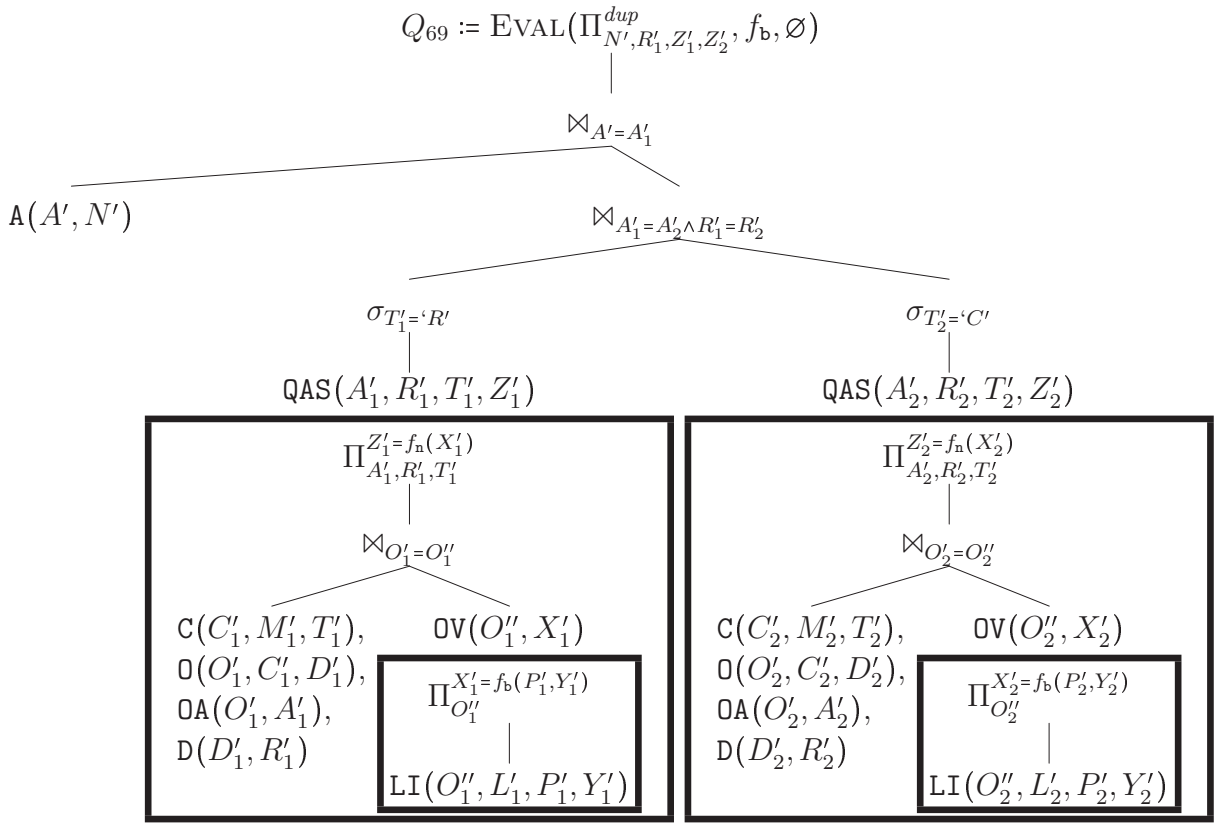


Figure 6.2: Q_{69}, the $\mathtt{COCQL}^{\{f_{\mathtt{b}}, f_{\mathtt{s}}, f_{\mathtt{n}}\}}$ translation of $\mathtt{SQL}$ query Q_3

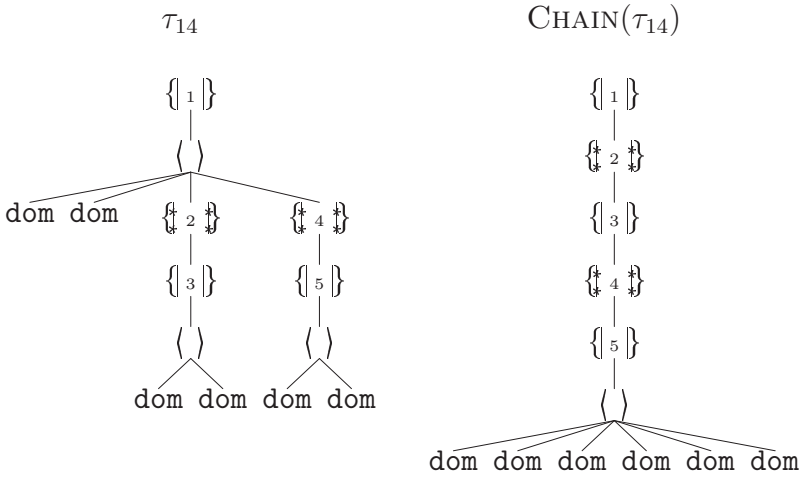


Figure 6.3: Output sort τ_{14} of Q_{68} and Q_{69}, and its transformation to a chain

$$Q_{70}(\overbrace{A, N, R}^{\overline{\mathcal{I}}_1}; \overbrace{D_1, O_1, N_2, D_2, O_2}^{\overline{\mathcal{I}}_2}; \overbrace{C_1, M_1, L_1, P_1, Y_1}^{\overline{\mathcal{I}}_3}; \overbrace{D_3, O_3, N_4}^{\overline{\mathcal{I}}_4}, D_4, O_4; \overbrace{C_4, M_4, L_4, P_4, Y_4}^{\overline{\mathcal{I}}_5}; \overbrace{N, R, P_1, Y_1, P_4, Y_4}^{\overline{\mathcal{V}}}) \coloneq$$

$$\begin{array}{lll}
\texttt{C}(C_1, M_1, \texttt{`R'}),\ \texttt{O}(O_1, C_1, D_1),\ \texttt{LI}(O_1, L_1, P_1, Y_1),\ \texttt{OA}(O_1, A),\ \texttt{A}(A, N),\ \texttt{D}(D_1, R), & \}(\texttt{AS}_1 \bowtie \texttt{D}_1) & \texttt{avgRsale} \\
\texttt{C}(C_2, M_2, \texttt{`C'}),\ \texttt{O}(O_2, C_2, D_2),\ \texttt{LI}(O_2, L_2, P_2, Y_2),\ \texttt{OA}(O_2, A),\ \texttt{A}(A, N_2),\ \texttt{D}(D_2, R), & \}(\texttt{AS}_2 \bowtie \texttt{D}_2) & \texttt{avgRsale} \\
\texttt{C}(C_3, M_3, \texttt{`R'}),\ \texttt{O}(O_3, C_3, D_3),\ \texttt{LI}(O_3, L_3, P_3, Y_3),\ \texttt{OA}(O_3, A),\ \texttt{A}(A, N),\ \texttt{D}(D_3, R), & \}(\texttt{AS}_1 \bowtie \texttt{D}_1) & \texttt{avgCsale} \\
\texttt{C}(C_4, M_4, \texttt{`C'}),\ \texttt{O}(O_4, C_4, D_4),\ \texttt{LI}(O_4, L_4, P_4, Y_4),\ \texttt{OA}(O_4, A),\ \texttt{A}(A, N_4),\ \texttt{D}(D_4, R) & \}(\texttt{AS}_2 \bowtie \texttt{D}_2) & \texttt{avgCsale}
\end{array}$$

$$Q_{71}(\overbrace{A', N', R'}^{\overline{\mathcal{I}}'_1}; \overbrace{D'_1, O'_1, C'_1, M'_1}^{\overline{\mathcal{I}}'_2}; \overbrace{L'_1, P'_1, Y'_1}^{\overline{\mathcal{I}}'_3}; \overbrace{D'_2, O'_2, C'_2, M'_2}^{\overline{\mathcal{I}}'_4}; \overbrace{L'_2, P'_2, Y'_2}^{\overline{\mathcal{I}}'_5}; \overbrace{N', R', P'_1, Y'_1, P'_2, Y'_2}^{\overline{\mathcal{V}}'}) \coloneq$$

$$\begin{array}{lll}
\texttt{C}(C'_1, M'_1, \texttt{`R'}),\ \texttt{O}(O'_1, C'_1, D'_1),\ \texttt{LI}(O'_1, L'_1, P'_1, Y'_1),\ \texttt{OA}(O'_1, A'),\ \texttt{D}(D'_1, R'),\ \texttt{A}(A', N'), & \}\texttt{QAS}_1 \bowtie \texttt{A} & \\
\texttt{C}(C'_2, M'_2, \texttt{`C'}),\ \texttt{O}(O'_2, C'_2, D'_2),\ \texttt{LI}(O'_2, L'_2, P'_2, Y'_2),\ \texttt{OA}(O'_2, A'),\ \texttt{D}(D'_2, R') & \}\texttt{QAS}_2 &
\end{array}$$

$$Q_{72}(\overbrace{A, N, R}^{\overline{\mathcal{I}}_1}; \overbrace{D_1, O_1, C_1, M_1, D_2, O_2, C_2, M_2}^{\overline{\mathcal{I}}_2}; \overbrace{L_1, P_1, Y_1}^{\overline{\mathcal{I}}_3}; \overbrace{D_3, O_3, C_3, M_3}^{\overline{\mathcal{I}}_4}, D_4, O_4, C_4, M_4; \overbrace{L_4, P_4, Y_4}^{\overline{\mathcal{I}}_5}; \overbrace{N, R, P_1, Y_1, P_4, Y_4}^{\overline{\mathcal{V}}}) \coloneq$$

$$\begin{array}{lll}
\texttt{C}(C_1, M_1, \texttt{`R'}),\ \texttt{O}(O_1, C_1, D_1),\ \texttt{LI}(O_1, L_1, P_1, Y_1),\ \texttt{OA}(O_1, A),\ \texttt{A}(A, N),\ \texttt{D}(D_1, R), & \}(\texttt{AS}_1 \bowtie \texttt{D}_1) & \texttt{avgRsale} \\
\texttt{C}(C_2, M_2, \texttt{`C'}),\ \texttt{O}(O_2, C_2, D_2),\ \texttt{LI}(O_2, L_2, P_2, Y_2),\ \texttt{OA}(O_2, A),\ \texttt{A}(A, N),\ \texttt{D}(D_2, R), & \}(\texttt{AS}_2 \bowtie \texttt{D}_2) & \texttt{avgRsale} \\
\texttt{C}(C_3, M_3, \texttt{`R'}),\ \texttt{O}(O_3, C_3, D_3),\ \texttt{LI}(O_3, L_3, P_3, Y_3),\ \texttt{OA}(O_3, A),\ \texttt{A}(A, N),\ \texttt{D}(D_3, R), & \}(\texttt{AS}_1 \bowtie \texttt{D}_1) & \texttt{avgCsale} \\
\texttt{C}(C_4, M_4, \texttt{`C'}),\ \texttt{O}(O_4, C_4, D_4),\ \texttt{LI}(O_4, L_4, P_4, Y_4),\ \texttt{OA}(O_4, A),\ \texttt{A}(A, N),\ \texttt{D}(D_4, R) & \}(\texttt{AS}_2 \bowtie \texttt{D}_2) & \texttt{avgCsale}
\end{array}$$

Figure 6.4: Queries $Q_{70} \coloneq \textsc{Encode}(Q_1)$, $Q_{71} \coloneq \textsc{Encode}(Q_3)$, and $Q_{72} \coloneq \texttt{chase}_{\Sigma}(Q_{70})$

The following two mappings $h : Q_{72} \to Q_{71}$ and $h' : Q_{71} \to Q_{72}$ are index-covering homomorphisms between the bnbnb-normal forms of Q_{72} and Q_{71}.

$$\begin{aligned}
h \coloneqq \{&A/A', N/N', R/R', \\
&C_1/C_1', M_1/M_1', O_1/O_1', D_1/D_1', L_1/L_1', P_1/P_1', Y_1/Y_1', \\
&C_2/C_2', M_2/M_2', O_2/O_2', D_2/D_2', L_2/L_2', P_2/P_2', Y_2/Y_2', \\
&C_3/C_1', M_3/M_1', O_3/O_1', D_3/D_1', L_3/L_1', P_3/P_1', Y_3/Y_1', \\
&C_4/C_2', M_4/M_2', O_4/O_2', D_4/D_2', L_4/L_2', P_4/P_2', Y_4/Y_2' \ \} \\
h' \coloneqq \{&A'/A, N'/N, R'/R, \\
&C_1'/C_1, M_1'/M_1, O_1'/O_1, D_1'/D_1, L_1'/L_1, P_1'/P_1, Y_1'/Y_1, \\
&C_2'/C_4, M_2'/M_4, O_2'/O_4, D_2'/D_4, L_2'/L_4, P_2'/P_4, Y_2'/Y_4 \ \}
\end{aligned}$$

Homomorphisms h and h' together prove $Q_{72} \doteq^{\Sigma}_{\texttt{bnbnb}} Q_{71}$. Because $Q_{72} \coloneqq \texttt{chase}_{\Sigma}(Q_{70})$ implies $Q_{72} \doteq^{\Sigma}_{\overline{\S}} Q_{70}$ for any compatible signature $\overline{\S}$, this entails $Q_{70} \doteq^{\Sigma}_{\texttt{bnbnb}} Q_{71}$ and therefore $Q_{68} \equiv^{\Sigma} Q_{69}$. It follows that the SQL queries Q_1 and Q_3 are equivalent over all database instances conforming to the schema constraints in Σ.

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 MAX as a set, and the output of COUNT(*) as a bag of zero-ary tuples, then Q_4 and Q_6 translate into the $\texttt{COCQL}^{\{f_\texttt{b}, f_\texttt{s}, f_\texttt{n}\}}$ 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 $\{\!|\, \langle\, \texttt{dom}, \{\, \{\!|\, \texttt{dom} \,|\!\}\, \}\, \rangle \,|\!\}$, which we linearize into the chain sort $(\texttt{bsb}, 1)$.

The encoding query $Q_{75} \coloneqq \textsc{Encode}(Q_{73})$ 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 bsb-normal form we must first convert its body into DACQ canonical form (see Chapter 4). Because there exists a homomorphism $h_b : \texttt{body}^+_b \to \texttt{body}^+_a$ that is the identity over $\mathcal{B}^+_a$,

$$h_b = \{C/C, R/R, O_1/O_1, D_1/D_1, M_1/M_1, O_2/O_1, D_2/D_1\}$$

the conversion to DACQ canonical form merges node $\boxed{b}$ into $\boxed{a}$. 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 bsb-NF.

Figure 6.6 also shows encoding query $Q_{77} \coloneqq \textsc{Encode}(Q_{74})$; 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 bsb-NF.

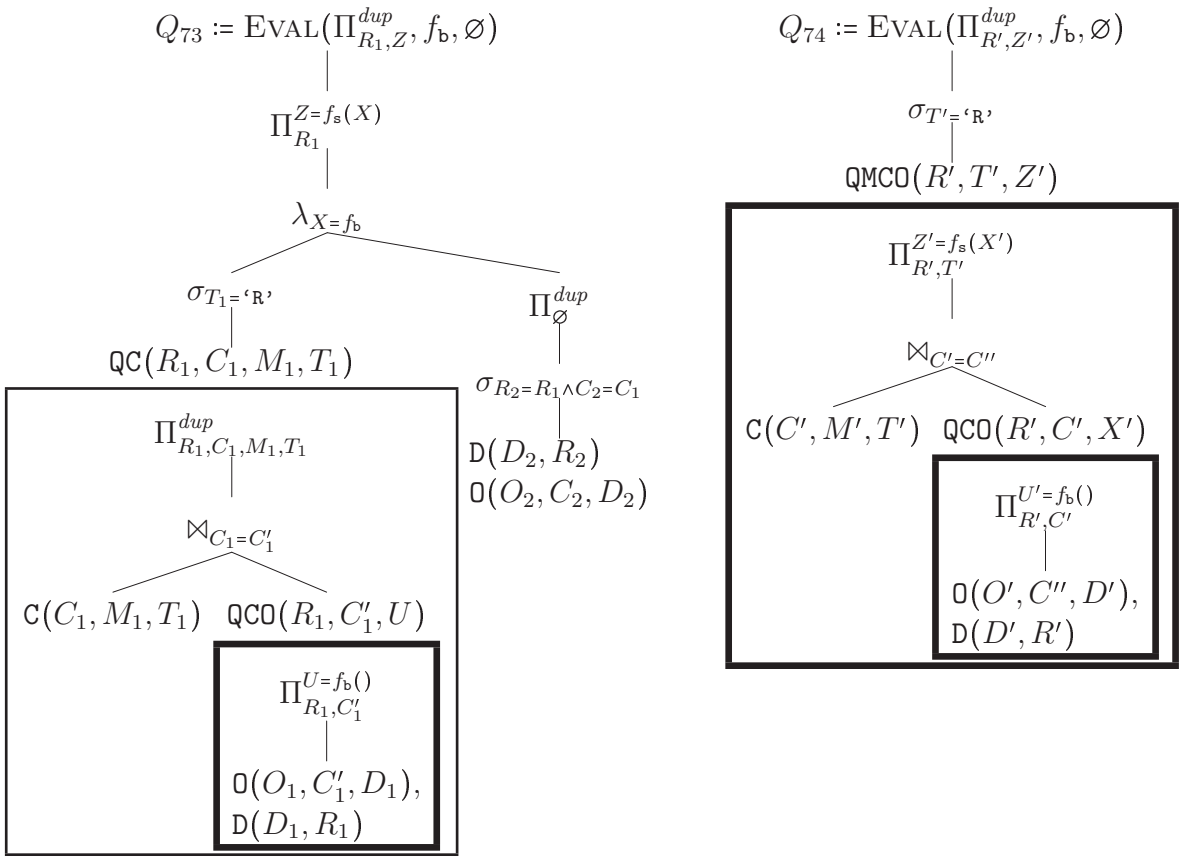


Figure 6.5: Q_{73} and Q_{74}, the $\mathtt{COCQL}^{\{f_{\mathtt{b}}, f_{\mathtt{s}}, f_{\mathtt{n}}\}}$ translations of $\mathtt{SQL}$ queries Q_4 and Q_6

$$Q_{75}(\overbrace{R^{[a]}}^{\overline{\mathcal{I}}_1}; \overbrace{C^{[a]}, M_1^{[a]}, \text{`R'}^{[a]}}^{\overline{\mathcal{I}}_2}; \overbrace{D_2^{[b]}, R^{[b]}, O_2^{[b]}, C^{[b]}}^{\overline{\mathcal{I}}_3}; \overbrace{R^{[a]}}^{\overline{\mathcal{V}}}) \coloneq \boxed{{}^{[a]}\mathtt{O}(O_1, C, D_1), \mathtt{D}(D_1, R), \mathtt{C}(C, M_1, \text{`R'})} \downarrow \boxed{{}^{[b]}\mathtt{O}(O_2, C, D_2), \mathtt{D}(D_2, R)}$$

$$Q_{76}(\overbrace{R}^{\overline{\mathcal{I}}_1}; \overbrace{C, M_1, \text{`R'}}^{\overline{\mathcal{I}}_2}; \overbrace{D_2, R, O_2, C}^{\overline{\mathcal{I}}_3}; \overbrace{R}^{\overline{\mathcal{V}}}) \coloneq \mathtt{O}(O_1, C, D_1), \mathtt{D}(D_1, R), \mathtt{C}(C, M_1, \text{`R'}), \mathtt{O}(O_2, C, D_2), \mathtt{D}(D_2, R)$$

$$Q_{77}(\overbrace{R', \text{`R'}}^{\overline{\mathcal{I}}'_1}; \overbrace{C', M'}^{\overline{\mathcal{I}}'_2}; \overbrace{D', O'}^{\overline{\mathcal{I}}'_3}; \overbrace{R'}^{\overline{\mathcal{V}}'}) \coloneq \mathtt{O}(O', C', D'), \mathtt{D}(D', R'), \mathtt{C}(C', M', \text{`R'})$$

Figure 6.6: Encoding queries derived from Q_{73} and Q_{74}

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 $Q_{76} \doteq_{\mathtt{bsb}} Q_{77}$. Because Q_{76} is the canonical form of Q_{75} and therefore equivalent to it, this entails $Q_{75} \doteq_{\mathtt{bsb}} Q_{77}$ and therefore $Q_{73} \equiv Q_{74}$. 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 $\mathtt{body}^+_v$.

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 $\mathtt{COCQL}^{\{f_\mathtt{b}, f_\mathtt{s}, f_\mathtt{n}\}}$ queries with output sorts $\{\!|\, \{\!|\, \mathtt{dom} \,|\!\}\, |\!\}$ and $\{\!|\, \{\!|\, \{\!|\, \mathtt{dom} \,|\!\}\, |\!\}\, |\!\}$, respectively.

```
Q:  SELECT SUM(qty)          Q′: SELECT SUM(X)
    FROM LI                      FROM (SELECT oid, SUM(qty) AS X
                                       FROM LI
                                       GROUP BY oid)
```

Many common SQL aggregation functions are *distributive* [48], and the ability to "re-aggregate" 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 $f_{\mathtt{s}}$, $f_{\mathtt{b}}$, $f_{\mathtt{s}}$ 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 $\overline{\S}$-equivalence of CEQs implies the existence of index-covering homomorphisms between $\overline{\S}$-normal forms. Throughout this section we will make the following assumptions:

[A1] $Q(\overline{\mathcal{I}}_1;\ldots;\overline{\mathcal{I}}_d;\overline{\mathcal{V}})$ and $Q'(\overline{\mathcal{I}}'_1;\ldots;\overline{\mathcal{I}}'_d;\overline{\mathcal{V}}')$ are two CEQs satisfying $Q \doteq_{\overline{\S}} Q'$.

[A2] Q and Q' are already in $\overline{\S}$-NF (justified by Theorem 3.1.17).

[A3] The body of Q is minimal relative to treating Q as a CQ with head $Q(\overline{\mathcal{I}}_{[1,d]})$ (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 $Q'(\overline{\mathcal{I}}'_{[1,d]})$.

[A4] Within the context of some database $\mathbb{D}$, we use Γ to denote both the embeddings of $\texttt{body}_Q$ into $\mathbb{D}$ and—by abuse of notation—the encoding relation $(Q)^{\mathbb{D}}$ with schema $Q(\overline{\mathcal{I}}_1;\ldots;\overline{\mathcal{I}}_d;\overline{\mathcal{V}})$. Then, given some value $\overline{a} \in \texttt{dom}(\overline{\mathcal{I}}_{[1,l-1]})$, we use $\Gamma[\overline{a}]$ to denote the sub-relation indexed by $\overline{a}$, which has schema $Q(\overline{\mathcal{I}}_l;\ldots;\overline{\mathcal{I}}_d;\overline{\mathcal{V}})$ and instance $\Pi_{\overline{\mathcal{I}}_{[l,d]}\overline{\mathcal{V}}}(\sigma_{\overline{\mathcal{I}}_{[1,l-1]}=\overline{a}}(\Gamma))$ (see Section 2.2.1). We define Φ 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 $\texttt{body}_Q$ to construct a "canonical database" $\mathbb{D}_Q$.
2. Choose a particular embedding $\gamma : \texttt{body}_Q \to \mathbb{D}_Q$ that yields a "canonical tuple" $\gamma(\overline{\mathcal{V}})$ within $(Q)^{\mathbb{D}_Q}$.
3. Use the definition of query equivalence/containment to argue the existence of an embedding $\phi : \texttt{body}_{Q'} \to \mathbb{D}_Q$ that yields the same canonical tuple $\phi(\overline{\mathcal{V}}') = \gamma(\overline{\mathcal{V}})$.
4. Define a mapping h in terms of ϕ, and use both the definition of $\mathbb{D}_Q$ 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 $\overline{\S}$-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 $\overline{\S}$.

The complexity due to nested (encoded) objects forces two modifications. First, we need to generalize the idea of "equal canonical tuples" $\gamma(\overline{\mathcal{V}}) = \phi(\overline{\mathcal{V}}')$ to "$\overline{\S}_{[i+1,d]}$-equal canonical sub-relations" $\Gamma[\overline{a}_{[1,i]}] \doteq_{\overline{\S}_{[i+1,d]}} \Phi[\overline{a}'_{[1,i]}]$ (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 $\overline{\S}$-certificate containing the necessary pseudo-trivial nodes, we require a carefully designed canonical database $\mathbb{D}_Q$ 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 $\mathbb{D}_Q$. 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., $\overline{\S} = \mathtt{b}^d$). Set equivalence is based upon existence, and so for set levels we inject large amounts of symmetry into the construction of $\mathbb{D}_Q$, 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 $\overline{\S} = \mathtt{s}^d$. 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 $\overline{\S} = \mathtt{n}^d$. 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 Γ and Φ, consider any $\overline{\S}$-certificate between them. A $\overline{\S}$-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 $\overline{\S}_{[i,d]}$ certificate that proves $\overline{\S}_{[i,d]}$-equality between two sub-relations $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$, for some $\overline{a} \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,i-1]}, \Gamma)$ and $\overline{a}' \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,i-1]}, \Phi)$.

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 $\texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}])$ to $\texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}'])$ and from $\texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}'])$ to $\texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}])$, and these mappings govern which $\overline{\S}_{[i+1,d]}$-certificates node n must have as children. We say that n is *trivial w.r.t. index value* $\overline{x}$ if any child $\overline{\S}_{[i+1,d]}$-certificate of n that involves sub-relation $\Gamma[\overline{a}\overline{x}]$ also involves sub-relation $\Phi[\overline{a}'\overline{x}]$, and vice versa (i.e., all of the mappings within n act as the identity for the value $\overline{x}$). We say that n is *trivial* if it is trivial w.r.t. every value within $\texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}]) \cup \texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}'])$.

We say that relations $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ *permit a proof that is trivial w.r.t. index value* $\overline{x}$ if

1. $\overline{x} \in \texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}])$,
2. $\overline{x} \in \texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}])$, and
3. $\Gamma[\overline{a}\overline{x}] \doteq_{\overline{\S}_{[i+1,d]}} \Phi[\overline{a}'\overline{x}]$.

If $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ permit a proof that is trivial w.r.t. index value $\overline{x}$, we call the node n *potentially trivial w.r.t. index value* $\overline{x}$. It is straightforward to show that if node n is potentially trivial w.r.t. $\overline{x}$ then we can re-arrange the mappings within n until we obtain a node that is trivial w.r.t. value $\overline{x}$ (while still forming a valid $\overline{\S}_{[i,d]}$-certificate between $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$).

We say that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ *permit a proof that is trivial* if for every value $\overline{x}$ in $\texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}]) \cup \texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}])$ they permit a proof that is trivial w.r.t. $\overline{x}$. 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 $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ only permit a trivial proof if $\Gamma[\overline{a}] \doteq_{\texttt{b}\overline{\S}_{[i+1,d]}} \Phi[\overline{a}']$; this follows because triviality implies equality (and hence bijectivity) between $\texttt{adom}(\overline{\mathcal{I}}_i, \Gamma)$ and $\texttt{adom}(\overline{\mathcal{I}}'_i, \Phi)$. If $\S_i \neq \texttt{b}$, proving that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ 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* $\overline{x}$ if n contains a child $\overline{\S}_{[i+1,d]}$-certificate that proves $\Gamma[\overline{a}\overline{x}_1] \doteq_{\overline{\S}_{[i+1,d]}} \Phi[\overline{a}'\overline{x}_2]$, where $\overline{x}_1 \in \texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}])$ and $\overline{x}_2 \in \texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}'])$ are two (not necessarily distinct) permutations of $\overline{x}$. We say that n is *pseudo-trivial* if it is pseudo-trivial w.r.t. every value within $\texttt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}]) \cup \texttt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}'])$.

We say that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ *permit a proof that is pseudo-trivial w.r.t. value* $\overline{x}$

1. $\exists \overline{x}_1 \in \mathtt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}])$ that is a permutation of $\overline{x}$,
2. $\exists \overline{x}_2 \in \mathtt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}])$ that is a permutation of $\overline{x}$, and
3. $\Gamma[\overline{a}\overline{x}_1] \doteq_{\overline{\S}_{[i+1,d]}} \Phi[\overline{a}'\overline{x}_2]$.

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 $\mathtt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}]) \cup \mathtt{adom}(\overline{\mathcal{I}}'_i, \Phi[\overline{a}])$ 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* $\overline{\mathcal{V}}$ *and* $\overline{\mathcal{V}}'$*. 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 $\mathbb{D}$ such that for some $\overline{a} \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,j]}, \Gamma)$, sub-relation $\Gamma[\overline{a}]$ contains two tuples with differing values for output column I. In contrast, for every $\overline{a}' \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,j]}, \Phi)$ the output column I' is constant across all tuples in $\Phi[\overline{a}']$, implying $\Gamma[\overline{a}] \not\doteq_{\overline{\S}_{[j+1,d]}} \Phi[\overline{a}']$ (a contradiction). Hence, conclude $i \not> j$. Conclude $i \not< j$ similarly. □

Corollary A.1.2 *If for some index level* $i \in [1, d]$ *both* $\mathcal{I}_i \subseteq \mathcal{V}$ *and* $\mathcal{I}'_i \subseteq \mathcal{V}'$*, then for any database* $\mathbb{D}$ *and any* $\overline{\S}$*-certificate between* Γ *and* Φ*, 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 $\overline{\S}$-certificate a particular bag node is trivial, we use a counting argument based upon uniqueness of polynomials. The following example illustrates the core idea.

Student

school	dept	name
UW	CS	Dave
UW	Art	Art

(a) $\mathbb{D}_{10}$ = {Student}

Student

school	dept	name
UW	CS	Dave
UW	CS	Dave
UW	Art	Art

(b) $\mathbb{D}_{11}$ = {Student}

Figure A.1: Databases $\mathbb{D}_{10}$ and $\mathbb{D}_{11}$ for Example 65

R_{21}

A	C	C
UW	Dave	Dave
UW	Art	Art

R_{22}

A	B	C	C
UW	CS	Dave	Dave
UW	Art	Art	Art

R_{23}

B	C	D	C
CS	Dave	CS	Dave
Art	Art	Art	Art

Figure A.2: Query results $R_{21} = (Q_{78})^{\mathbb{D}_{10}}$, $R_{22} = (Q_{79})^{\mathbb{D}_{10}}$, and $R_{23} = (Q_{80})^{\mathbb{D}_{10}}$.

Example 65 Consider the following three conjunctive encoding queries

$$Q_{78}(A, C; C) \text{ :- } \texttt{Student}(A, B, C)$$
$$Q_{79}(A, B, C; C) \text{ :- } \texttt{Student}(A, B, C)$$
$$Q_{80}(B, C, D; C) \text{ :- } \texttt{Student}(A, B, C), \texttt{Student}(A, D, C)$$

and the database $\mathbb{D}_{10}$ shown in Figure A.1a. The results of evaluating the queries over $\mathbb{D}_{10}$ 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 $\mathbb{D}_{11}$ shown in Figure A.1b which was created from $\mathbb{D}_{10}$ by "diversifying" the constant CS into two distinct constants (which we show as CS and CS). Under $\mathbb{D}_{11}$, 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.

R_{24}

A	C	C
UW	Dave	Dave
UW	Art	Art

R_{25}

A	B	C	C
UW	CS	Dave	Dave
UW	CS	Dave	Dave
UW	Art	Art	Art

R_{26}

B	C	D	C
CS	Dave	CS	Dave
CS	Dave	CS	Dave
CS	Dave	CS	Dave
CS	Dave	CS	Dave
Art	Art	Art	Art

Figure A.3: Query results $R_{24} = (Q_{78})^{\mathbb{D}_{11}}$, $R_{25} = (Q_{79})^{\mathbb{D}_{11}}$, and $R_{26} = (Q_{80})^{\mathbb{D}_{11}}$.

A.2.1 Colour-Diversification

Suppose that we are given an infinite palette of colours, each indexed by a positive integer:

$$\texttt{colour}_1\ \texttt{colour}_2\ \texttt{colour}_3\ \texttt{colour}_4\ \texttt{colour}_5\ldots$$

(It is intentional and significant that $\texttt{colour}_1$ appears transparent.)

Let $\widehat{\texttt{dom}} \subset \texttt{dom}$ be any set of n constants adhering to some total ordering $<$.

$$\widehat{\texttt{dom}} = \{\texttt{c}_1, \ldots, \texttt{c}_n\} \qquad \forall 1 \leq i < j \leq n : \texttt{c}_i < \texttt{c}_j$$

For each integer $i \geq 2$, let $\widehat{\texttt{dom}_i}$ be a fresh set of constants isomorphic to $\widehat{\texttt{dom}}$, and let $\delta_i : \widehat{\texttt{dom}} \to \widehat{\texttt{dom}_i}$ be a function that "paints" each $\texttt{c}_j \in \widehat{\texttt{dom}}$ with colour $\texttt{colour}_i$ to yield the constant $\texttt{c}_j \in \widehat{\texttt{dom}_i}$. As implied by the transparency of $\texttt{colour}_1$, we define $\widehat{\texttt{dom}_i} = \widehat{\texttt{dom}}$ and the painting function $\delta_1 : \widehat{\texttt{dom}} \to \widehat{\texttt{dom}_1}$ as the identity function. Finally, because the different paintings of $\widehat{\texttt{dom}}$ are mutually disjoint, we can define a single whitewash[1] function $\delta^{-1} : \bigcup_i \widehat{\texttt{dom}_i} \to \widehat{\texttt{dom}}$ as follows.

$$\forall i\ \forall \texttt{c}_j \in \widehat{\texttt{dom}_i} :\ \delta^{-1}(\texttt{c}_j) := \texttt{c}_j \tag{A.1}$$

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 $\widehat{\texttt{dom}}$ of n constants, a* diversification point for $\widehat{\texttt{dom}}$ *is any n-dimensional point $\overline{r} \in \mathbb{N}^n$ whose coordinates are positive integers.*

Definition A.2.2 ($\overline{r}$-Colour-Diversification) *Given domain $\widehat{\texttt{dom}}$, diversification point $\overline{r}$ for $\widehat{\texttt{dom}}$, and any tuple t over $\widehat{\texttt{dom}}$,*

$$t = \langle\, \texttt{c}_{i_1}, \texttt{c}_{i_2}, \ldots, \texttt{c}_{i_m} \,\rangle$$

the colour-diversification of t w.r.t. $\overline{r}$*—denoted $\Delta^{\overline{r}}(t)$—is the set of all possible "paintings" of t generated by independently choosing for each tuple component $\texttt{c}_{i_j}$ one of the first r_{i_j} colours in the colour palette. We also refer to $\Delta^{\overline{r}}(t)$ the* $\overline{r}$-colour-diversification of t *or the* $\overline{r}$-diversification of t.

We can colour-diversify an entire database $\mathbb{D}_i$ by colour-diversifying each tuple $t \in \mathbb{D}_i$ with the same point $\overline{r}$ (where $\overline{r}$ is a diversification point for active domain $\texttt{adom}(\mathbb{D}_i)$). We then obtain a new database $\mathbb{D}_j$

$$\mathbb{D}_j = \Delta^{\overline{r}}(\mathbb{D}_i) = \bigcup_{t \in \mathbb{D}_i} \Delta^{\overline{r}}(t)$$

[1] The inverse of painting

Student

school	dept	name
UW	CS	Dave
UW	CS	Dave
UW	CS	Dave
UW	Art	Art
UW	Art	Art
UW	Art	Art
UW	Art	Art
UW	CS	Dave
UW	CS	Dave
UW	CS	Dave
UW	Art	Art
UW	Art	Art
UW	Art	Art
UW	Art	Art

Figure A.4: Database $\Delta^{[2,1,3,2]}(\mathbb{D}_{10})$

which we call the $\overline{r}$*-(colour-)diversification of* $\mathbb{D}_i$. We also say that $\mathbb{D}_j$ is *colour-diverse w.r.t. point* $\overline{r}$—a.k.a. $\overline{r}$*-colour-diverse* or $\overline{r}$*-diverse*—if $\mathbb{D}_j$ is the result of performing $\overline{r}$-colour-diversification on some other database $\mathbb{D}_i$. Because colour-diversification is defined in terms of painting, the whitewashing function δ^{-1} from equation A.1 is also the inverse of $\overline{r}$-diversification (when extended to sets of tuples). Hence, given a database $\mathbb{D}_j$ that is $\overline{r}$-colour-diverse, we can always re-obtain the original database $\mathbb{D}_i$ by whitewashing $\mathbb{D}_j$.

$$\mathbb{D}_i = \delta^{-1}(\mathbb{D}_j) = \delta^{-1} \circ \Delta^{\overline{r}}(\mathbb{D}_i)$$

Note that when we say $\mathbb{D}_j$ is $\overline{r}$-colour-diverse, it implies that $\overline{r}$ is a diversification point for active domain $\mathtt{adom}(\delta^{-1}(\mathbb{D}_j)) = \mathtt{adom}(\mathbb{D}_i)$, *not* active domain $\mathtt{adom}(\mathbb{D}_j)$ (which is already colour-diverse).

Example 66 Consider database $\mathbb{D}_{10}$ in Figure A.1a which contains four distinct constants in its active domain. Assume the following ordering for $\mathtt{adom}(\mathbb{D}_{10})$.

$$\mathtt{adom}(\mathbb{D}_{10}) = \{\mathsf{UW}, \mathsf{CS}, \mathsf{Dave}, \mathsf{Art}\}$$

Then, database $\mathbb{D}_{11}$ in Figure A.1b is the $[1,2,1,1]$-diversification of $\mathbb{D}_{10}$ because constant $\mathsf{c}_2 = \mathsf{CS}$ is painted with two colours. As another example, the $[2,1,3,2]$-diversification of $\mathbb{D}_{10}$ is shown in Figure A.4.

The cardinality of a tuple's colour-diversification depends upon both $\overline{r}$ and the number of occurrences of each constant $\mathsf{c}_i \in \widehat{\mathtt{dom}}$ within the tuple. For a given tuple t, let $\#(t, \mathsf{c}_i)$ denote the number of occurrences of constant c_i within t. Then, the set $\Delta^{\overline{r}}(t)$ has the following cardinality.

$$|\Delta^{\overline{r}}(t)| = \prod_{\mathsf{c}_i \in \widehat{\mathtt{dom}}} (r_i)^{\#(t,\mathsf{c}_i)} \tag{A.2}$$

Observe that if tuple t is fixed, then each value $\#(t, \mathsf{c}_i)$ 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 $\widehat{\texttt{dom}}$, the cardinality of the colour-diversification of S is then given by the following multivariate polynomial f_S over variables $\overline{r}$.

$$f_S(\overline{r}) = |\Delta^{\overline{r}}(S)| = |\bigcup_{t \in S} \Delta^{\overline{r}}(t)| = \sum_{t \in S} |\Delta^{\overline{r}}(t)| = \sum_{t \in S} \prod_{\mathsf{c}_i \in \widehat{\texttt{dom}}} (r_i)^{\#(t, \mathsf{c}_i)} \tag{A.3}$$

Furthermore, given two arbitrary sets S_1 and S_2 of tuples over $\widehat{\texttt{dom}}$, equation A.3 implies the following relationships between polynomials f_{S_1} and f_{S_2}.

$$f_{S_1}(\overline{r}) + f_{S_2}(\overline{r}) = f_{(S_1 \cup S_2)}(\overline{r}) \tag{A.4}$$

$$f_{S_1}(\overline{r}) \times f_{S_2}(\overline{r}) = f_{(S_1 \times S_2)}(\overline{r}) \tag{A.5}$$

Example 67 Consider again Example 66 and observe that database $\mathbb{D}_{10}$ contains two tuples $t_1 := \texttt{Student}(\mathsf{UW}, \mathsf{CS}, \mathsf{Dave})$ and $t_2 := \texttt{Student}(\mathsf{UW}, \mathsf{Art}, \mathsf{Art})$. The colour-diversification of t_1 and t_2 yields the following monomials

$$f_{t_1}(\overline{r}) = |\Delta^{\overline{r}}(\langle\, \mathsf{UW}, \mathsf{CS}, \mathsf{Dave} \,\rangle)| = (r_1)^1 (r_2)^1 (r_3)^1 (r_4)^0 = (r_1)(r_2)(r_3)$$
$$f_{t_2}(\overline{r}) = |\Delta^{\overline{r}}(\langle\, \mathsf{UW}, \mathsf{Art}, \mathsf{Art} \,\rangle)| = (r_1)^1 (r_2)^0 (r_3)^0 (r_4)^2 = (r_1)(r_4)^2$$

and so the $[2,1,3,2]$-diversification of $\mathbb{D}_{10}$ (shown in Figure A.4) has the following cardinality.

$$\begin{aligned} f_{\mathbb{D}_{10}}([2,1,3,2]) &= f_{t_1}([2,1,3,2]) + f_{t_2}([2,1,3,2]) \\ &= 2 \times 1 \times 3 + 2 \times 2^2 \\ &= 14 \end{aligned}$$

The symmetry that colour-diversification injects into the base relations of a database also propagates to the result of any queries. More specifically, let $\mathbb{D}$ be any database, $\overline{r}$ any diversification point for $\texttt{adom}(\mathbb{D})$, and Q'' be any conjunctive query over $\mathbb{D}$ 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.

$$(Q'')^{\Delta^{\overline{r}}(\mathbb{D})} = \Delta^{\overline{r}}\left((Q'')^{\mathbb{D}}\right) \tag{A.6}$$

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.

$$\delta^{-1}\left((Q'')^{\Delta^{\overline{r}}(\mathbb{D})}\right) = \delta^{-1} \circ \Delta^{\overline{r}}\left((Q'')^{\mathbb{D}}\right) = (Q'')^{\mathbb{D}} \tag{A.7}$$

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 $\mathbb{D}_{10}$ we distinguished between the results of Q_{78}, Q_{79}, and Q_{80} under $\mathtt{b}$-equality (and $\mathtt{n}$-equality). However, the reader can verify that point $[1,2,2,2]$ fails to distinguish between any of the queries under $\mathtt{b}$-equality (or $\mathtt{n}$-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 $\overline{r}$ for domain $\widehat{\mathtt{dom}}$, we say that $\overline{r}$ is* k-distinguishing *if for any two sets S_1, S_2 composed of tuples over $\widehat{\mathtt{dom}}$ with maximum arity k, either $f_{S_1} = f_{S_2}$ or $f_{S_1}(\overline{r}) \neq f_{S_2}(\overline{r})$.*

Definition A.2.4 (k-Distinguishing Database) *We say that database $\mathbb{D}$ is* k-distinguishing *if there exists some k-distinguishing diversification point $\overline{r}$ for domain $\delta^{-1}(\mathtt{adom}(\mathbb{D}))$ such that $\mathbb{D}$ is colour-diverse w.r.t. $\overline{r}$.*

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 $f(x_1,\ldots,x_n) \not\equiv 0$ 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 $\overline{r} \in S^n$, then*

$$\Pr[\exists i, j \in [1,m].(i < j \wedge f_i(\overline{r}) = f_j(\overline{r}))] < \frac{km^2}{2|S|}$$

Corollary A.2.7 *For any finite totally-ordered domain $\widehat{\mathtt{dom}}$ and non-negative integer k, there exists an infinite number of k-distinguishing diversification points.*

Corollary A.2.8 *Given any database $\mathbb{D}$ and any non-negative integer k, there exists a colour-diversification of $\mathbb{D}$ 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 $\mathbb{D}$ and any $\overline{\S}$-certificate proving $\Gamma \doteq_{\overline{\S}} \Phi$. For any $i \in [1,d]$, let n be any node in level i; then, node n proves $\Gamma[\overline{a}] \doteq_{\overline{\S}_{[i,d]}} \Phi[\overline{a}']$ for some $\overline{a} \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,i-1]}, \Gamma)$ and $\overline{a}' \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,i-1]}, \Phi)$. Regardless of the value of semantic indicator $\S_i$, node n must at the very least enforce that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ encode equivalent sets of sub-objects; therefore, let $\mathtt{subobjects}(n)$ denote the set of sub-objects encoded within the sub-relations of $\Gamma[\overline{a}]$ and $\Phi[\overline{a}]$.

$$\mathtt{subobjects}(n) \coloneqq \textsc{Decode}(\Gamma[\overline{a}], \mathtt{s}\overline{\S}_{[i+1,d]}) = \textsc{Decode}(\Phi[\overline{a}'], \mathtt{s}\overline{\S}_{[i+1,d]}) \tag{A.8}$$

For each sub-object $o \in \mathtt{subobjects}(n)$, let $\Gamma[\overline{a}]|_o$ denote the subset of $\Gamma[\overline{a}]$ containing only the embeddings that correspond to an encoding of sub-object o.

$$\Gamma[\overline{a}]|_o \coloneqq \{\gamma \mid \exists \overline{b} \in \mathtt{adom}(\overline{\mathcal{I}}_i, \Gamma[\overline{a}]).\ \big(\gamma \in \Gamma[\overline{a}\overline{b}] \wedge \textsc{Decode}(\Gamma[\overline{a}\overline{b}], \overline{\S}_{[i+1,d]}) = o\big)\} \tag{A.9}$$

When viewed as an encoding relation, $\Gamma[\overline{a}]|_o$ has the same schema as $\Gamma[\overline{a}]$. Define relation $\Phi[\overline{a}']|_o$ analogously.

In order to invoke count-based reasoning, we need to model sub-object cardinalities as polynomials. Consider the relation $\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}])$—because $\mathbb{D}$ is colour-diverse w.r.t. some point $\overline{r}$, it follows from equation A.6 that $\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}])$ is $\overline{r}$-diverse. Now, for each sub-object $o \in \mathtt{subobjects}(n)$, consider the relation $\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}]|_o)$—unfortunately, this relation *is not* necessarily $\overline{r}$-diverse, because all of the values in attributes $\mathcal{I}_i \cap \mathcal{V}$ will be constant (they are functionally determined by o). However, if we define $\overline{\mathcal{J}}_i$ as the non-output columns in $\overline{\mathcal{I}}_i$, then the relation $\Pi_{\overline{\mathcal{J}}_i}(\Gamma[\overline{a}]|_o)$ *is* $\overline{r}$-diverse. Next, define set S_o as the whitewashing of the tuples in this relation instance

$$S_o \coloneqq \delta^{-1}\left(\mathbb{T}(\Pi_{\overline{\mathcal{J}}_i}(\Gamma[\overline{a}]|_o))\right) \tag{A.10}$$

and define polynomial f_{S_o} as in equation A.3. Finally, define $\overline{\mathcal{J}}'_i$, 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 $\mathbb{D}$ is $\overline{r}$-diverse.

$$\#(\Gamma[\overline{a}], o) = |\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}]|_o)| = |\Pi_{\overline{\mathcal{J}}_i}(\Gamma[\overline{a}]|_o)| = |\Delta^{\overline{r}}(S_o)| = f_{S_o}(\overline{r}) \tag{A.11}$$

$$\#(\Phi[\overline{a}'], o) = |\Pi_{\overline{\mathcal{I}}'_i}(\Phi[\overline{a}']|_o)| = |\Pi_{\overline{\mathcal{J}}'_i}(\Phi[\overline{a}']|_o)| = |\Delta^{\overline{r}}(S'_o)| = f_{S'_o}(\overline{r}) \tag{A.12}$$

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., $\overline{\S} = \mathtt{b}^d$).

Lemma A.2.9 *Given*

- *some database* $\mathbb{D}$ *such that* Γ *and* Φ *are non-empty,*

- *some* $\overline{\S}$*-certificate that proves* $\Gamma \doteq_{\overline{\S}} \Phi$*, and*
- *some level* $i \in [1, d]$ *such that* $\S_i = \mathtt{b}$*;*

if $\mathbb{D}$ *is* $(|\mathcal{I}_{[1,d]}| + |\mathcal{I}'_{[1,d]}|)$*-distinguishing then every node* $n^{\mathtt{b}}$ *in level* i *of the certificate is potentially pseudo-trivial.*

PROOF. Suppose that $\mathbb{D}$ 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 $n^{\mathtt{b}}$ in level i of the certificate. Then, $n^{\mathtt{b}}$ proves $\Gamma[\overline{a}] \doteq_{\mathtt{b}\overline{\S}_{[i+1,d]}} \Phi[\overline{a}']$ for some $\overline{a} \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,i-1]})$ and $\overline{a}' \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,i-1]})$;

For each $o \in \mathtt{subobjects}(n^{\mathtt{b}})$, bag equivalence requires that $\#(\Gamma[\overline{a}], o) = \#(\Phi[\overline{a}'], o)$. Because $\mathbb{D}$ is colour-diverse w.r.t. some k-distinguishing point $\overline{r}$, by equations A.11 and A.12, this implies $f_{S_o}(\overline{r}) = f_{S'_o}(\overline{r})$. 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 $\Pi_{\overline{\mathcal{J}}_i}(\Gamma[\overline{a}]|_o)$ and $\Pi_{\overline{\mathcal{J}}'_i}(\Phi[\overline{a}']|_o)$.

Lemma A.1.1 implies that the same set of constant columns both extends relation $\Pi_{\overline{\mathcal{J}}_i}(\Gamma[\overline{a}]|_o)$ into $\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}]|_o)$ and extends relation $\Pi_{\overline{\mathcal{J}}'_i}(\Phi[\overline{a}']|_o)$ into $\Pi_{\overline{\mathcal{I}}'_i}(\Phi[\overline{a}']|_o)$. Hence, the "permuting mappings" between $\Pi_{\overline{\mathcal{J}}_i}(\Gamma[\overline{a}]|_o)$ and $\Pi_{\overline{\mathcal{J}}'_i}(\Phi[\overline{a}']|_o)$ are easily extended into a "permuting mappings" between $\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}]|_o)$ and $\Pi_{\overline{\mathcal{I}}'_i}(\Phi[\overline{a}']|_o)$—in other words, $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ permit pseudo-trivial proofs for all of the index values in $\Pi_{\overline{\mathcal{I}}_i}(\Gamma[\overline{a}]|_o)$ and $\Pi_{\overline{\mathcal{I}}'_i}(\Phi[\overline{a}']|_o)$. Repeating this argument for each $o \in \mathtt{subobjects}(n^{\mathtt{b}})$, we conclude that $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ permit a pseudo-trivial proof (which means $n^{\mathtt{b}}$ is potentially pseudo-trivial). □

Lemma A.2.10 *If* $\overline{\S} = \mathtt{b}^d$ *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 $\mathbb{D}_Q$ as follows

$$\mathbb{D}_Q := \Delta^{\overline{r}}(\mathtt{body}_Q) \tag{A.13}$$

where $\overline{r}$ 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 $\mathtt{body}_Q$).

Consider any $\mathtt{b}^d$-certificate proving $\Gamma \doteq_{\mathtt{b}^d} \Phi$. Choose any $\gamma \in \Gamma$ that satisfies $\delta^{-1} \circ \gamma(\overline{\mathcal{I}}_{[1,d]}) = \overline{\mathcal{I}}_{[1,d]}$ (by construction of $\mathbb{D}_Q$, there exist many such γ). Let $\overline{a}_{[1,d]}$ denote the tuple of index values $\gamma(\overline{\mathcal{I}}_{[1,d]})$. By Lemma A.2.9, the root node of the certificate is *potentially* pseudo-trivial w.r.t. $\overline{a}_1$; therefore, re-arrange

the mappings within the node until it *is* pseudo-trivial w.r.t. $\overline{a}_1$. By recursing into any child $\mathtt{b}^{d-1}$-certificate involving sub-relation $\Gamma[\overline{a}_1]$ and repeating this procedure, we eventually arrive at a tuple node proving $\Gamma[\overline{a}_{[1,d]}] \doteq_{\varnothing} \Phi[\overline{a}'_{[1,d]}]$ for some $\overline{a}'_{[1,d]} \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,d]}, \Phi)$. Choose any $\phi \in \Phi[\overline{a}'_{[1,d]}]$.

Define $h := \delta^{-1} \circ \phi$. Then, h is a mapping from $\mathcal{B}'$ to $\delta^{-1}(\mathtt{adom}(\mathbb{D}_Q)) = \mathcal{B} \cup \mathcal{C}_Q$ that satisfies the following properties.

1. $h(\mathtt{body}_{Q'}) = \delta^{-1} \circ \phi(\mathtt{body}_{Q'}) \subseteq \delta^{-1}(\mathbb{D}_Q) = \mathtt{body}_Q$
2. $h(\overline{\mathcal{V}}') = \delta^{-1} \circ \phi(\overline{\mathcal{V}}') = \delta^{-1} \circ \gamma(\overline{\mathcal{V}}) = \overline{\mathcal{V}}$
3. $\forall i \in [1, d]$, $h(\mathcal{I}'_i) = \delta^{-1} \circ \phi(\mathcal{I}'_i) = \delta^{-1} \circ \gamma(\mathcal{I}_i) = \mathcal{I}_i$
   which holds because the node proving $\Gamma[\overline{a}_{[1,i-1]}] \doteq_{\overline{\S}_{[i,d]}} \Phi[\overline{a}'_{[1,i-1]}]$ was rearranged to be pseudo-trivial w.r.t. $\overline{a}_i$ and so $\phi(\overline{\mathcal{I}}'_i) = \overline{a}'_i$ is a permutation of $\overline{a}_i = \gamma(\overline{\mathcal{I}}_i)$.

Hence, h satisfies Definition 3.2.1. The proof is analogous for h' from Q to Q'. □

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 ($\overline{\S} = \mathtt{s}^d$).

Example 68 Consider a relation schema `Wrote(Name, Title, 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 $\mathbb{D}_{12}$ containing relation `Wrote`.

Now consider the following CEQ Q_{81}. Figure A.5 shows the encoding relation R_{27} that results from evaluating Q_{81} over $\mathbb{D}_{12}$, as well as the object o_{16} that results from decoding R_{27} with signature `ss`.

$$Q_{81}(N, M, Y; T; T) \text{ :- } \mathtt{Wrote}(N, T, Y), \mathtt{Wrote}(M, T, Y)$$

Wrote

Name	Title	Year
Amy	A1	2007
Amy	AB1	2007
Amy	A2	2008
Amy	AB2	2008
Amy	AB3	2008
Amy	AC1	2008
Amy	AC2	2008
Bob	B1	2007
Bob	AB1	2007
Bob	B2	2008
Bob	AB2	2008
Bob	AB3	2008
Chad	C1	2008
Chad	AC1	2008
Chad	AC2	2008

$o_{16} = \{$ { A1, AB1 },
{ A2, AB2, AB3, AC1, AC2 },
{ AB1 },
{ AB2, AB3 },
{ AC1, AC2 },
{ B1, AB1 },
{ B2, AB2, AB3 },
{ C1, AC1, AC2 } $\}$

R_{27}

N	M	Y	T	T
Amy	Amy	2007	A1	A1
Amy	Amy	2007	AB1	AB1
Amy	Amy	2008	A2	A2
Amy	Amy	2008	AB2	AB2
Amy	Amy	2008	AB3	AB3
Amy	Amy	2008	AC1	AC1
Amy	Amy	2008	AC2	AC2
Amy	Bob	2007	AB1	AB1
Amy	Bob	2008	AB2	AB2
Amy	Bob	2008	AB3	AB3
Amy	Chad	2008	AC1	AC1
Amy	Chad	2008	AC2	AC2
Bob	Amy	2007	AB1	AB1
Bob	Amy	2008	AB2	AB2
Bob	Amy	2008	AB3	AB3
Bob	Bob	2007	B1	B1
Bob	Bob	2007	AB1	AB1
Bob	Bob	2008	B2	B2
Bob	Bob	2008	AB2	AB2
Bob	Bob	2008	AB3	AB3
Chad	Amy	2008	AC1	AC1
Chad	Amy	2008	AC2	AC2
Chad	Chad	2008	C1	C1
Chad	Chad	2008	AC1	AC1
Chad	Chad	2008	AC2	AC2

Figure A.5: Database $\mathbb{D}_{12}$ = {Wrote}, query result R_{27} = $(Q_{81})^{\mathbb{D}_{12}}$, and object o_{16} = DECODE(R_{27}, ss).

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}.

$$Q_{82}(N, Y; T; T) \text{ :- } \texttt{Wrote}(N, T, Y)$$
$$Q_{83}(T; ; T) \text{ :- } \texttt{Wrote}(N, T, Y)$$
$$Q_{84}(N, M; T; T) \text{ :- } \texttt{Wrote}(N, T, Y), \texttt{Wrote}(M, T, Y)$$

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} and the same argument would apply to any other singleton set.

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.

$$Q_{85}(T', T''; T; T) \text{ :- } \texttt{Wrote}(N, T, Y), \texttt{Wrote}(M, T, Y), \\ \texttt{Wrote}(N, T', Y), \texttt{Wrote}(M, T'', Y)$$

The reader can verify that the relation $(Q_{85})^{\mathbb{D}_{12}}$ 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 $\mathbb{D}_{12}$, occurrences of the values A2 and B2 are corrolated with occurrences of other constants. Specifically, any embedding of Q_{85} into $\mathbb{D}_{12}$ 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 `Wrote`(Don, A2, 2008) and `Wrote`(Chad, A2, 2007) to $\mathbb{D}_{12}$ does not change the sub-object corresponding to A2-B2 or B2-A2 in $(Q_{85})^{\mathbb{D}_{12}}$, because these tuples do not increase the possible choices for authors who co-authored with Bob in 2008. However, by adding tuple `Wrote`(Chad, B2, 2008) to $\mathbb{D}_{12}$ we allow variable M to map to either Bob or Chad, which changes the sub-object A2-B2 (and B2-A2) in $(Q_{85})^{\mathbb{D}_{12}}$ to { AB2, AB3, AC1, AC2 } without affecting the value of sub-object Amy-Bob-2008/Bob-Amy-2008 in $(Q_{27})^{\mathbb{D}_{12}}$.

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 $\mathbb{D}_Q$

We now describe a method for constructing canonical database $\mathbb{D}_Q$ 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.

$$N > \max(|\mathcal{I}_{[1,d]}|, |\mathcal{I}'_{[1,d]}|) + 1 \tag{A.14}$$

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.

$$LGC_i := \{(\overline{y}_i, \overline{z}_i) \mid \overline{y}_i, \overline{z}_i \in [1, N]^{|\mathcal{I}_i|}\} \tag{A.15}$$

We say that component $c = (\overline{y}_i, \overline{z}_i)$ contains a *conflict at position* $i.j$ if $y_{i.j} = z_{i.j}$, and we use $CF\text{-}LGC_i$ to denote the conflict-free subset of LGC_i.

$$CF\text{-}LGC_i := LGC_i \setminus \{(\overline{y}_i, \overline{z}_i) \mid \exists j.\, (y_{i.j} = z_{i.j})\} \tag{A.16}$$

Observe that set LGC_i has size $N^{2|\mathcal{I}_i|}$, while set $CF\text{-}LGC_i$ has size $N^{|\mathcal{I}_i|}(N-1)^{|\mathcal{I}_i|}$.

Example 69 Suppose $Q = Q_{81}$ and $N = 6$. Because every integer in $[1, N]$ is a single digit, we will represent each $\overline{x} \in [1, N]^{|\mathcal{I}_i|}$ simply as a $|\mathcal{I}_i|$-digit number. Then, sets LGC_2 and $CF\text{-}LGC_2$ are as follows

$$LGC_2 = \{(1,1), \ldots, (1,6), (2,1), \ldots, (6,6)\}$$
$$CF\text{-}LGC_2 = LGC_2 \setminus \{(1,1), (2,2), \ldots, (6,6)\}$$

with sizes $N^{2|\mathcal{I}_2|} = 36$ and $N^{|\mathcal{I}_2|}(N-1)^{|\mathcal{I}_2|} = 30$, respectively. Similarly, LGC_1 and $CF\text{-}LGC_1$ are as follows

$$LGC_1 = \{(111,111),\ldots(111,666),(112,111),\ldots(666,666)\}$$
$$CF\text{-}LGC_1 = LGC_1 \setminus \{(111,111),\ldots,(111,661),(112,111),\ldots,(666,666)\}$$

with sizes $N^{2|\mathcal{I}_1|} = 46656$ and $N^{|\mathcal{I}_1|}(N-1)^{|\mathcal{I}_1|} = 27000$, respectively.

Let LGS denote the set of *label-generating sequences*, composed out of label-generating components as follows.

$$LGS := \{c_1.c_2\ldots c_d \mid \forall i \in [1,d] : c_i \in LGC_i\} \tag{A.17}$$

We say that a sequence $s \in LGS$ is *conflict-free* if it is composed entirely of conflict-free components, and we use $CF\text{-}LGS$ to denote the conflict-free subset of LGS.

For each integer $m \in [1,d]$, let LGP_m denote the set of *label-generating prefixes of length* m, defined as follows.

$$LGP_m := \{c_1.c_2\ldots c_{(m-1)}.\overline{y}_m \mid \forall i \in [1,m-1].\,(c_i \in LGC_i) \wedge \exists \overline{z}_m.\,((\overline{y}_m,\overline{z}_m) \in LGC_m)\} \tag{A.18}$$

Let LGP denote the set of all label-generating prefixes. Every sequence $s \in LGS$ corresponds to a unique prefix in each of $LGP_1,\ldots,LGP_d$. Conversely, every prefix $p \in LGP_m$ with $m < d$ can be extended in $N^{|\mathcal{I}_m|+|\mathcal{I}_{(m+1)}|}$ different ways to yield a prefix in LGP_{m+1}, while every prefix $p \in LGP_d$ can be extended in $N^{|\mathcal{I}_d|}$ different ways to yield a complete sequence in LGS. We say that a prefix $p \in LGP$ is *conflict-free* if it can be iteratively extended into a conflict-free sequence, and we use $CF\text{-}LGP_m$ ($CF\text{-}LGP$) to denote the conflict-free subset of LGP_m (LGP). Every conflict-free prefix $p \in CF\text{-}LGP_m$ with $m < d$ can be extended in $(N-1)^{|\mathcal{I}_m|}N^{|\mathcal{I}_{(m+1)}|}$ different ways to yield a conflict-free prefix in $CF\text{-}LGP_{m+1}$, while every conflict-free prefix $p \in CF\text{-}LGP_d$ can be extended in $(N-1)^{|\mathcal{I}_d|}$ different ways to yield a conflict-free sequence in $CF\text{-}LGS$.

Example 70 Continuing Example 69, consider the following label-generating sequences $s_1, s_2, s_3 \in LGS$ and prefixes $p_{12}, p_3, p_{123} \in LGP$.

$$s_1 = (354,133).(6,5) \qquad p_{12} = (354,133).6$$
$$s_2 = (354,133).(6,6) \qquad p_3 = (354,134).1$$
$$s_3 = (354,134).(1,2) \qquad p_{123} = 354$$

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, 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

in 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 $\alpha^l : \mathcal{B} \to \mathcal{B}^l$ be a "labelling function" from each $x \in \mathcal{B}$ to the corresponding constant in $\mathcal{B}^l$ (which we will denote as x^l). Because all such sets $\mathcal{B}^l$ are mutually disjoint, we use α^{-1} to denote the obvious "de-labelling function" $\alpha^{-1} : (\bigcup_l \mathcal{B}^l) \to \mathcal{B}$ which we extend with identity over constants in $\mathcal{C}_Q$ (the constants in query Q).

For each label-generating sequence $s \in LGS$, we now define the following label-generating function $\beta_s : \mathcal{B} \to (\bigcup_l \mathcal{B}^l)$ which performs a particular labelling of the variables of Q.

$$\beta_{(\overline{y}_1,\overline{z}_1)\ldots(\overline{y}_d,\overline{z}_d)}(x) := \begin{cases} \alpha^{y_{1.j}}(x) & \text{if } \exists j \in [1,|\mathcal{I}_1|] \text{ such that } x = I_{1.j} \\ \alpha^{\overline{z}_1.y_{2.j}}(x) & \text{if } \exists j \in [1,|\mathcal{I}_2|] \text{ such that } x = I_{2.j} \\ \alpha^{\overline{z}_1.\overline{z}_2.y_{3.j}}(x) & \text{if } \exists j \in [1,|\mathcal{I}_3|] \text{ such that } x = I_{3.j} \\ \vdots & \vdots \\ \alpha^{\overline{z}_1\ldots\overline{z}_{(d-1)}.y_{d.j}}(x) & \text{if } \exists j \in [1,|\mathcal{I}_d|] \text{ such that } x = I_{d.j} \\ \alpha^{\overline{z}_1\ldots\overline{z}_d}(x) & \text{otherwise} \end{cases} \tag{A.19}$$

We immediately extend function β_s to tuples, sets, and atoms and with identity on constants in $\mathcal{C}$. We make the following observations about function β_s.

1. Because β_s is defined purely in terms of labelling functions, the de-labelling function α^{-1} acts as an inverse of β_s.
2. For any level $m \in [1,d]$, the labels that β_s assigns to index variables in $\mathcal{I}_m$ depend only on the length m prefix of s. Therefore, for each $p \in LGP_m$ we can define the function $\beta_p : \mathcal{I}_{[1,m]} \to (\bigcup_l \mathcal{B}^l)$ such that $\beta_p(\mathcal{I}_{[1,m]}) = \beta_s(\mathcal{I}_{[1,m]})$ for every extension s of p.
3. Suppose $s \in LGS$ 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 $\mathcal{I}_{[i+1,d]}$), the labelled tuple $\beta_s(t)$ *evidences the conflict at* $i.j$. That is, because $\beta_s(I_{i.j}) = I_{i.j}^{\overline{z}_1\ldots\overline{z}_{(i-1)}.y_{i.j}}$ and $\beta_s(I)$ is assigned a label that starts with $\overline{z}_1 \ldots \overline{z}_i$, 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 $\mathbb{D}_Q$ as follows.

$$\mathbb{D}_Q := \bigcup_{s \in CF\text{-}LGS} \beta_s(\mathtt{body}_Q) \tag{A.20}$$

Because variable s ranges over only conflict-free label-generating sequences, database $\mathbb{D}_Q$ does not contain any tuple that evidences any conflicts. Furthermore, observe that by de-labelling $\mathbb{D}_Q$ we re-obtain $\mathtt{body}_Q$.

$$\alpha^{-1}(\mathbb{D}_Q) = \mathtt{body}_Q \tag{A.21}$$

Example 71 Continuing Example 70, functions $\beta_{p_{123}}$, $\beta_{p_{12}}$, and β_{p_3} work as follows (recall that we assumed $Q = Q_{81}$).

$$\begin{array}{lll} \beta_{p_{123}}(\mathcal{I}_1) & = \beta_{s_1}(\mathcal{I}_1) = \beta_{s_2}(\mathcal{I}_1) = \beta_{s_3}(\mathcal{I}_1) & = \{N^3, M^5, Y^4\} \\ \beta_{p_{12}}(\mathcal{I}_{[1,2]}) & = \beta_{s_1}(\mathcal{I}_{[1,2]}) = \beta_{s_2}(\mathcal{I}_{[1,2]}) & = \{N^3, M^5, Y^4, T^{133.6}\} \\ \beta_{p_3}(\mathcal{I}_{[1,2]}) & = \beta_{s_3}(\mathcal{I}_{[1,2]}) & = \{N^3, M^5, Y^4, T^{134.1}\} \end{array}$$

Thus, applying functions β_{s_1}, β_{s_2}, and β_{s_3} to $\mathtt{body}_{Q_{81}}$ yields the following sets of tuples.

$$\begin{aligned} \beta_{s_1}(\mathtt{body}_{Q_{81}}) &= \{\mathtt{Wrote}(N^3, T^{133.6}, Y^4), \mathtt{Wrote}(M^5, T^{133.6}, Y^4)\} \\ \beta_{s_2}(\mathtt{body}_{Q_{81}}) &= \{\mathtt{Wrote}(N^3, T^{133.6}, Y^4), \mathtt{Wrote}(M^5, T^{133.6}, Y^4)\} \\ \beta_{s_3}(\mathtt{body}_{Q_{81}}) &= \{\mathtt{Wrote}(N^3, T^{134.1}, Y^4), \mathtt{Wrote}(M^5, T^{134.1}, Y^4)\} \end{aligned}$$

Database $\mathbb{D}_{Q_{81}}$ is constructed out of $|\mathit{CF\text{-}LGS}| = |\mathit{CF\text{-}LGC}_1| \times |\mathit{CF\text{-}LGC}_2| = 162000$ labelled copies of $\mathtt{body}_{Q_{81}}$, one for each conflict-free label-generating sequence.[2] Note that in this particular example, $\mathtt{body}_{Q_{81}}$ does not contain any non-index variables which implies

$$\beta_{s_2}(\mathtt{body}_{Q_{81}}) = \beta_{p_{12}}(\mathtt{body}_{Q_{81}}) = \beta_{s_1}(\mathtt{body}_{Q_{81}}) \subset \mathbb{D}_{Q_{81}}$$

even though $s_2 \notin \mathit{CF\text{-}LGS}$; 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 $\beta_{s_3}(\mathtt{body}_{Q_{81}}) \notin \mathbb{D}_{Q_{81}}$.

A.3.2 Canonical Objects

Having just defined the canonical database $\mathbb{D}_Q$, we now define the *canonical objects* encoded within Γ.

Definition A.3.1 (Canonical Object) *Given any integer $m \in [1, d]$ and any label-generating prefix $p \in LGP_m$, we define the* canonical object for prefix p *—denoted o_p—as follows.*

$$o_p := \textsc{Decode}(\Gamma[\beta_p(\overline{\mathcal{I}}_{[1,m]})], \overline{\S}_{[m+1,d]})$$

Observe that if $m = d$, the canonical object for prefix $p \in LGP_d$ is simply the tuple $\beta_p(\overline{\mathcal{V}})$. The following two lemmas reveal how the design of database $\mathbb{D}_Q$ endows canonical objects with specific properties.

Lemma A.3.2 *Given any prefix $p \in LGP_d$,*

1. *if p is conflict-free then $\beta_p(\overline{\mathcal{I}}_{[1,d]}) \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,d]}, R)$; and*
2. *if $\beta_p(\overline{\mathcal{I}}_{[1,d]}) \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,d]}, R)$ then for any position $i.j$ for which p contains a conflict, that conflict is evidenced within the tuple $\beta_p(\overline{\mathcal{I}}_{[i+1,d]}\overline{\mathcal{V}})$.*

[2] That database $\mathbb{D}_Q$ has exponential size compared to Q is not important because we never materialize $\mathbb{D}_Q$ nor evaluate any query against it. For the purposes of the proof, it suffices that $\mathbb{D}_Q$ is finite.

PROOF. If p is conflict-free, then there exists some $s \in \mathit{CF\text{-}LGS}$ that extends p, and so $\beta_p(\overline{\mathcal{I}}_{[1,d]}) = \beta_s(\overline{\mathcal{I}}_{[1,d]}) \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,d]}, R)$ follows by equation A.20 Conversely, suppose both that $\beta_p(\overline{\mathcal{I}}_{[1,d]}) \in \mathtt{adom}(\overline{\mathcal{I}}_{[1,d]}, R)$ and that p contains a conflict at position $i.j$; this entails $i \in [1, d-1]$, $j \in [1, |\mathcal{I}_i|]$, and $|\mathcal{I}_{[i+1,d]}| > 0$. If attribute $I_{i.j} \in \mathcal{V}$, then tuple $\beta_p(\overline{\mathcal{I}}_{[i+1,d]}\overline{\mathcal{V}})$ is guaranteed to evidence the conflict (see Section A.3.1). If $I_{i.j} \notin \mathcal{V}$, let $s \in \mathit{LGS}$ be any label-generating sequence that extends p. By the design of $\mathbb{D}_Q$, embedding $\gamma := \beta_s$ proves $Q \vDash (\mathcal{I}_{[1,i]} \setminus \{I_{i.j}\})\{ \twoheadrightarrow I_{i.j}\}$ which, by Definition 3.1.11, implies that Q is not in $\mathtt{s}^d$-NF (a contradiction). □

Lemma A.3.3 *Given any integer $m \in [1, d]$ and any prefix $p \in \mathit{CF\text{-}LGP}_m$, canonical object o_p contains enough information to reconstruct the value $\beta_p(\overline{\mathcal{I}}_m)$.*

PROOF. By induction on m. Our assumption that Q is in $\mathtt{s}^d$-NF implies that $\mathcal{I}_d \subseteq \mathcal{V}$ (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 $\beta_p(\overline{\mathcal{I}}_m)$. 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 $\beta_p(\overline{\mathcal{I}}_m)$ only requires deducing the sequence $\overline{z}_1 \ldots \overline{z}_{(m-1)}.\overline{y}_m$. Consider each prefix $q \in \mathit{LGP}_{(m+1)}$ that extends p. If q is conflict-free, then by the induction hypothesis o_q contains enough information to identify the value $\beta_q(\overline{\mathcal{I}}_{(m+1)})$, from which we can deduce the values of sequence $\overline{z}_1 \ldots \overline{z}_{(m-1)}.\overline{z}^q_m.\overline{y}^q_{(m+1)}$, where the values $\overline{z}^q_m$ and $\overline{y}^q_{(m+1)}$ 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 $\overline{z}^q_m$ corresponds to a conflict-free extension of p, from which we can deduce the value of $\overline{y}_m$. □

A.3.3 Pseudo-Trivial Certificate Nodes

We now show that sub-relations of Γ that encode canonical objects can only be equated to sub-relations of Φ by pseudo-trivial certificate nodes.

Lemma A.3.4 *Given*

- *any integer $m \in [1, d]$,*
- *any prefix $p \in \mathit{CF\text{-}LGP}_m$, 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 $\beta_p(\mathcal{I}_m)$.

PROOF. For each $j \in [1, |\overline{\mathcal{I}}_m|]$, consider the index attribute $I_{m.j} \in \mathcal{I}_m$ and the corresponding value $\beta_p(I_{m.j}) = I^{\overline{z}_1 \ldots \overline{z}_{(m-1)}.y_{m.j}}_{m.j}$. 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 $\mathbb{D}_Q$, 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 $I_{m.j}^{\overline{z}_1 \ldots \overline{z}_{(m-1)}.n}$. Therefore, isolating $\beta_p(I_{m.j})$ via data triangulation requires at least $N-1 > |\mathcal{I}'_{[1,d]}|$ triangulation points, which exceeds the size of $\overline{a}'_{[1,m]}$. Hence, $\overline{a}'_{[1,m]}$ must contain the value $\beta_p(I_{m.j})$.

□

Lemma A.3.5 *Given*

- *any conflict-free label-generating sequence $s \in$ CF-LGS,*
- *any $\overline{\S}$-certificate that proves $\Gamma \doteq_{\overline{\S}} \Phi$, and*
- *any integer $m \in [1, d]$;*

every certificate node involving sub-relation $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,m-1]})]$ is pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_m)$.

PROOF. Suppose that $Q \doteq_{\mathtt{s}^d} Q'$. 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 $|\overline{\mathcal{I}}_m| > |\overline{\mathcal{I}}'_m|$ 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 $\mathtt{s}^d$-certificate proving $R \doteq_{\mathtt{s}^d} R'$, the certificate must contain a tuple node proving $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,d]})] \doteq_{\varnothing} \Phi[\overline{a}'_{[1,d]}]$ for some $\overline{a}'_{[1,d]} \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,d]}, \Phi)$. Lemma A.3.4 implies that $\overline{a}'_1$ contains all values in $\beta_s(\overline{\mathcal{I}}_1)$, and so by choice of Q above conclude that $|\overline{\mathcal{I}}_1| = |\overline{\mathcal{I}}'_1|$ and therefore $\overline{a}'_1$ is a permutation of $\beta_s(\overline{\mathcal{I}}_1)$. A simple induction proves that $\overline{a}'_m$ is a permutation of $\beta_s(\overline{\mathcal{I}}_m)$ for every $m \in [1, d]$.

□

Lemma A.3.6 *If $\overline{\S} = \mathtt{s}^d$ 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 $\mathtt{s}^d$-certificate proving $\Gamma \doteq_{\mathtt{s}^d} \Phi$. The certificate must contain a tuple node proving $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,d]})] \doteq_{\varnothing} \Phi[\overline{a}'_{[1,d]}]$ for some $\overline{a}'_{[1,d]} \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,d]}, \Phi)$. Choose any $\phi \in \Phi[\overline{a}'_{[1,d]}]$.

Define $h := \alpha^{-1} \circ \phi$. Then, h is a mapping from $\mathcal{B}'$ to $\alpha^{-1}(\mathtt{adom}(\mathbb{D}_Q)) = \mathcal{B} \cup \mathcal{C}_Q$ that satisfies the following properties.

1. $h(\mathtt{body}_{Q'}) = \alpha^{-1} \circ \phi(\mathtt{body}_{Q'}) \subseteq \alpha^{-1}(\mathbb{D}_Q) = \mathtt{body}_Q$
2. $h(\overline{\mathcal{V}}') = \alpha^{-1} \circ \phi(\overline{\mathcal{V}}') = \alpha^{-1} \circ \beta_s(\overline{\mathcal{V}}) = \overline{\mathcal{V}}$
3. $\forall i \in [1, d]$, $h(\mathcal{I}'_i) = \alpha^{-1} \circ \phi(\mathcal{I}'_i) = \alpha^{-1} \circ \beta_s(\mathcal{I}_i) = \mathcal{I}_i$
   By Lemma A.3.5 the node proving $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,i-1]})] \doteq_{\overline{\S}_{[i,d]}} \Phi[\overline{a}'_{[1,i-1]}]$ is pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_i)$, and so $\phi(\overline{\mathcal{I}}'_i) = \overline{a}'_i$ is a permutation of $\beta_s(\overline{\mathcal{I}}_i)$.

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 $\overline{\S}$-certificate (assuming $Q \doteq_{\overline{\S}} Q'$). 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 ($\overline{\S} = \mathtt{n}^d$).

Example 72 Consider the following two CEQs, which assume the same relation schema `Wrote(Name, Title, Year)` as used in Example 68.

$$Q_{86}(T, Y; Y) \coloneq \mathtt{Wrote}(N, T, Y)$$
$$Q_{87}(N, T, Y; Y) \coloneq \mathtt{Wrote}(N, T, Y)$$

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 $\mathbb{D}_{13}$ formed from $\mathtt{body}_{Q_{86}}$,

$$\mathbb{D}_{13} = \{\mathtt{Wrote}(N, T, Y)\}$$

assume $\mathtt{adom}(\mathbb{D}_{13})$ has order $\{N, T, Y\}$, and choose any 3-distinguishing diversification point $\overline{r}$. For conciseness of presentation we will use $\overline{r} = [2, 3, 2]$ 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 $\mathbb{D}_{14} = \Delta^{[2,3,2]}(\mathbb{D}_{13})$. Whereas $\mathbb{D}_{14}$ 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 $\mathbb{D}_{15}$ formed from two distinct copies of $\mathtt{body}_{Q_{86}}$,

$$\mathbb{D}_{15} = \{\mathtt{Wrote}(N, T, Y), \mathtt{Wrote}(N', T', Y')\}$$

assume $\mathtt{adom}(\mathbb{D}_{13})$ has order $\{N, T, Y, N', T', Y'\}$, and choose $\overline{r} = [2, 3, 2, 1, 1, 1]$. Figure A.7 shows the result of evaluating Q_{86} and Q_{87} over database $\Delta^{[2,3,2,1,1,1]}(\mathbb{D}_{15})$, along with the differing normalized bags obtained after decoding.

Wrote

Name	Title	Year
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

R_{28}

T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y

$o_{17} = \{\!\{ Y, Y \}\!\}$

R_{29}

N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y

Figure A.6: Database $\mathbb{D}_{14} = \Delta^{[2,3,2]}(\mathbb{D}_{13})$, query results $R_{28} = (Q_{86})^{\mathbb{D}_{14}}$ and $R_{29} = (Q_{87})^{\mathbb{D}_{14}}$, and object $o_{17} = \textsc{Decode}(R_{28}, \mathtt{n}) = \textsc{Decode}(R_{29}, \mathtt{n})$

Wrote

Name	Title	Year
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'

R_{30}

T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T	Y	Y
T'	Y'	Y'

R_{31}

N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N	T	Y	Y
N'	T'	Y'	Y'

$o_{18} = \{\!\{ Y, Y, Y, Y, Y, Y, Y' \}\!\}$

$o_{19} = \{\!\{ Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y' \}\!\}$

Figure A.7: Database $\mathbb{D}_{16} = \Delta^{[2,3,2,1,1,1]}(\mathbb{D}_{15})$, query results $R_{30} = (Q_{86})^{\mathbb{D}_{16}}$ and $R_{31} = (Q_{87})^{\mathbb{D}_{16}}$, and objects $o_{18} = \textsc{Decode}(R_{30}, \mathtt{n})$ and $o_{19} = \textsc{Decode}(R_{31}, \mathtt{n})$.

The key lesson from the above example is that in order for colour diversification to distinguish the results of two encoding queries under $\mathtt{n}$-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 $(|\mathcal{I}_{[1,d]}| + |\mathcal{I}'_{[1,d]}|)$-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 $\mathbb{D}_Q$

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.

$$LGC_i := \{1, 2\} \tag{A.22}$$

Let LGS denote the set of *label-generating sequences*, composed out of label-generating components as follows.

$$LGS := \{c_1.c_2 \dots c_d \mid \forall i \in [1, d] : c_i \in LGC_i\} \tag{A.23}$$

For each integer $m \in [1, d]$, let LGP_m denote the set of *label-generating prefixes of length* m, defined as follows.

$$LGP_m := \{c_1.c_2 \dots c_m \mid \forall i \in [1, m]. (c_i \in LGC_i)\} \tag{A.24}$$

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 $CF\text{-}LGC_i := LGC_i$ which then entails $CF\text{-}LGP_m = LGP_m$, $CF\text{-}LGP = LGP$, and $CF\text{-}LGS = LGS$.

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 $\mathcal{B}^l$ and the labelling function $\alpha^l : \mathcal{B} \to \mathcal{B}^l$ and the de-labelling function $\alpha^{-1} : (\bigcup_l \mathcal{B}^l) \to \mathcal{B}$ exactly as in Section A.3. For each label-generating sequence $s \in LGS$, we define the following label-generating function $\beta_s : \mathcal{B} \to (\bigcup_l \mathcal{B}^l)$ which performs a particular

labelling of the variables of Q.

$$\beta_{c_1.c_2\ldots c_d}(x) \coloneqq \begin{cases} \alpha^{c_1}(x) & \text{if } x \in \mathcal{I}_1 \\ \alpha^{c_1.c_2}(x) & \text{if } x \in \mathcal{I}_2 \\ \vdots & \vdots \\ \alpha^{c_1\ldots c_{(d-1)}}(x) & \text{if } x \in I_{(d-1)} \\ \alpha^{c_1\ldots c_d}(x) & \text{if } x \in I_d \\ \alpha^{c_1\ldots c_d}(x) & \text{otherwise} \end{cases} \tag{A.25}$$

We immediately extend function β_s to tuples, sets, and atoms and with identity on constants in $\mathcal{C}$. Equation A.25 is much simpler than equation A.19, and in this case it suffices that $\mathbb{L} \coloneqq LGP$ (which was not true for nested sets). We observe that α^{-1} again acts as an inverse of β_s, and for any level $m \in [1,d]$ and each $p \in LGP_m$ we again define the function $\beta_p : \mathcal{I}_{[1,m]} \to (\bigcup_l \mathcal{B}^l)$ as in Section A.3.

We now define the canonical database $\mathbb{D}_Q$ in two stages. First, we form the database $\mathbb{D}_Q^{\text{pre}}$ in terms of the label-generating sequences just defined, analogous to the canonical database used for nested sets (equation A.20).

$$\mathbb{D}_Q^{\text{pre}} \coloneqq \bigcup_{s \in CF\text{-}LGS} \beta_s(\mathtt{body}_Q) \tag{A.26}$$

Next, we let $\overline{r}$ be any $(|\mathcal{I}_{[1,d]}|+|\mathcal{I}'_{[1,d]}|)$-distinguishing diversification point for $\mathtt{adom}(\mathbb{D}_Q^{\text{pre}})$ and we use colour-diversification to define canonical database $\mathbb{D}_Q$, as we did with nested bags (equation A.13).

$$\mathbb{D}_Q \coloneqq \Delta^{\overline{r}}(\mathbb{D}_Q^{\text{pre}}) \tag{A.27}$$

Observe that by first whitewashing and then de-labelling $\mathbb{D}_Q$ we re-obtain $\mathtt{body}_Q$.

$$\alpha^{-1} \circ \delta^{-1}(\mathbb{D}_Q) = \mathtt{body}_Q \tag{A.28}$$

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 $\gamma\left(\mathcal{I}_{[1,d]} \setminus (\mathcal{I}_i \setminus \mathcal{V})\right) = \beta_s\left(\mathcal{I}_{[1,d]} \setminus (\mathcal{I}_i \setminus \mathcal{V})\right)$ *and* $\delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) \neq \beta_s(\mathcal{I}_i \setminus \mathcal{V})$
then $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) \neq (\mathcal{I}_i \setminus \mathcal{V})$.

Proof. Assume w.l.o.g. that $s = c_1 \ldots c_d$. Assume to the contrary that

- $\gamma\left(\mathcal{I}_{[1,d]} \setminus (\mathcal{I}_i \setminus \mathcal{V})\right) = \beta_s\left(\mathcal{I}_{[1,d]} \setminus (\mathcal{I}_i \setminus \mathcal{V})\right)$,
- $\delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) \neq \beta_s(\mathcal{I}_i \setminus \mathcal{V})$, and
- $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) = (\mathcal{I}_i \setminus \mathcal{V})$.

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 $\overline{\S} = \mathtt{n}^d$) and Theorem 3.1.10 it follows that

1. $|(\mathcal{I}_i \cap \mathcal{V}) \cup \mathcal{I}_{[i+1,d]}| > 0$, and
2. set $\mathcal{I}_{[1,i-1]}$ is *not* a strong $(I_{i.j}, (\mathcal{I}_i \cap \mathcal{V}) \cup \mathcal{I}_{[i+1,d]})$-articulation set of hypergraph H^Q. In other words, if we delete vertices $\mathcal{I}_{[1,i-1]}$ from hypergraph H^Q, the remaining hypergraph must contain an $(I_{i.j}, I)$-path for some variable $I \in (\mathcal{I}_i \cap \mathcal{V}) \cup \mathcal{I}_{[i+1,d]}$.

Now consider the function $\alpha^{-1} \circ \delta^{-1} \circ \gamma$. Because whitewashing function δ^{-1} acts as the identity over $\texttt{colour}_1$ and because α^{-1} is the inverse of β_s, it follows that $\alpha^{-1} \circ \delta^{-1} \circ \gamma\left(\mathcal{I}_{[1,d]} \setminus (\mathcal{I}_i \setminus \mathcal{V})\right) = \left(\mathcal{I}_{[1,d]} \setminus (\mathcal{I}_i \setminus \mathcal{V})\right)$. Combined with the above assumption and with equation A.28, this implies that $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\overline{\mathcal{I}}_{[1,d]}) = \overline{\mathcal{I}}_{[1,d]}$ and $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\texttt{body}_Q) \subseteq \texttt{body}_Q$. Because $\texttt{body}_Q$ 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 $\gamma(\mathcal{I}_{[1,i-1]})$ 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 $\mathcal{I}_{[i,d]}$—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 $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) = (\mathcal{I}_i \setminus \mathcal{V})$, conclude that $\delta^{-1} \circ \gamma(I_{i.j}) \in \beta_s(\mathcal{I}_i \setminus \mathcal{V})$.

Repeating this argument for each value of $j \in [1, |\mathcal{I}_i|]$ such that $I_{i.j} \notin \mathcal{V}$, and combining it with $\alpha^{-1} \circ \delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) = (\mathcal{I}_i \setminus \mathcal{V})$, conclude the contradiction $\delta^{-1} \circ \gamma(\mathcal{I}_i \setminus \mathcal{V}) = \beta_s(\mathcal{I}_i \setminus \mathcal{V})$. □

A.4.2 Canonical Objects

Having just defined the canonical database $\mathbb{D}_Q$, we now define the *canonical objects* encoded within Γ exactly the same as we defined them for nested sets (Definition A.3.1). That is, the canonical object o_p for prefix $p \in LGP_m$ is defined as follows.

$$o_p \coloneqq \textsc{Decode}(\Gamma[\beta_p(\overline{\mathcal{I}}_{[1,m]})], \overline{\S}_{[m+1,d]})$$

Observe that given any integer $m \in [1, d-1]$ and any prefix $p \in LGP_m$, 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 Γ that encode canonical objects can only be equated to sub-relations of Φ by pseudo-trivial certificate nodes.

Lemma A.4.2 *Given*

- *any label-generating sequence $s \in LGS$,*
- *any $\overline{\S}$-certificate that proves $\Gamma \doteq_{\overline{\S}} \Phi$, and*
- *any level $m \in [1, d]$;*

every certificate node involving sub-relation $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,m-1]})]$ *is potentially pseudo-trivial w.r.t.* $\beta_s(\overline{\mathcal{I}}_m)$.

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 $\mathbb{D}_Q$). 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 $|\mathcal{I}_l| > |\mathcal{I}'_l|$ at the first level l upon which Q and Q' disagree, starting at level d and counting down. Also, assume $s = c_1 \dots c_d$ 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 $p := c_1 \dots c_{(d-1)}$. 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 $n^{\mathtt{n}}$ be any certificate node proving $\Gamma[\overline{a}] \doteq_{\S_d} \Phi[\overline{a}']$ for $\overline{a} := \beta_s(\overline{\mathcal{I}}_{[1,d-1]})$ and some $\overline{a}' \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,d-1]}, \Phi)$. 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

$$\gamma(\mathcal{I}_{[1,d-1]}) = \beta_{p.1}(\mathcal{I}_{[1,d-1]}) = \beta_s(\mathcal{I}_{[1,d-1]}) \tag{A.29}$$

while by definition of canonical object $o_{p.1}$ every $\gamma \in \Gamma[\overline{a}]|_{o_{p.1}}$ satisfies the following.

$$\gamma(\mathcal{I}_d \cap \mathcal{V}) = \beta_{p.1}(\mathcal{I}_d \cap \mathcal{V}) = \beta_s(\mathcal{I}_d \cap \mathcal{V}) \tag{A.30}$$

Tuples in $S_{o_{p.1}}$ are formed by projecting $\overline{\mathcal{J}}_d$ 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) $\alpha^{-1} \circ \delta^{-1}(t)$ 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 γ satisfying

$$\alpha^{-1} \circ \delta^{-1} \circ \gamma(\mathcal{J}_d) = \mathcal{J}_d \tag{A.31}$$

$$\delta^{-1} \circ \gamma(\mathcal{J}_d) \neq \beta_{p.1}(\mathcal{J}_d) = \beta_s(\mathcal{J}_d) \tag{A.32}$$

which contradicts Lemma A.4.1. In contrast, the design of $\mathbb{D}_Q$ implies that $\beta_{p.1}(\overline{\mathcal{I}}_d) \in \mathtt{adom}(\overline{\mathcal{I}}_d, \Gamma[\overline{a}])$, and the definition of canonical object $o_{p.1}$ further implies $\beta_{p.1}(\overline{\mathcal{I}}_d) \in \mathtt{adom}(\overline{\mathcal{I}}_d, \Gamma[\overline{a}]|_{o_{p.1}})$; therefore, $S_{o_{p.1}}$ does contain the tuple $\beta_s(\overline{\mathcal{J}}_d) = \beta_{p.1}(\overline{\mathcal{J}}_d)$, for some ordering $\overline{\mathcal{J}}_d$ of $\mathcal{J}_d$.

3. Each monomial in $f_{S_{o_{p.1}}}$ (resp., $f_{S_{o_{p.2}}}$) corresponds to a collection of values occurring within a tuple in $S_{o_{p.1}}$ (resp., $S_{o_{p.2}}$). Define polynomial $f_{\text{GCD}} \coloneqq \texttt{GCD}(f_{S_{o_{p.1}}}, f_{S_{o_{p.2}}})$. By definition of greatest common divisor, there exists two polynomials f_1,f_2 with $\texttt{GCD}(f_1, f_2) = 1$ that satisfy $f_{S_{o_{p.1}}} = f_1 \times f_{\text{GCD}}$ and $f_{S_{o_{p.2}}} = f_2 \times f_{\text{GCD}}$. We will now prove by contradiction that $\texttt{degree}(f_{\text{GCD}}) = 0$. Suppose that $\texttt{degree}(f_{\text{GCD}}) = k > 0$. Because $S_{o_{p.1}}$ contains tuple $\beta_{p.1}(\overline{\mathcal{J}}_d)$, polynomial $f_{S_{o_{p.1}}}$ 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 $\mathbb{D}_Q$, 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 $f_{S_{o_{p.1}}}$ contains a monomial corresponding to values $\beta_{p.1}(\mathcal{J}_d \setminus J'_d) \cup \beta_{p.2}(J'_d)$, which further implies the existence of tuple $t \in S_{o_{p.1}}$ 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, $\texttt{GCD}(f_{S_{o_{p.1}}}, f_{S_{o_{p.2}}})$ is a constant.

4. Because $\Gamma[\overline{a}]$ and $\Phi[\overline{a}']$ encode the same normalized bag, we know

$$\frac{f_{S_{o_{p.1}}}(\overline{r})}{f_{S_{o_{p.2}}}(\overline{r})} = \frac{f_{S'_{o_{p.1}}}(\overline{r})}{f_{S'_{o_{p.2}}}(\overline{r})}$$

and so

$$f_{S_{o_{p.1}}}(\overline{r}) \times f_{S'_{o_{p.2}}}(\overline{r}) = f_{S_{o_{p.2}}}(\overline{r}) \times f_{S'_{o_{p.1}}}(\overline{r}) \tag{A.33}$$

follows. During construction of database $\mathbb{D}_Q$ point $\overline{r}$ was chosen to be $(|\mathcal{I}_{[1,d]}| + |\mathcal{I}'_{[1,d]}|)$-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.

$$f_{S_{o_{p.1}}} \times f_{S'_{o_{p.2}}} = f_{S_{o_{p.2}}} \times f_{S'_{o_{p.1}}} \tag{A.34}$$

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 $\phi \in \Phi[\overline{a}']|_{o_{p.1}}$ 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.

$$\texttt{degree}(f_{S_{o_{p.1}}}) = |\mathcal{I}_d \setminus \mathcal{V}| \geq |\mathcal{I}'_d \setminus \mathcal{V}'| = \texttt{degree}(f_{S'_{o_{p.1}}}) \tag{A.35}$$

Because $\texttt{GCD}(f_{S_{o_{p.1}}}, f_{S_{o_{p.2}}})$ is a constant, from equation A.34 conclude that $f_{S_{o_{p.1}}}$ must be a constant multiple of $f_{S'_{o_{p.1}}}$. Therefore, for each tuple $t \in S_{o_{p.1}}$ there exists a permutation of t within $S'_{o_{p.1}}$, and vice versa.

7. By the definitions of $\overline{a}$ and canonical object $o_{p.1}$ By $\beta_s(\overline{\mathcal{I}}_{[1,d-1]}) = \overline{a}$ and $\beta_s(\overline{\mathcal{I}}_d) = \beta_{p.1}(\overline{\mathcal{I}}_d)$, we know that $\Gamma[\overline{a}\beta_s(\overline{\mathcal{I}}_d)]$ 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 $\overline{x}' \in \texttt{adom}(\overline{\mathcal{I}}'_d, \Phi[\overline{a}'])$ such that $\Phi[\overline{a}'\overline{x}']$ encodes $o_{p.1}$ and $\overline{x}'$ is

a permutation of $\beta_s(\overline{\mathcal{I}}_d)$. Therefore, node $n^{\mathtt{n}}$ is potentially pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_d)$.

Induction Hypothesis: Suppose that there exists $i \in [1, d-1]$ such that the lemma is true whenever $m > i$.

Inductive Step: Suppose $m = i$. Define prefix $p := c_1 \dots c_{(i-1)}$. If $|\mathcal{I}_i| = 0$ then every certificate node at level i is trivially pseudo-trivial; therefore, assume that $|\mathcal{I}_i| > 0$. The proof is identical to the base case, with the following additions.

- By the induction hypothesis, for every level $j \in [i+1, d]$ every certificate node involving sub-relation $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,j-1]})]$ is *potentially* pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_j)$. Because potentially pseudo-trivial nodes can always be re-arranged until they *are* pseudo-trivial, assume that that within the given certificate every certificate node involving sub-relation $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,j-1]})]$ *is* pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_j)$.
- Within the second step we need to note that there exists an assignment $\gamma \in \Gamma[\overline{a}]|_{o_{p.1}}$ that satisfies $\gamma(\overline{\mathcal{I}}_{[i,d]}) = \beta_s(\overline{\mathcal{I}}_{[i,d]})$. Every tuple $t \in S_{o_{p.1}}$ corresponds to a sub-relation of $\Gamma[\overline{ax}]$ that is $\overline{\S}_{\overline{\S}_{[i+1,d]}}$-equal to $\Gamma[\overline{a}\beta_s(\overline{\mathcal{I}}_i)]$, and so by the induction hypothesis there must exist an assignment $\gamma' \in \Gamma[\overline{ax}]$ satisfying $\gamma'(\mathcal{I}_j) = \beta_s(\mathcal{I}_j)$ for all $j \in [i+1, d]$. This implies $\alpha^{-1} \circ \delta^{-1} \circ \gamma'(\mathcal{I}_{[i+1,d]}) = \beta_s(\mathcal{I}_{[i+1,d]})$, which is needed to contradict Lemma A.4.1.
- Within the sixth step, $|\mathcal{I}_i| \geq |\mathcal{I}'_i|$ follows from the choice of Q because the induction hypothesis implies that $|\mathcal{I}_j| = |\mathcal{I}'_j|$ for every $j \in [i+1, d]$.

□

Lemma A.4.3 *If $\overline{\S} = \mathtt{n}^d$ then there exists index-covering homomorphisms in both directions between Q and Q'.*

Proof. Choose any label-generating sequence $s \in LGS$. Consider any $\mathtt{n}^d$-certificate proving $\Gamma \doteq_{\mathtt{n}^d} \Phi$. By Lemma A.4.2, the root node of the certificate is *potentially* pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_1)$; therefore, re-arrange the mappings within the node until it *is* pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_1)$. By recursing into any child $\mathtt{n}^{d-1}$-certificate involving sub-relation $\Gamma[\beta_s(\overline{\mathcal{I}}_1)]$ and repeating this procedure, we eventually arrive at a tuple node proving $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,d]})] \doteq_{\varnothing} \Phi[\overline{a}'_{[1,d]}]$ for some $\overline{a}'_{[1,d]} \in \mathtt{adom}(\overline{\mathcal{I}}'_{[1,d]}, \Phi)$ such that for each $i \in [1, d]$, $\overline{a}'_i$ is a permutation of $\beta_s(\overline{\mathcal{I}}_i)$. Choose any $\phi \in \Phi[\overline{a}'_{[1,d]}]$.

Define $h := \alpha^{-1} \circ \delta^{-1} \circ \phi$. Then, h is a mapping from $\mathcal{B}'$ to $\alpha^{-1} \circ \delta^{-1}(\mathtt{adom}(\mathbb{D}_Q)) = \mathcal{B} \cup \mathcal{C}_Q$ that satisfies the following properties.

1. $h(\mathtt{body}_{Q'}) = \alpha^{-1} \circ \delta^{-1} \circ \phi(\mathtt{body}_{Q'}) \subseteq \alpha^{-1} \circ \delta^{-1}(\mathbb{D}_Q) = \mathtt{body}_Q$
2. $h(\overline{\mathcal{V}}') = \alpha^{-1} \circ \delta^{-1} \circ \phi(\overline{\mathcal{V}}') = \alpha^{-1} \circ \delta^{-1} \circ \beta_s(\overline{\mathcal{V}}) = \overline{\mathcal{V}}$
3. $\forall i \in [1, d]$, $h(\mathcal{I}'_i) = \alpha^{-1} \circ \delta^{-1} \circ \phi(\mathcal{I}'_i) = \alpha^{-1} \circ \delta^{-1} \circ \beta_s(\mathcal{I}_i) = \mathcal{I}_i$
   which follows because $\phi(\overline{\mathcal{I}}'_i) = \overline{a}'_i$ is a permutation of $\beta_s(\overline{\mathcal{I}}_i)$.

Hence, h satisfies Definition 3.2.1. The proof is analogous for h' from Q to Q'. □

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 $\S_m$ = s, we define LGC_m, $CF\text{-}LGC_m$, LGP_m, and $CF\text{-}LGP_m$ as we defined them in Section A.3. When $\S_m$ = n we instead use the corresponding definitions from Section A.4. When $\S_m$ = 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 $\beta_s : \mathcal{B} \rightarrow (\bigcup_l \mathcal{B}^l)$ is formed from equations A.19 and A.25 by selecting the labelling system appropriate to the semantics of each level. The canonical database $\mathbb{D}_Q$ 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 $\S_m$ = b, we use Lemma A.2.9 to argue that nodes involving $\Gamma[\beta_s(\overline{\mathcal{I}}_{[1,m-1]})]$ are potentially pseudo-trivial w.r.t. $\beta_s(\overline{\mathcal{I}}_m)$. When $\S_m$ = s, the argument uses Lemmas A.3.4 and A.3.5, and when $\S_m$ = n the argument uses Lemma A.4.2. The construction of index-covering homomorphisms is then identical to Lemma A.4.3.

References

[1] Hibernate: Relational persistence for Java and .NET. Available at http://www.hibernate.org. Retrieved 17 September 2009.

[2] Microsoft SQL Server 2008: ADO.NET entity framework. Available at http://www.microsoft.com/sqlserver/2008/en/us/ado-net-entity.aspx. Retrieved 17 September 2009.

[3] Ruby on Rails. Available at http://www.rubyonrails.org. Retrieved 17 September 2009.

[4] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, 1995.

[5] Serge Abiteboul and Nicole Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS'84)*, pages 191–200. ACM Press, 1984.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[7] William Ward Armstrong. Dependency structures of data base relationships. In *Proceedings of the IFIP Congress*, pages 580–583, August 1974.

[8] Paolo Atzeni and Nicola M. Morfuni. Functional dependencies in relations with null values. *Information Processing Letters*, 18(4):233–238, May 1984.

[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS'86)*, pages 1–15, New York, NY, USA, 1986. ACM.

[10] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[11] Claude Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[12] Gautam Bhargava, Piyush Goel, and Bala Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 304–315, New York, NY, USA, 1995. ACM.

[13] Nicole Bidoit. The Verso algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321–364, 1987.

[14] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[15] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 70–80, 2008.

[16] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th annual ACM Symposium on Theory of Computing*, pages 77–90. ACM Press, 1977.

[17] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998)*, pages 34–43. ACM Press, 1998.

[18] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, pages 190–200, 1995.

[19] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjuctive queries. In *Proceedings of the 12th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 1993)*, pages 59–70. ACM Press, 1993.

[20] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *Proceedings of the 6th International Conference on Database Theory (ICDT '97)*, volume 1186 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 1997.

[21] Sara Cohen. *Equivalence, Containment and Rewriting of Aggregate Queries.* PhD thesis, Hebrew University, May 2004.

[22] Sara Cohen. Equivalence of queries combining set and bag-set semantics. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'06)*, pages 70–79. ACM Press, 2006.

[23] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Containment of aggregate queries. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2003.

[24] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Transactions on Database Systems*, 31(2):672–715, 2006.

[25] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *Journal of the ACM*, 54(2):5, 2007.

[26] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 1999)*, pages 155–166. ACM Press, 1999.

[27] Sara Cohen, Yehoshua Sagiv, and Werner Nutt. Equivalences among aggregate queries with negation. *ACM Transactions on Computational Logic*, 6(2):328–360, 2005.

[28] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of 13th International Conference on Very Large Data Bases (VLDB'87)*, pages 197–208, 1987.

[29] David DeHaan, Per-Åke Larson, and Jingren Zhou. Stacked indexed views in Microsoft SQL Server. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 179–190, New York, NY, USA, 2005. ACM.

[30] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'08)*, pages 149–158, New York, NY, USA, 2008. ACM.

[31] Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, pages 225–241, 2003.

[32] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested XML queries. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 132–143, 2004.

[33] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[34] Carles Farré, Werner Nutt, Ernest Teniente, and Toni Urpí. Containment of conjunctive queries over databases with null values. In *Proceedings of the 11th International Conference on Database Theory (ICDT 2007)*, number 4353 in Lecture Notes in Computer Science, pages 389–403. Springer, 2007.

[35] Antonio L. Furtado and Larry Kerschberg. An algebra of quotient relations. In *Proceedings of the 1977 ACM SIGMOD International Conference on Management of Data*, pages 1–8, New York, NY, USA, 1977. ACM Press.

[36] César Galindo-Legaria. *Algebraic optimization of outerjoin queries*. PhD thesis, Harvard University, June 1992.

[37] César Galindo-Legaria. Outerjoins as disjunctions. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 348–358. ACM Press, 1994.

[38] César Galindo-Legaria. Parameterized queries and nesting equivalencies. Technical Report MSR-TR-2000-31, Microsoft Research, 2000.

[39] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 571–581, New York, NY, USA, 2001. ACM.

[40] César Galindo-Legaria and Arnon Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In Forouzan Golshani, editor, *Proceedings of the 8th International Conference on Data Engineering (ICDE'92)*, pages 402–409. IEEE Computer Society, 1992.

[41] César Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, 1997.

[42] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[43] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[44] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 331–342. ACM Press, 2001.

[45] Gang Gou, Maxim Kormilitsin, and Rada Chirkova. Query evaluation using overlapping views: Completeness and efficiency. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 37–48. ACM Press, 2006.

[46] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 160–172, New York, NY, USA, 1987. ACM.

[47] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the 9th International Conference on Data Engineering (ICDE'93)*, pages 209–218. IEEE Computer Society, 1993.

[48] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Total. In Stanley Y. W. Su, editor, *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, pages 152–159. IEEE Computer Society, 1996.

[49] Stéphane Grumbach, Tova Milo, and Yoram Kornatzky. Calculi for bags and their complexity. In *Proceedings of the 4th International Workshop on Database Programming Languages (DBPL-4)*, Workshops in Computing, pages 65–79. Springer, 1993.

[50] Stëphane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. Querying aggregate data. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'99)*, pages 174–184. ACM Press, 1999.

[51] Stéphane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. On the equivalence and rewriting of aggregate queries. *Acta Informatica*, 40(8):529–584, July 2004.

[52] Stephane Grumbach and Leonardo Tininini. Automatic aggregation using explicit metadata. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management (SSDBM 2000)*, pages 85–94, 2000.

[53] Stéphane Grumbach and Leonardo Tininini. On the content of materialized aggregate views. *Journal of Computer and System Sciences*, 66(1):133–168, February 2003.

[54] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 358–369. Morgan Kaufmann, 1995.

[55] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. On a hierarchy of classes for nested databases. *Information Processing Letters*, 36(5):259–266, 1990.

[56] Laura M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 377–388. ACM Press, 1989.

[57] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[58] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[59] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 205–216. ACM Press, 1996.

[60] Gerhard Hill and Andrew Ross. Reducing outer joins. *The VLDB Journal*, 18(3):599–610, June 2009.

[61] Oscar H. Ibarra and Jianwen Su. On the containment and equivalence of database queries with linear constraints (extended abstract). In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, pages 32–43. ACM Press, 1997.

[62] Tomasz Imielinski and Jr. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[63] International Standards Organization. Information technology—database language SQL (proposed revised text of DIS 9075). ISO/IEC 9075:1992, July 1992. Available at http://www.contrib.andrew.cmu.edu/%7Eshadow/sql/sql1992.txt. Retrieved 17 September 2009.

[64] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, 1995.

[65] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS'82)*, pages 124–138, New York, NY, USA, 1982. ACM.

[66] T.S. Jayram, Phokion G. Kolaitis, and Erik Vee. The containment problem for real conjunctive queries. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'06)*, pages 80–89. ACM Press, 2006.

[67] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167–189, 1984.

[68] Vitaliy L. Khizder and Grant E. Weddell. Reasoning about uniqueness constraints in object relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1295–1306, 2003.

[69] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.

[70] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[71] Per-Åke Larson and Jingren Zhou. View matching for outer-join views. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, 2005.

[72] Mark Levene and George Loizou. Semantics for null extended nested relations. *ACM Transactions on Database Systems*, 18(3):414–459, 1993.

[73] Mark Levene and George Loizou. Axiomatisation of functional dependencies in incomplete relations. *Theoretical Computer Science*, 206(1-2):283–300, October 1998.

[74] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views (extended abstract). In *Proceedings of the 14th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 1995)*, pages 95–104. ACM Press, 1995.

[75] Alon Y. Levy and Inderpal Singh Mumick. Reasoning with aggregation constraints. In *Advances in Database Technology - Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, volume 1057 of *Lecture Notes in Computer Science*, pages 514–534. Springer, 1996.

[76] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 96–107. Morgan Kaufmann, 1994.

[77] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects (extended abstract). In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, pages 20–31. ACM Press, 1997.

[78] Leonid Libkin and Limsoon Wong. Aggregate functions, conservative extensions, and linear orders. In *Proceedings of the 4th International Workshop on Database Programming Languages (DBPL-4)*, Workshops in Computing, pages 282–294. Springer, 1993.

[79] Leonid Libkin and Limsoon Wong. New techniques for studying set languages, bag languages and aggregate functions. In *Proceedings of the 13th ACM*

SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1994), pages 155–166. ACM Press, 1994.

[80] Hong-Cheu Liu and Jeffery X. Yu. Algebraic equivalences of nested relational operators. *Information Systems*, 30(3):167–204, 2005.

[81] Michael Meier, Michael Schmidt, and Georg Lausen. On chase termination beyond stratification. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*, 2009.

[82] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, page 706, New York, NY, USA, 2006. ACM.

[83] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Transactions on Database Systems*, 33(4):1–50, 2008.

[84] Edward F. Moore and Claude E. Shannon. Reliable circuits using less reliable relays. *Journal of the Franklin Institute*, 262(3,4):191–208, 281–297, 1956.

[85] I. S. Mumick, S. J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 247–258, New York, NY, USA, 1990. ACM.

[86] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, pages 314–330, New York, NY, USA, 1990. ACM.

[87] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 264–277. Morgan Kaufmann, 1990.

[88] G. Özsoyoglu, Z. M. Özsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[89] Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems*, 17(1):65–93, March 1992.

[90] Chang-Sup Park, Myoung Ho Kim, and Yoon-Joon Lee. Finding an efficient rewriting of OLAP queries using materialized views in data warehouses. *Decision Support Systems*, 32(4):379–399, March 2002.

[91] Glenn N. Paulley and Per-Åke Larson. Exploiting uniqueness in query optimization. In *Proceedings of the 10th International Conference on Data Engineering (ICDE 1994)*, pages 68–79. IEEE Computer Society, 1994.

[92] Glenn Norman Paulley. *Exploiting Functional Dependence in Query Optimization*. PhD thesis, University of Waterloo, May 2000.

[93] J. S. Provan and M. O. Ball. Computing network reliability in time polyomial in the number of cuts. *Operations Research*, 32(3):516–526, May-June 1984.

[94] Jun Rao, Hamid Pirahesh, and Calisto Zuzarte. Canonical abstraction for outerjoin optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 671–682, New York, NY, USA, 2004. ACM Press.

[95] Riccardo Rosati. On the decidability and finite controllability of query processing in databases with incomplete information. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '06)*, pages 356–365, New York, NY, USA, 2006. ACM.

[96] Arnon Rosenthal and César Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 291–299, New York, NY, USA, 1990. ACM.

[97] Kenneth A. Ross, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. *Theoretical Computer Science*, 193(1-2):149–179, February 1998.

[98] Steven Rudich and Avi Wigderson, editors. *Computational Complexity Theory*, volume 10 of *IAS/Park City Mathematics Series*, chapter Complexity Theory: from Godel to Feynman, pages 5–87. American Math Society, 2004.

[99] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[100] Marc H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proceedings of the International Conference on Database Theory (ICDT'86)*, volume 243 of *Lecture Notes in Computer Science*, pages 380–396. Springer, 1986.

[101] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of 22nd International Conference on Very Large Data Bases (VLDB'96)*, pages 318–329. Morgan Kaufmann, 1996.

[102] Dan Suciu. Bounded fixpoints for complex objects. *Theoretical Computer Science*, 176(1-2):283–328, April 1997.

[103] Stan J. Thomas and Patrick C. Fischer. Nested relational structures. In Paris C. Kanellakis, editor, *The Theory of Databases*, volume 3 of *Advances in Computing Research*, pages 269–307. JAI Press, 1986.

[104] Transaction Processing Performance Council. TPC benchmark DS (decision support): Standard specification, version 1.0.0d, December 2007. Available at http://www.tpc.org/tpcds/spec/tpcds1.0.0.d.pdf. Retrieved 17 September 2009.

[105] Transaction Processing Performance Council. TPC benchmark H (decision support): Standard specification, revision 2.8.0, September 2008. Available at http://www.tpc.org/tpch/spec/tpch2.8.0.pdf. Retrieved 17 September 2009.

[106] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the 6th International Conference on Database Theory (ICDT '97)*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 1997.

[107] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking for the nested relational calculus. *Theoretical Computer Science*, 371(3):183–199, March 2007.

[108] Ron van der Meyden. The complexity of querying infinite data about linearly ordered domains. *Journal of Computer and System Sciences*, 52(1):113–135, Feb. 1997.

[109] Yannis Vassiliou. Null values in data base management: a denotational semantics approach. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 162–169, New York, NY, USA, 1979. ACM.

[110] Yannis Vassiliou. Functional dependencies and incomplete information. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80)*, pages 260–269. VLDB Endowment, 1980.

[111] Fang Wei and Georg Lausen. Containment of conjunctive queries with safe negation. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2003.

[112] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'93)*, pages 26–36. ACM Press, 1993.

[113] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the 10th International Conference on Data Engineering (ICDE'94)*, pages 89–100. IEEE Computer Society, 1994.

[114] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *Proceedings of 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 345–357. Morgan Kaufmann, 1995.

[115] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex SQL queries using automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 105–116. ACM Press, 2000.