Preservation of Codd semantics in databases with SQL nulls

Konrad Pijanowski

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2022

Abstract

The *marked nulls* used in theoretical models of incompleteness in database research are commonly misrepresented by a single syntactic object `NULL` in SQL databases. It has been already shown that interpreting SQL nulls as *Codd nulls* (non-repeated marked nulls) in input databases alone is not enough to reconcile the two approaches. The main reason is that Codd semantics of SQL nulls should be also preserved in query answers, which are incomplete databases themselves. Unfortunately, the class of relational algebra queries preserving Codd semantics is not recursively enumerable. It does not mean, however, that we cannot recognize such queries. There exist a number of sufficient conditions ensuring the preservation of Codd semantics by respective relational algebra operations.

In this report, we introduce a query normalization step to the Codd semantics verification process, which transforms the query in question into an equivalent query that is more likely to satisfy sufficient conditions. Moreover, we introduce a variadic intersection operator which leads to more relaxed constraints for intersections of many tables. Thanks to the preprocessing step, we can apply the milder restrictions corresponding to this and other derived operations even if they are not used directly in the query. Also, we refine the model of the propagation of nullable values in the query. We do this by incorporating information about non-nullable attributes in query answers derived from a condition of a selection operation. All of these findings enable us to capture even more Codd semantics preserving queries.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Konrad Pijanowski*)

Acknowledgements

I would like to thank my supervisor, Paolo Guagliardo, for all the time he dedicated to guiding me in the development of my ideas. Our cooperation allowed me to gain invaluable experience and skills that I can now use to contribute to the scientific world.

I am also endlessly grateful to my parents for their unconditional love and support, which allowed me to become the person I am today.

Last but not least, I would like to thank my partner and my friends. The time spent together gave me all the energy and motivation I needed to get through this demanding journey.

Table of Contents

Chapter 1

Introduction

Nowadays, incomplete information in databases is a norm rather than an exception. Therefore, the ongoing research in database theory must take it into account and allow for the appropriate handling of missing values. Traditionally, this has been done using a well-established concept of "marked" nulls [5, 10]. For example, they frequently appear in various theoretical models of query evaluation applications, such as data exchange or data integration [1, 9]. On the other hand, many real-life systems operate on relational databases using SQL whose interpretation of nulls is significantly different. Most notably, using marked nulls we can express the fact that two nulls are the same ($\perp_1 = \perp_1$) or different ($\perp_1 \neq \perp_2$), while it is impossible to do so using SQL nulls. This is because all of them are denoted by the same syntactic symbol **NULL**.

To accommodate for this discrepancy, scientific literature considered SQL nulls to be modelled as Codd nulls (non-repeating marked nulls) with adjusted comparison semantics to mimic the three-valued logic used by the SQL standard [5]. After all, a single object representing all nulls is no longer a problem, as each **NULL** is interpreted as a distinct null anyway. However, this assumption was shown to be inaccurate in many cases [7]. The main argument is that the Codd semantics of SQL nulls should not only apply to incomplete databases but also should be preserved by queries executed on them.

To illustrate this point, let D be a SQL database D with two relations: $R = \{A : \textbf{NULL}, 2\}$ and $S = \{B : \textbf{NULL}\}$. Also, let D' be a copy of D in which all SQL nulls were replaced by unique marked nulls, e.g., $R = \{A : \perp_R, 2\}$ and $S = \{B : \perp_S\}$. Then, the answers to the query $R \times S$ evaluated on these two databases are following:

$Q(D)$:

A	B
NULL	**NULL**
2	**NULL**

$Q(D')$:

A	B
$\perp_R$	$\perp_S$
2	$\perp_S$

If SQL nulls were modelled by Codd nulls, then we could conclude that $Q(D)$ contains three distinct nulls since there are three **NULL** objects in the resulting table. Although, the evaluation of $Q(D')$ tells us that the two nulls in column B are the same, even though initially every null in D' was distinct. The inconsistency in the interpretation of the missing values in the results shows that even this simple query does not preserve the

Codd semantics of SQL nulls.

The argument behind the preservation of Codd semantics described before is captured in Figure 1. To explain it, let $\mathsf{codd}(D)$ be the result of replacing each null in a SQL database D with a unique marked null. To be precise, $\mathsf{codd}(D)$ is a set of isomorphic Codd databases as the names of the fresh marked nulls can be chosen arbitrarily. Now, if Codd nulls were to reliably model the SQL nulls, then an answer to a query on an incomplete database D should be the same (subject to the renaming of the nulls) regardless of whether we decide to:

- first replace the SQL nulls in D and then execute the query: $D \xrightarrow{\mathsf{codd}} D' \xrightarrow{Q} Q(D')$
- execute the query on a database with SQL nulls and then apply Codd semantics as the answer can be an incomplete table itself: $D \xrightarrow{Q} Q(D) \xrightarrow{\mathsf{codd}} Q(D')$

Unfortunately, this does not hold even for many simple conjunctive queries that are of special interest in the theoretical world.

In the light of the discrepancy between the interpretations of marked and SQL nulls and the fact that SQL fails to model Codd nulls, one needs to refer to other solutions attempting to bridge the gap between the theory and practice. One such approach could be to extend SQL with the ability to ”mark” nulls. Some progress in this direction has been made by [12], which implemented marked nulls for integer and varchar data types in the Postgres flavour of SQL. Although promising, that project is more of a prototype and would require further work if it was to become fully usable.

In the meantime, we can try to identify queries for which the Codd interpretation of SQL nulls is preserved. In general, [7] proved that it is impossible to recursively enumerate the set of all relational algebra queries that preserve Codd semantics. Instead, its approach utilises **`PRIMARY KEY`** / **`NOT NULL`** constraints on database schemas to analyse the propagation of nullable attributes in a query. Based on that, as well as other properties of queries, [7] introduced sufficient conditions for the preservation of Codd semantics of SQL nulls in an answer to the query. We will present these conditions later in chapter 2 in Theorem 1.

However, the test for the preservation of Codd semantics proposed in [7] has its shortcomings as well. Firstly, it struggles to recognise many Codd semantics preserving queries involving intersections that are applied one after another. In fact, in chapter 4, we show that there exists Codd semantics preserving queries with as few as two inter-

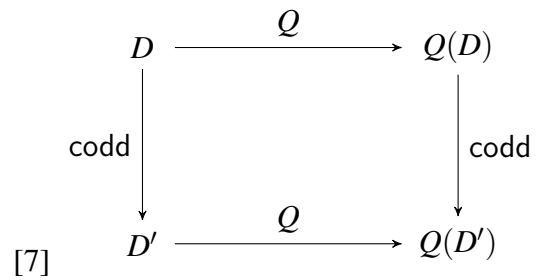


Figure 1: Condition for the preservation of Codd semantics for SQL nulls. Source: [7].

sections in a row that will never satisfy current conditions. Moreover, when working on this problem, we noticed that the testing process depends heavily on the exact syntactic formulation of the query. For example, even though two Codd semantics preserving queries $Q_1 = R$ and $Q_2 = R \cap R$ are equivalent, the latter is not always recognised as preserving Codd semantics, whereas the first one is.

1.1 Contributions

Motivated by the aforementioned shortcomings, we introduced a number of improvements to the process of recognizing Codd semantics preserving queries described in [7]. Specifically:

- We showed how the information about which attributes are certain to be made non-nullable by the selection operation can be derived from its condition.
- We introduced a variadic intersection operator together with a corresponding sufficient condition that enabled us to recognize more queries preserving Codd semantics.
- We suggested a normalization procedure which transforms the query in question into an equivalent query that is more likely to satisfy the sufficient conditions.
- We implemented the verification of sufficient conditions in relational algebra queries as a Java library *coddifier*.

1.2 Organisation

This report is structured in the following way:

- In chapter 2, we introduce a data model, relational algebra query language, and sufficient conditions for the preservation of Codd semantics.
- In chapter 3, we refine the propagation of nullable attributes in the selection operation.
- In chapter 4, we formally introduce the variadic intersection, which enables us to devise a milder restriction for an intersection of multiple tables.
- In chapter 5, we present a number of query rewrite rules that increase the chances of satisfying the sufficient conditions. Based on them, we suggest a query preprocessing step in the verification procedure which enables us to detect more queries preserving Codd semantics.
- In chapter 6, we provide a brief overview of the Java library *coddifier* that implements the findings described in [7] and this report.
- In chapter 7, we summarize the work done in the first part of the MInf project and set out plans for its second part.

Chapter 2

Preliminaries

Before we present any concrete findings, let us introduce a data model and a relational algebra query language based on which the sufficient conditions for the preservation of Codd semantics were derived. To be consistent, we will use the same definitions and follow the same conventions as in [7].

2.1 Data model, schemas, and (incomplete) databases

We start by defining a *bag* as an unordered collection of objects in which instances of the same element, unlike in sets, can repeat. We say that an element e in a bag B has multiplicity k, denoted as $\#(e,B) = k$ or $e \in_k B$, if e appears k times in B. Similarly, we can write $e \in B$ and $e \notin B$ to state a general fact that e is or is not in B, respectively. We also use notation $B \subseteq B'$, if $\#(e,B) \leq \#(e,B')$ for every $e \in B$. Finally, we define four bag operations: *union* $\cup$, *intersection* $\cap$, *difference* $-$, and *duplicate elimination* ε. If $e \in_m B$ and $e \in_n B'$, then: $\#(e, B \cup B') = m+n$, $\#(e, B \cap B') = \min(m,n)$, $\#(e, B - B') = \max(0, m-n)$, and $\#(e, \varepsilon(B)) = 1$ if $e \in B$, otherwise $\#(e, \varepsilon(B)) = 0$.

Now, let us take two countably infinite and disjoint sets of *names* and *values*. Any finite subset of names can be a *signature*. Then, a *record* is a map from some signature to values. Using these concepts we define a *table* as a bag of *records* over the same signature. $\mathsf{sig}(r)$ / $\mathsf{sig}(T)$ denotes the signature of a record r / table T, respectively.

The *projection* of a record r on a subset α of its signature is the restriction of r on α, denoted by $\pi_\alpha(r)$. For two records r and s of disjoint signatures, the *product* of r with s, denoted by $r \times s$, is the record over $\mathsf{sig}(r) \cup \mathsf{sig}(s)$ whose projections on $\mathsf{sig}(r)$ and $\mathsf{sig}(s)$ are r and s, respectively. For a record r, given $N \in \mathsf{sig}(r)$ and $N' \notin \mathsf{sig}(r)$, we define the following *renaming* operation:

$$\rho_{N \to N'}(r) \stackrel{\text{def}}{=\!=} \pi_{\mathsf{sig}(r)-N}(r) \times \{N' \mapsto r(N)\}\,.$$

The operations on records described above extend naturally to tables:

$$\pi_\alpha(T) \stackrel{\text{def}}{=\!=} \left\{ \underbrace{s, \ldots, s}_{k \text{ times}} \;\middle|\; k = \sum_{\substack{r \in T \\ \pi_\alpha(r) = s}} \#(r,T) \right\}$$

$$T \times T' \stackrel{\text{def}}{=\!=} \{ \underbrace{r \times s, \ldots, r \times s}_{m \cdot n \text{ times}} \mid r \in_m T,\ s \in_n T' \}$$

$$\rho_{N \to N'}(T) \stackrel{\text{def}}{=\!=} \{ \underbrace{r', \ldots, r'}_{k \text{ times}} \mid r \in_k T,\ r' = \rho_{N \to N'}(r) \}$$

The bag operations $\cup$, $\cap$ and $-$ can be applied to tables of the same a signature, which ensures the result is a table. Duplicate elimination ε applies without restrictions.

Next, a relational schema is a set of relation names together with a function sig which associates every relation name R with a set of its attributes $\mathsf{sig}(R)$ - its signature. Then, a database D maps each relation name R with a table $[\![R]\!]_D$ that is over the same signature as R. Each database instance can store values from only two countably infinite and disjoint sets of *constants* (Const) and nulls (Null). The Null set contains a special value $\mathtt{N}$ that is used to represent SQL's `NULL` object. Other nulls are denoted $\bot$, potentially with some subscript. By $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$ we denote sets of constants and nulls present in a database D.

Real-life databases support several constraints on data in a table. We are specifically interested in constraints that mark attributes as `NOT NULL`. To reflect them in our model, we partition the signature of a relation R into *nullable* and *non-nullable* signatures, denoted by $\mathsf{n\text{-}sig}(R)$ and $\mathsf{c\text{-}sig}(R)$ respectively. The non-nullable signature of R is a set of its non-nullable attributes, that is, attributes that are allowed to take only constant values (e.g., because of the `NOT NULL` or `PRIMARY KEY` constraint). On the other hand, there is no such restriction for nullable attributes in the nullable signature of R which can take both null and constant values.

Depending on the presence of nulls in a database as a whole, we can recognize four types of databases. A database D is a:

- *Complete* database if D does not contain any nulls - $\mathsf{Null}(D) = \varnothing$.
- *Naive* database if D does not contain any SQL nulls - $\mathtt{N} \notin \mathsf{Null}(D)$.
- *SQL* database if all nulls in D are SQL nulls - $\mathsf{Null}(D) = \{\mathtt{N}\}$.
- *Codd* database if D is a naive database and each null in the database is different.

Remark. We can say that a table is a complete, naive, SQL, or Codd table if it satisfies equivalent conditions.

Directly related to the notion of SQL and Codd databases is the idea of Codd interpretation of SQL nulls. Rephrasing what we said in the introduction using the newly defined concepts, theoreticians model incomplete databases using naive databases, whereas in practice SQL operates on SQL databases. To bridge this gap, SQL databases are usually interpreted as Codd databases. This is achieved by replacing each SQL null $\mathtt{N}$ with a unique element of $\mathsf{Null} - \{\mathtt{N}\}$ that is not yet present in the database.

To formalise this idea, given a record r, we denote by $\mathsf{sql}(r)$ the record r' over $\mathsf{sig}(r)$ such that:

$$r'(A) = \begin{cases} r(A) & \text{if } r(A) \in \mathsf{Const}, \\ \mathtt{N} & \text{otherwise} \end{cases}$$

Moreover, by $\mathsf{sql}^{-1}(r)$ we denote the set of all records r' such that $\mathsf{sql}(r') = r$.

The sql and sql^{-1} notation can be further extended to tables and databases. Namely, for a table T over $\mathsf{sig}(T)$, $\mathsf{sql}(T)$ is a table over the same signature that consists of records r, such that:

$$\#\big(r, \mathsf{sql}(T)\big) = \sum_{s \in \mathsf{sql}^{-1}(r)} \#(s, T)$$

For a database D, $\mathsf{sql}(D)$ denotes a database, having the same schema as D, where:

$$[\![R]\!]_{\mathsf{sql}(D)} = \mathsf{sql}([\![R]\!]_D)$$

The $\mathsf{sql}^{-1}(T)$ and $\mathsf{sql}^{-1}(D)$ denote sets of all tables T' and databases D', respectively, such that $\mathsf{sql}(T') = T$ and $\mathsf{sql}(D') = D$.

Finally, we can formalise what we mean by the Codd interpretation of a SQL database. For that we define $\mathsf{codd}(D)$ to be a set of all Codd databases in $\mathsf{sql}^{-1}(D)$. Even though this set may be infinite as the set of Null is countably infinite, all databases in it are isomorphic since they differ only in the names of the nulls. For that reason, we allow ourselves to talk about a single interpretation that is unique up to the renaming of nulls.

2.2 Query language

The sufficient conditions for Codd semantics preservation considered in this report apply to queries written in relational algebra (RA) for bags. To be consistent, we follow the syntax and semantics of the language as described in [7]. The syntax consists of two main constructs, that is *expressions* E and *conditions* θ, whose semantics are summarised by Figure 2.

A *term* t is either a name or a value, and its semantics $[\![t]\!]_r$ is given with respect to a record r: if t is a name in $\mathsf{sig}(r)$, then $[\![t]\!]_r = r(t)$, otherwise, if t is a value, $[\![t]\!]_r = t$.

Atomic conditions are equality/inequality comparisons between terms and tests that determine whether a term is null or constant. Complex conditions are constructed from

$$[\![R]\!]_D \text{ is given for every } R$$
$$[\![E_1 \,\mathsf{op}\, E_2]\!]_D \stackrel{\text{def}}{=\!=} [\![E_1]\!]_D \,\mathsf{op}\, [\![E_2]\!]_D \quad \text{for } \mathsf{op} \in \{\times, \cup, \cap, -\}$$
$$[\![\pi_\alpha(E)]\!]_D \stackrel{\text{def}}{=\!=} \pi_\alpha\big([\![E]\!]_D\big)$$
$$[\![\sigma_\theta(E)]\!]_D \stackrel{\text{def}}{=\!=} \sigma_\theta\big([\![E]\!]_D\big)$$
$$[\![\varepsilon(E)]\!]_D \stackrel{\text{def}}{=\!=} \varepsilon\big([\![E]\!]_D\big)$$
$$[\![\rho_{N\to N'}(E)]\!]_D \stackrel{\text{def}}{=\!=} \rho_{N\to N'}\big([\![E]\!]_D\big)$$

(a) EXPRESSIONS

$$[\![t_1 = t_2]\!]_r = \mathbf{t} \iff [\![t_1]\!]_r = [\![t_2]\!]_r \in \mathsf{Const}$$
$$[\![t_1 \neq t_2]\!]_r = \mathbf{t} \iff [\![t_1 = t_2]\!]_r \neq \mathbf{t}$$
$$[\![\mathsf{null}(t)]\!]_r = \mathbf{t} \iff [\![t]\!]_r \in \mathsf{Null}$$
$$[\![\mathsf{const}(t)]\!]_r = \mathbf{t} \iff [\![t]\!]_r \in \mathsf{Const}$$
$$[\![\theta_1 \wedge \theta_2]\!]_r = \mathbf{t} \iff [\![\theta_1]\!]_r = [\![\theta_2]\!]_r = \mathbf{t}$$
$$[\![\theta_1 \vee \theta_2]\!]_r = \mathbf{t} \iff [\![\theta_1]\!]_r = \mathbf{t} \vee [\![\theta_2]\!]_r = \mathbf{t}$$

(b) CONDITIONS

Figure 2: Semantics of relational algebra. Source: [7].

atomic ones by means of conjunction and disjunction. There is no explicit negation, as it can be propagated all the way down to atoms. The signature of a condition θ, denoted by $\mathsf{sig}(\theta)$, is the set of names appearing in it. Its semantics $[\![\theta]\!]_r$ is defined with respect to a record r such that $\mathsf{sig}(\theta) \subseteq \mathsf{sig}(r)$: it can be either $\mathbf{t}$ (true) or $\mathbf{f}$ (false), as determined by the rules in Figure 2b.

For a table T and a condition θ such that $\mathsf{sig}(\theta) \subseteq \mathsf{sig}(T)$, we can then define the following *selection* operation:

$$\sigma_\theta(T) \stackrel{\text{def}}{=\!=} \{\, \underbrace{r, \ldots, r}_{k \text{ times}} \mid r \in_k T,\ [\![\theta]\!]_r = \mathbf{t} \,\}$$

As for expressions, these are names of base relations present in the schema that can be further composed using standard operations of union $\cup$, intersection $\cap$, difference $-$, Cartesian product $\times$, selection σ, projection π, renaming ρ, and duplicate elimination ε. The *base* of an expression E, denoted by $\mathsf{base}(E)$, is the set of relation names that appear in it (i.e., the set of its atomic subexpressions).

The signature of each expression is defined recursively as follows:

$$\begin{aligned}
\mathsf{sig}(R) &\text{ is given for every } R \\
\mathsf{sig}(E_1 \,\mathsf{op}\, E_2) &= \mathsf{sig}(E_1) \text{ for } \mathsf{op} \in \{\cup, \cap, -\} \\
\mathsf{sig}(E_1 \times E_2) &= \mathsf{sig}(E_1) \cup \mathsf{sig}(E_2) \\
\mathsf{sig}\big(\sigma_\theta(E)\big) &= \mathsf{sig}\big(\varepsilon(E)\big) = \mathsf{sig}(E) \\
\mathsf{sig}\big(\pi_\alpha(E)\big) &= \alpha \\
\mathsf{sig}\big(\rho_{A \to B}(E)\big) &= \big(\mathsf{sig}(E) - \{A\}\big) \cup \{B\}
\end{aligned}$$

In a similar manner, we define the nullable signature of each expression:

$$\begin{aligned}
\mathsf{n\text{-}sig}(R) &\text{ is given for every } R \\
\mathsf{n\text{-}sig}(E_1 \,\mathsf{op}\, E_2) &= \mathsf{n\text{-}sig}(E_1) \cup \mathsf{n\text{-}sig}(E_2) \text{ for } \mathsf{op} \in \{\cup, \times\} \\
\mathsf{n\text{-}sig}(E_1 \cap E_2) &= \mathsf{n\text{-}sig}(E_1) \cap \mathsf{n\text{-}sig}(E_2) \\
\mathsf{n\text{-}sig}(E_1 - E_2) &= \mathsf{n\text{-}sig}(E_1) \\
\mathsf{n\text{-}sig}\big(\sigma_\theta(E)\big) &= \mathsf{n\text{-}sig}\big(\varepsilon(E)\big) = \mathsf{n\text{-}sig}(E) \\
\mathsf{n\text{-}sig}\big(\pi_\alpha(E)\big) &= \mathsf{n\text{-}sig}(E) \cap \alpha \\
\mathsf{n\text{-}sig}\big(\rho_{A \to B}(E)\big) &= \mathsf{n\text{-}sig}(E)[A/B]
\end{aligned}$$

where $\mathsf{n\text{-}sig}(E)[A/B]$ means that attribute A is replaced by B in $\mathsf{n\text{-}sig}(E)$. The non-nullable signature of any expression can be computed using the relation $\mathsf{c\text{-}sig}(E) = \mathsf{sig}(E) - \mathsf{n\text{-}sig}(E)$.

A relational algebra query over some schema is an expression that is well-defined with respect to this schema. One can recursively determine whether an expression is well-defined using the following rules:

- An atomic expression R is well-defined if R is a relation name in the schema.

- $E_1 \ \mathsf{op} \ E_2$, for $\mathsf{op} \in \{\cup, \cap, -\}$, is well-defined if both E_1 and E_2 are well-defined and $\mathsf{sig}(E_1) = \mathsf{sig}(E_2)$.
- $E_1 \times E_2$ is well-defined if E_1 and E_2 are well-defined and $\mathsf{sig}(E_1) \cap \mathsf{sig}(E_2) = \varnothing$.
- $\sigma_\theta(E)$ is well-defined if E is well defined and $\mathsf{sig}(\theta) \subseteq \mathsf{sig}(E)$.
- $\pi_\alpha(E)$ is well-defined if E is well defined and $\mathsf{sig}(\alpha) \subseteq \mathsf{sig}(E)$.
- $\rho_{A \to B}(E)$ is well-defined if E is well defined, $A \in \mathsf{sig}(E)$, and $B \notin \mathsf{sig}(E) - \{A\}$.
- $\varepsilon(E)$ is well-defined if E is well-defined.

Given two queries Q and Q' we say that Q is contained in Q', written as $Q \subseteq Q'$, if for every database D it is the case that $[\![Q]\!]_D \subseteq [\![Q']\!]_D$.

Closely related to the relational algebra query is a notion of its syntax tree.

Definition 1 ([7])**.** The *syntax tree* of an RA query Q is a binary (ordered) tree constructed as follows:

- Each relation symbol R is a single node labelled R.
- For each unary operation symbol op_1, the syntax tree of $\mathsf{op}_1(Q)$ has root labelled op_1 and the syntax tree of Q rooted at its single child.
- For each binary operation symbol op_2, the syntax tree of $Q \ \mathsf{op}_2 \ Q'$ has root labelled op_2 and the syntax trees of Q and Q' rooted at its left child and right child, respectively.

Remark. Each node in the syntax tree of Q defines a subquery of Q, so we can associate properties of such queries with properties of syntax tree nodes.

Finally, given two nodes N and N' we write $N \prec N'$ to indicate that N is a parent of N' or, equivalently, that N' is a child of N.

2.3 Queries preserving Codd semantics

We have already said that in order to be able to interpret SQL nulls as Codd nulls it must be the case that this interpretation not only applies to input databases but is also preserved by query answers. Figure 1 in the introduction depicts the intuition behind the condition for the Codd semantics preservation, which is formally defined below.

Definition 2 ([7])**.** A query Q preserves Codd semantics if for every Codd database D it holds that:

$$\mathsf{sql}\big([\![Q]\!]_D\big) = [\![Q]\!]_{\mathsf{sql}(D)} \qquad \text{(1a)} \qquad \text{and } [\![Q]\!]_D \text{ is a Codd table.} \qquad \text{(1b)}$$

Unfortunately, [7] proved that the class of queries preserving Codd semantics is not recursively enumerable. For that reason, we can only come up with syntactic restrictions

that guarantee this property. Guagliardo and Libkin introduced in [7] a number of conditions which can be satisfied by a node in a syntax tree.

Definition 3 ([7])**.** A node in the syntax tree of a query satisfies:

`NNC` *(non-nullable child)* - if one of its children is non-nullable.

`NNA` *(non-nullable ancestor/self)* - if either itself or one of its ancestors is non-nullable.

`DJN` *(disjoint nullable attributes)* - if its children have no common nullable attributes.

`DJB` *(disjoint bases)* - if its children have bases with no relation names in common.

Using these constraints they came up with sufficient conditions for the preservation of Codd semantics.

Theorem 1 ([7])**.** *Let Q be an RA query whose syntax tree is such that:*

a) *each* ε *node satisfies* `NNC`*;*

b) *each* $\cap$ *and* $-$ *node satisfies* `DJN`*;*

c) *each* $\times$ *node satisfies* `NNA`*;*

d) *each* $\cup$ *node satisfies* `NNC` *or* `DJB` *or* `NNA`*.*

Then, Q preserves Codd semantics.

Theorem 1 enables us to quickly verify whether a query is guaranteed to preserve Codd semantics. We will illustrate this process on the query $\pi_{A,B}(T \times S) \cap ((R-T) \cup \varepsilon(U))$ whose syntax tree is presented in Figure 3. The signature of each subexpression is given to the left of the node (underlined attributes are non-nullable). The respective sufficient conditions satisfied by the nodes are marked on the right. Now, in the right subquery, the ε node satisfies `NNC` as relation U is non-nullable. The $-$ node satisfies `DJN`, as its children have non-overlapping nullable signatures. The union satisfies all respective conditions, although one is enough for the preservation of Codd semantics. In the left subquery, the $\times$ node satisfies the `NNA` condition because the root of the query is non-nullable, which is the case, as the $\cap$ node satisfies `DJN`.

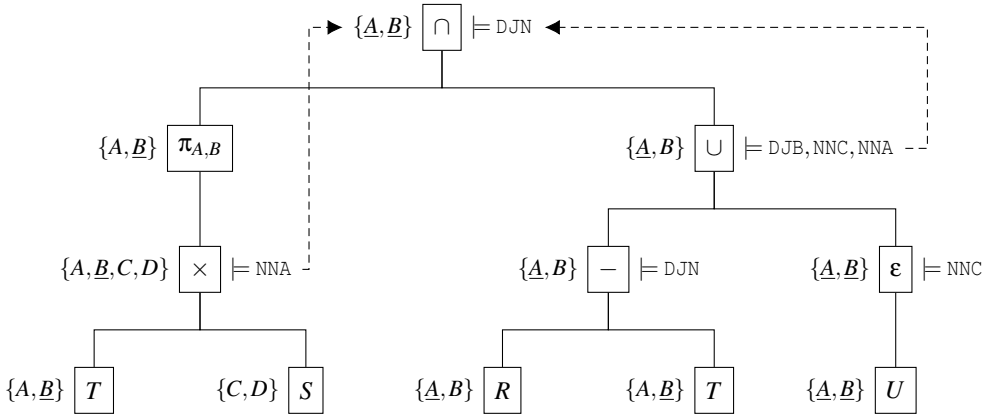


Figure 3: Syntax tree of the query $\pi_{A,B}(T \times S) \cap ((R-T) \cup \varepsilon(U))$. Each node is marked with the sufficient conditions it satisfies. The underlined attributes are non-nullable.

Chapter 3

Nullable Attributes of Selection

In chapter 2, we defined the nullable attributes of an expression $\sigma_\theta(E)$ to be the same as those of the expression E. However, this definition does not reflect the fact that, depending on the condition θ, the selection operation might remove all records r such that $r(A) \in \mathsf{Null}$ for some attribute $A \in \mathsf{sig}(\theta)$. In such case, A would effectively become non-nullable in the query answer. Take a query $\sigma_{\mathsf{const}(A)}(R)$ as an example. Regardless of whether the attribute A contains any nulls in the base relation R, an answer to $\sigma_{\mathsf{const}(A)}(R)$ is guaranteed not to have any nulls for the attribute A. In fact, in our query language there are two types of atomic conditions that ensure a "non-nullability" of the attribute they are applied to, i.e.: constant test and equality comparison (recall that: $[\![t_1 = t_2]\!]_r = \mathbf{t} \iff [\![t_1]\!]_r = [\![t_2]\!]_r \in \mathsf{Const}$).

Assessing if any attributes become non-nullable when the condition is not atomic is more complicated. For this investigation, let θ_A be a condition such that $\sigma_{\theta_A}(E)$ makes some attribute A non-nullable and let θ be any condition. Now, we can combine the two conditions using either the conjunction $\wedge$ or the disjunction $\vee$. Clearly, the condition $\theta_A \wedge \theta$ is at least as restrictive as θ_A, so $\sigma_{\theta_A \wedge \theta}(E) \subseteq \sigma_{\theta_A}(E)$. Hence, $\theta_A \wedge \theta$ ensures that attribute A is non-nullable as well. On the other hand, the condition $\theta_A \vee \theta$ may be weaker than θ_A so $\sigma_{\theta_A}(E) \subseteq \sigma_{\theta_A \vee \theta}(E)$. Therefore, in general, we cannot claim with certainty that $\sigma_{\theta_A \vee \theta}(E)$ makes the attribute A non-nullable. Nonetheless, if we knew that for θ, σ_θ makes the attribute A non-nullable too, then the result of the $\sigma_{\theta_A \vee \theta}(E)$ would not contain nulls for A as well. This is because $\sigma_{\theta_A \vee \theta}(E) \subseteq \sigma_{\theta_A}(E) \cup \sigma_\theta(E)$ and neither $\sigma_{\theta_A}(E)$ nor $\sigma_\theta(E)$ would have nulls for the attribute A.

To formalize the above observations, we define the set of all attributes that are certain to become non-nullable in the answer to the selection σ_θ.

Definition 4. The set $\mathsf{c\text{-}sig}(\theta)$ is a set of all attributes in the signature of the condition θ for the which the selection operation σ_θ is guaranteed to remove all records mapping any of these attributes to a null value. It is defined inductively as follows:

$$\mathsf{c\text{-}sig}(t_1 \neq t_2) \stackrel{\text{def}}{=\!=} \mathsf{c\text{-}sig}(\mathsf{null}(t)) = \varnothing$$

$$\mathsf{c\text{-}sig}(t_1 = t_2) \stackrel{\text{def}}{=\!=} \mathsf{sig}(t_1 = t_2)$$

$$\mathsf{c\text{-}sig}(\mathsf{const}(t)) \stackrel{\text{def}}{=\!=} \mathsf{sig}(\mathsf{const}(t))$$

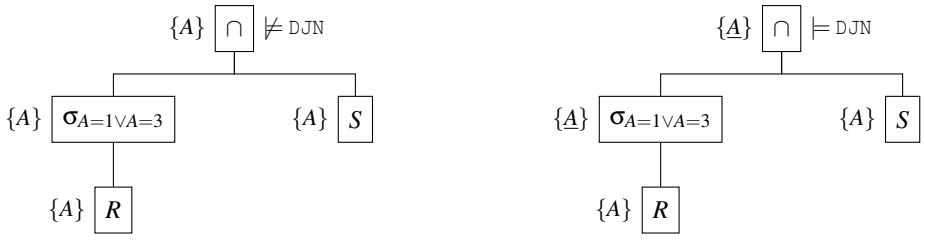
$$\text{c-sig}(\theta_1 \wedge \theta_2) \stackrel{\text{def}}{=\!=} \text{c-sig}(\theta_1) \cup \text{c-sig}(\theta_2)$$
$$\text{c-sig}(\theta_1 \vee \theta_2) \stackrel{\text{def}}{=\!=} \text{c-sig}(\theta_1) \cap \text{c-sig}(\theta_2)$$

Using this new concept we are able to incorporate the extra information derived from the condition into the definition of the nullable attributes of the selection expression, namely:

$$\text{n-sig}(\sigma_\theta(E)) \stackrel{\text{def}}{=\!=} \text{n-sig}(E) - \text{c-sig}(\theta) \tag{2}$$

This result is important not only in theory but also in practice. The more accurate analysis of the propagation of nullable attributes makes it easier to satisfy the conditions of the Theorem 1 since many of its conditions rely on the notion of nullable signature. Take query $Q = \sigma_{A=1}(R) \cap S$ as an example. Figure 4 depicts the propagation of nullable attributes in the query, on a database with relations R and S over a single nullable attribute A, using both definitions of n-sig. The most notable difference is that using the new definition we can say that the attribute A is non-nullable in the answer to the query. As a consequence, the $\cap$ node meets the DJN condition, which is required for Q to satisfy the premises of Theorem 1. Thus, without the new definition of $\text{n-sig}(\sigma_\theta(E))$ we would not be able to capture the preservation of Codd semantics in this query.

The key point to remember is that even though there is no condition for the selection node itself, its ability to narrow down the set of nullable attributes in the answer to the query can facilitate the satisfiability of DJN, NNC, and NNA conditions by other nodes in the syntax tree. For that reason, we will come back to the selection operation and its refined definition of nullable attributes in chapter 5 where we talk about transformations that produce equivalent queries which are more likely to be recognized as Codd semantics preserving.



(a) Using old definition: $\text{n-sig}(\sigma_{A=1})(R) = \text{n-sig}(R)$

(b) Using new definition: $\text{n-sig}(\sigma_{A=1\vee A=3})(R) = \text{n-sig}(R) - \{A\}$

Figure 4: Propagation of nullable attributes in the query $\sigma_{A=1\vee A=3}(R) \cap S$. The underlined attributes are non-nullable. Annotations in (a) use the old definition of $\text{n-sig}(\sigma_\theta)$ presented in [7]; in (b) use the new one described in this chapter. In (b), the new nullable signature is computed as follows: $\text{n-sig}(\sigma_{A=1\vee A=3}) = \text{n-sig}(R) - \text{c-sig}(A = 1 \vee A = 3) = \{A\} - (\text{c-sig}(A = 1) \cap \text{c-sig}(A = 3)) = \{A\} - (\{A\} \cap \{A\}) = \{A\} - \{A\} = \varnothing$.

Chapter 4

Chains of Intersections

Let us recall from [7] that, in general, whether we convert marked nulls into SQL nulls before or after an intersection makes a difference; i.e., $\mathsf{sql}(\llbracket Q_1 \cap Q_2 \rrbracket_D) \neq \llbracket Q_1 \cap Q_2 \rrbracket_{\mathsf{sql}(D)}$ (see Figure 5). This is because the intersection operation matches nulls syntactically. Two different marked nulls will not be matched, whereas their SQL counterparts will be considered the same.

$$\mathsf{sql}\left(\begin{array}{|c|}\hline \llbracket Q_1 \rrbracket_D \\ \hline A \\ \hline \bot_1 \\ \hline \end{array} \cap \begin{array}{|c|}\hline \llbracket Q_2 \rrbracket_D \\ \hline A \\ \hline \bot_2 \\ \hline \end{array}\right) = \varnothing \neq \begin{array}{|c|}\hline A \\ \hline \mathbf{N} \\ \hline \end{array} = \begin{array}{|c|}\hline \llbracket Q_1 \rrbracket_{\mathsf{sql}(D)} \\ \hline A \\ \hline \bot_1 \\ \hline \end{array} \cap \begin{array}{|c|}\hline \llbracket Q_2 \rrbracket_{\mathsf{sql}(D)} \\ \hline A \\ \hline \bot_2 \\ \hline \end{array}$$

Figure 5: Example illustrating that, in general, $\mathsf{sql}(\llbracket Q_1 \cap Q_2 \rrbracket_D) \neq \llbracket Q_1 \cap Q_2 \rrbracket_{\mathsf{sql}(D)}$.

For this reason, the DJN condition from Theorem 1 requires that the intersection operation is only applied to subqueries with disjoint sets of nullable attributes. Logically, if the two queries do not have nullable attributes in common, then the intersection cannot match any two records on the null value at all. This constraint ensures that $\mathsf{sql}(\llbracket Q_1 \cap Q_2 \rrbracket_D) = \mathsf{sql}(\llbracket Q_1 \rrbracket_D) \cap \mathsf{sql}(\llbracket Q_2 \rrbracket_D)$ which is then used to prove that such queries preserve Codd semantics [7].

Whilst the DJN condition is sufficient for a single intersection node on its own, it becomes unnecessarily restrictive as soon as more intersections are chained together. Indeed, there exist chains of intersections that preserve Codd semantics but for which the DJN condition cannot be satisfied by all of the $\cap$ nodes, and therefore the overall expression does not fulfil the requirements of Theorem 1. One such example is given in Figure 6.

Before jumping to the proposed solution, let us understand the implications of the current constraints imposed on individual nodes for the satisfiability of the conditions of Theorem 1 by chained intersections. For the purpose of this analysis, we adopt the following terminology.

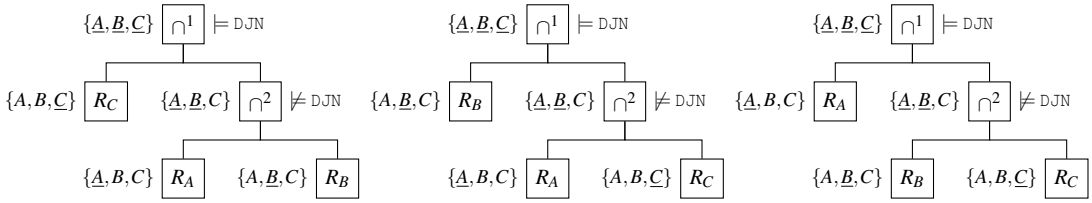


Figure 6: All three ways to intersect relations R_A, R_B, and R_C over attributes A,B,C, such that $\text{n-sig}(R_i) = \{i\}$, for $i \in \{A,B,C\}$. The underlined attributes are non-nullable. Note that no permutation of the operands makes the node $\cap^2$ satisfy the DJN condition, even though all queries do indeed preserve the Codd semantics.

Definition 5. (Chains of Intersections)

1. A *chain of intersections* is a query $Q_\cap$ complying with the grammar $Q_\cap := Q \cap Q$ and $Q := Q \cap Q \mid E$ where E is an expression that does not have $\cap$ as its root.
2. A *terminal intersection* in the chain of intersections is an intersection of the form $E \cap E$.

We refer to leaves of a chain of intersections as $E_1, \ldots, E_n$.

Now, regardless of the exact formulation of $Q_\cap$, at least one intersection in the query must be of the form $E_i \cap E_j$, for some $i, j \in \{1, \ldots, n\}$. For each such terminal intersection, $\text{n-sig}(E_i) \cap \text{n-sig}(E_j) = \varnothing$ becomes the necessary condition for $Q_\cap$ to meet the requirements of Theorem 1, as the $\cap$ node must satisfy the DJN condition. This insight led us to the following relationship:

Proposition 1. *Let $Q_\cap$ be a chain of intersections. Then, all intersections in the chain will satisfy the* DJN *condition if and only if all terminal intersections satisfy the* DJN *condition.*

The proof of the above proposition is given in Appendix A.

Combining Proposition 1 with the fact that for each chain of intersections there exists an equivalent chain that uses only one terminal intersection, we can state the condition for the existence of a chain satisfying the requirements of Theorem 1:

Corollary 1. *Given expressions $E_1, \ldots, E_n$, there always exists a chain of intersections with leaves E_1 to E_n which satisfies the premises of Theorem 1 if and only if:*

$$\exists\, E_p, E_q \in \{E_1, \ldots, E_n\} : \text{n-sig}(E_p) \cap \text{n-sig}(E_q) = \varnothing \tag{3}$$

In simple words, we can find a chain that intersects a given set of expressions and satisfies the requirements of Theorem 1 if and only if at least two of those expressions have no nullable attributes in common. This explains why in the example described in Figure 6 it is impossible to formulate a chain of intersections that would satisfy the premises of Theorem 1. Namely, the sets of nullable attributes of relations R_A, R_B, and R_C overlap with each other, making the condition (3) impossible to satisfy.

That said, three problems can be identified with this implicit requirement for the chain of intersections identified in Corollary 1:

1. When two or more queries are intersected together, it should be enough that for each attribute $A \in \text{sig}(E_1 \cap \cdots \cap E_n)$ there is an expression, E_A, which has the attribute A marked as non-nullable. Intuitively, if E_A does not contain any record with a null value for the attribute A, then the result of intersecting E_A with other queries cannot have such a record as well. The `DJN` condition captures this idea correctly for the binary case. However, for chains of intersections, the condition (3) is too strict as it ignores the fact that more than two subqueries can contribute to the final result of the overall intersection.
2. Neither the `DJN` condition nor the intersection operation itself captures information about other queries that take part in the entire intersection of multiple subqueries.
3. The satisfiability of the condition (3) merely indicates whether a query meeting the criteria of Theorem 1 exists. As indicated before, when testing an actual query for Codd semantics preservation, the satisfiability of the `DJN` conditions by all of the chained intersection nodes depends on the query's formulation. This is at odds with the fact that the intersection operation is commutative and associative. Ideally, all equivalent intersection chains should be equally likely to satisfy the premises of the theorem.

In what follows, we present our solution which addresses the described shortcomings. In section 4.1, we introduce a new variadic RA operation - the n-ary intersection. It enables us to express the intersection of multiple queries using a single operator, and thus contains information about its nullable attributes. In section 4.2, we adjust the `DJN` condition to the new operation and prove that the refined condition guarantees the Codd semantics preservation by the n-ary intersection.

4.1 Variadic intersection

Let us extend the language of relational algebra with a new operation: *n-ary intersection* $\bigcap(E_1, \ldots, E_n)$. It is well-defined with respect to a schema when:

- $E_1, \ldots, E_n$ are well-defined expressions;
- $\text{sig}(E_1) = \cdots = \text{sig}(E_n)$

The signature of a well-defined n-ary intersection $\bigcap(E_1, ..., E_n)$ is defined as follows:

$$\text{sig}\Big(\bigcap(E_1, \ldots, E_n)\Big) \stackrel{\text{def}}{=\!=} \text{sig}(E_1) = \cdots = \text{sig}(E_n)$$

The set of nullable attributes is given by:

$$\text{n-sig}\big(\bigcap(E_1, ..., E_n)\big) \stackrel{\text{def}}{=\!=} \bigcap_{i=1}^{n} \text{n-sig}(E_i)$$

The semantics of an n-ary intersection $\bigcap(E_1, \ldots, E_n)$ w.r.t. a database D is defined as follows:

$$\Big[\!\Big[\bigcap(E_1, \ldots, E_n)\Big]\!\Big]_D \stackrel{\text{def}}{=\!=} [\![E_1]\!]_D \cap \cdots \cap [\![E_n]\!]_D$$

Remark. The RHS of the previous definition is well-defined since the intersection operation is associative and commutative.

For notational convenience, we also define the n-ary intersection of bags $B_1, \ldots, B_n$ as $\bigcap(B_1, \ldots, B_n) = B_1 \cap \cdots \cap B_n$. Then, the following holds trivially:

$$\left[\!\left[\bigcap(E_1, \ldots, E_n) \right]\!\right]_D = \bigcap([\![E_1]\!]_D, \ldots, [\![E_n]\!]_D) \tag{4}$$

The definition of the syntax tree is also expanded to accommodate for the n-ary operator:

- For each operator $\bigcap(...)$, the syntax tree of $\bigcap(Q_1, ..., Q_n)$ has the root labelled with $\cap$ and syntax trees of $Q_1, ..., Q_n$ as its children. The order of operands is preserved, meaning that the syntax trees of Q_1 and Q_n are rooted as the leftmost and rightmost child nodes respectively.

All of the above definitions are generalizations of corresponding definitions for the binary intersection operation. For that reason, it should not be a surprise that queries $Q = Q_1 \cap Q_2$ and $Q' = \bigcap(Q_1, Q_2)$ are equivalent. As expected, the semantics of queries Q and Q', as well as their properties (e.g., the signature, the nullable signature, etc.), are the same without regard to whether the rules for the regular binary intersection or the n-ary intersection are used.

Below, we present relevant properties of the newly defined n-ary intersection which we will later use extensively in various proofs.

Theorem 2 (Algebraic Properties of the n-ary Intersection of Bags)**.** *Let $B_1, \ldots, B_m$, $B_{m+1}, \ldots, B_n$, for $2 \leq m < n$, be bags. Then:*

(a) The order of bags in the n-ary intersections does not affect the result;

(b) $\bigcap(B_1, B_1) = B_1$ - idempotent law;

(c) $\bigcap(B_1, \ldots, B_1, B_2, \ldots, B_n) = \bigcap(B_1, B_2, \ldots, B_n)$ - the result of an n-ary intersection where some bag appears more than once is equal to the result of an n-ary intersection where each bag appears only once;

(d) $\bigcap\left(\bigcap(B_1, \ldots, B_m), B_{m+1}, \ldots, B_n\right) = \bigcap(B_1, \ldots, B_n)$.

A proof of the theorem can be found in Appendix A.

Remark. Theorem 2 presents the properties of the n-ary intersection of bags. However, because of the relation (4), all of these properties can be lifted to the query level as long as an RA expression $\bigcap(\cdots)$ is well-defined.

Knowing the basic properties of the n-ary intersection, it should not be difficult to see that every chain of intersections $Q_\cap$ combining expressions E_1 to E_n is equivalent to the expression $\bigcap(E_1, \ldots, E_n)$. To put it in another way, regardless of the order in which we decide to intersect these expressions, the outcome will be always the same as if we combined them "simultaneously" using a single n-ary intersection $\bigcap(\cdots)$. This idea is formalized by Proposition 4 in Appendix B. The intuition is that each chain of intersections can be reduced to a single n-ary intersection by the means of Theorem 2(d).

4.2 Sufficient condition for the n-ary intersection

4.2.1 Refining the DJN condition

The fact that $\cap$ node can have multiple children in the syntax tree introduces an ambiguity in the definition of the DJN condition. Namely, for a node N with more than two children, the statement "children have no common nullable attributes" can have a twofold meaning:

1. $\forall X_i, X_j \in \{X | X \text{ is a child of } N\} : X_i \neq X_j \implies \text{n-sig}(X_i) \cap \text{n-sig}(X_j) = \varnothing$
2. $\bigcap_{i=1}^{n} \text{n-sig}(X_i) = \varnothing$, where $X_1, \ldots, X_n$ are all children of N

Both interpretations result in $\mathsf{n\text{-}sig}\big(\bigcap(X_1, \ldots, X_n)\big) = \varnothing$, but the first definition of the condition is much stricter than the second one. Therefore, the latter is more desirable as it can be satisfied by more intersection nodes.

Remark. Requiring an n-ary intersection node to satisfy the first interpretation would not enable us to detect more queries preserving Codd semantics. As all children of such node would have non-overlapping nullable attributes, every possible terminal intersection in the equivalent chain of binary intersections would satisfy the DJN condition. Consequently, by Proposition 1, all chains equivalent to such n-ary intersection would already meet the criteria of Theorem 1.

For the reasons outlined above, we decided to clarify the DJN condition in the refined syntax tree setup and define it as follows:

Definition 6. A node, N, in the syntax tree of a query satisfies the DJN condition if $\bigcap_{i=1}^{n} \text{n-sig}(X_i) = \varnothing$, where $X_1, \ldots, X_n$ are all children of N.

4.2.2 Proving that the refined DJN is a sufficient condition

Equipped with the new RA operator and the refined DJN condition we present extended conditions for the Codd semantics preservation:

Theorem 3. *Let Q be an RA query whose syntax tree is such that:*

a) *each* ε *node satisfies* NNC*;*

b) *each* $-$ *node satisfies* DJN*;*

c) *each* $\cap$ *node satisfies* DJN *(both binary & n-ary intersection);*

d) *each* $\times$ *node satisfies* NNA*;*

e) *each* $\cup$ *node satisfies* NNC *or* DJB *or* NNA*.*

Then, Q preserves Codd semantics.

To prove Theorem 3 which extends Theorem 1 with the n-ary intersection operator and its corresponding DJN condition, we first need to understand when a query Q is Codd semantics preserving. Let us recall that RA query Q preserves Codd semantics if for every Codd database D it holds that $\mathsf{sql}(\llbracket Q \rrbracket_D) = \llbracket Q \rrbracket_{\mathsf{sql}(D)}$ and $\llbracket Q \rrbracket_D$ is a Codd table.

Not surprisingly, the proof of Theorem 1 presented in [7] consists of two parts. One for each of the two properties (1a) and (1b). Both statements are proved by the induction on the structure of a relation algebra query. To prove Theorem 3, we extend the proofs of Theorem 1 to handle the case of the n-ary intersection. In order not to repeat the work from the original paper, here, we only present base cases of those inductions and complement their inductive steps with our proofs showing that properties (1a) and (1b) hold for the new n-ary intersection satisfying the refined `DJN` condition. For the full proof of Theorem 1 which we are extending, see [7].

To begin with, we notice that the n-ary intersection poses problems mainly with the property (1a). This is because, in general:

$$\mathsf{sql}(\llbracket \bigcap(Q_1,\ldots,Q_n) \rrbracket_D) \neq \llbracket \bigcap(Q_1,\ldots,Q_n) \rrbracket_{\mathsf{sql}(D)} \tag{5}$$

As it was the case with the binary intersection, the moment when we convert nulls into SQL nulls can impact the result (recall Figure 5). Therefore, we first introduce constraints that make sql operation distributive over the n-ary intersection:

Lemma 1. *Let $T_1,\ldots,T_n$ be tables with the same signature. Assume that, for every attribute A, there do not exist records $r_1,\ldots,r_n$ such that $r_i \in T_i$ and $r_i(A) \in \mathsf{Null}$, for $i = 1,\ldots,n$. Then, the following holds:*

$$\mathsf{sql}\big(\bigcap(T_1,\ldots,T_n)\big) = \bigcap(\mathsf{sql}(T_1),\ldots,\mathsf{sql}(T_n)) = \bigcap(T_1,\ldots,T_n)$$

Proof of the above Lemma can be found in Appendix A.

Remark. When the root of $Q = \bigcap(Q_1,\ldots,Q_n)$ satisfies the `DJN` condition, we have that $\bigcap_{i=1}^{n} \mathsf{n\text{-}sig}(Q_i) = \varnothing$. That is, for each attribute $A \in \mathsf{sig}(Q)$ there exists a table $T_A \in \{\llbracket Q_1 \rrbracket_D,\ldots,\llbracket Q_n \rrbracket_D\}$ such that $A \in \mathsf{c\text{-}sig}(T_A)$, which ensures that the assumptions of Lemma 1 are satisfied.

Equipped with Lemma 1, we can prove Theorem 3. We proceed in two parts. Each proves that any query satisfying the requirements of Theorem 3 satisfies the conditions (1a) and (1b) respectively.

Proof of condition (1a). Let Q be a query whose nodes in the syntax tree satisfy one of the conditions required by the theorem. Also, let D be any database. Using induction we prove that $\mathsf{sql}(\llbracket Q \rrbracket_D) = \llbracket Q \rrbracket_{\mathsf{sql}(D)}$.

Base case ([7]): Q is a relation name R. In such case, Q satisfies the condition as by the definition of $\mathsf{sql}(D)$: $\llbracket R \rrbracket_{\mathsf{sql}(D)} = \mathsf{sql}(\llbracket R \rrbracket)$

Inductive step: $Q = \bigcap(Q_1,\ldots,Q_n)$. Then:

$$\begin{aligned}
\mathsf{sql}\Big(\llbracket \bigcap(Q_1,\ldots,Q_n) \rrbracket_D\Big) &= \mathsf{sql}\Big(\bigcap\big(\llbracket Q_1 \rrbracket_D,\ldots,\llbracket Q_n \rrbracket_D\big)\Big) && \text{(by semantics)}\\
&= \bigcap\big(\mathsf{sql}(\llbracket Q_1 \rrbracket_D),\ldots,\mathsf{sql}(\llbracket Q_n \rrbracket_D)\big) && \text{(by Lemma 1)}\\
&= \bigcap\big(\llbracket Q_1 \rrbracket_{\mathsf{sql}(D)},\ldots,\llbracket Q_n \rrbracket_{\mathsf{sql}(D)}\big) && \text{(induction hypothesis)}\\
&= \Big\llbracket \bigcap(Q_1,\ldots,Q_n) \Big\rrbracket_{\mathsf{sql}(D)} && \text{(by semantics)}
\end{aligned}$$

For the remaining operations and conditions see the full proof of Theorem 1 in [7]. □

Proof of condition (1b). To satisfy this condition, we need to prove that for every Codd database D, $[\![Q]\!]_D$ is a Codd table. For that, let Q be a query whose nodes in the syntax tree satisfy one of the conditions required by the theorem. We proceed with the proof by induction as follows:

Base cases ([7]):

- Q is a relation name R. In such a case, $[\![Q]\!]_D$ is trivially a Codd table since D is a Codd database.
- Q is non-nullable. Then $[\![Q]\!]_D$ is a complete table which is also a Codd table.

Inductive step: If $Q = \bigcap(Q_1, \ldots, Q_n)$, then the DJN condition ensures that Q is non-nullable. Consequently, the new n-ary intersection is covered by the base case.

For the remaining operations and conditions see the full proof of Theorem 1 in [7]. □

Using Theorem 3 and the explicit n-ary intersection operation instead of chains of intersection we can express queries preserving Codd semantics which couldn't be recognized as such by the means of Theorem 1.

For instance, at the beginning of the chapter we showed an example with three relations R_A, R_B, and R_C over attributes A, B, C, such that $\mathsf{n\text{-}sig}(R_i) = \{i\}$, for $i \in \{A, B, C\}$. Because of Corollary 1 we claimed that there does not exist a chain of intersection that combines these relations and at the same time satisfies the premises Theorem 1 (see Figure 6). Having proved Theorem 3 this is no longer the case. The query $\bigcap(R_A, R_B, R_C)$ can be recognized as Codd semantics preserving because the $\cap$ node satisfies the DJN condition since $\mathsf{n\text{-}sig}(R_A) \cap \mathsf{n\text{-}sig}(R_B) \cap \mathsf{n\text{-}sig}(R_C) = \varnothing$.

Chapter 5

Transforming Queries to Capture Codd Semantics Preservation

The new variadic intersection together with the refined `DJN` condition enabled us to capture Codd semantics preserving intersections of multiple queries which would not otherwise be able to satisfy the conditions of Theorem 1. Moreover, satisfiability of Theorem 3 by the n-ary intersection does not depend on the order of intersected operands as is the case with equivalent chains of intersections. It might seem as if the presented solution resolves all the shortcomings of chained binary intersections. However, the introduction of the new operation did not change the fact that many queries containing chains of intersections still cannot be recognized as Codd semantics preserving on their own. Moreover, it is up to the author of a query to use the n-ary intersection and to use it properly. That is to ensure that the variadic intersection always represents the overall intersection and is not chained with other intersections.

It turns out that chains of intersections face exactly the same issues regardless of what operators they use. For example, an intersection of queries Q_1, Q_2, Q_3, Q_4 can be expressed using many equivalent queries:

- $\bigcap(Q_1, Q_2, Q_3, Q_4)$
- $\bigcap(\bigcap(Q_1, Q_2), Q_3, Q_4)$
- $\bigcap(Q_1, Q_2, Q_3) \cap Q_4$
- $\ldots$

Depending on the nullable attributes of queries Q_1 to Q_4, syntax trees of none, some, or all of the above intersections may meet the sufficient conditions for the preservation of Codd semantics. Most importantly, some query Q may be considered as Codd semantics preserving, whilst another equivalent query Q' is not. The issue here is the fact that the classification of whether a query is preserving Codd semantics, depends solely on its formulation. However, if Q and Q' are equivalent and Q preserves Codd semantics, then Q' must preserve Codd semantics as well.

Motivated by this observation, in section 5.1, we present and justify transformations that

output an equivalent syntax tree of query that is more likely to satisfy the conditions of Theorem 3. Proposed transformation rules are of the form $A \rightarrow B$ where A is a *template tree* and B is a *target tree*. A single application of a transformation rule consists of matching the syntax tree of an input query against the template of the rule and replacing the matched subtree with a target subtree. In sections 5.2 - 5.4, we prove for each transformation rule that if an input query satisfies the criteria of Theorem 3, then the output query satisfies them as well. We call such property *condition preservation*. Finally, in section 5.5, we combine findings from the previous sections and propose an algorithm that transforms a query into an equivalent query that is more likely to satisfy the conditions of Theorem 3.

5.1 Favourable transformations

We have already suggested that if a syntax tree of some query Q does not meet the conditions of Theorem 3, then one could look for an equivalent query that does meet the sufficient conditions. But exhaustive generation and analysis of all equivalent queries is impossible in general, so we need some heuristics to guide the search. In what follows, we present carefully motivated query rewrite rules that allow us to produce such equivalent queries.

5.1.1 Merging chained intersections

In the previous chapter, we showed that by using a single n-ary intersection $\bigcap(\cdots)$ one can capture more queries preserving Codd semantic than using a chain of intersections. The reasoning is that each terminal intersection node imposes an extra condition on the chain to satisfy the criteria of Theorem 3. Thus, having a single n-ary intersection representing the whole complex expression gives the best chance of satisfying the criteria. This insight can be used to find an equivalent query that is more likely to satisfy the sufficient conditions. Namely, we can merge chained intersections in a query to get an equivalent query in the intersection reduced form.

Definition 7. A query Q is in the *intersection reduced form* (IRF) if its syntax tree does not contain any chained intersections (binary nor n-ary).

Remark. Any query Q that is not in IRF is equivalent to at least one query in IRF.

The simplest syntax tree transformation rule that captures the idea of converting query into intersection reduced form is the Intersection Merge (IM) transformation rule presented in Figure 7. It merges two neighbouring intersections into a single n-ary intersection. The equivalence of input and output queries is guaranteed by Theorem 2(d).

Theoretically, even a single application of the rule increases the chances of satisfying the requirements of Theorem 3. Indeed, in the query on the left of Figure 7 both node $\cap_1$ and $\cap_2$ would need to satisfy `DJN`, which requires no common nullable attributes among subqueries $Q_1, \ldots, Q_n$, but also among subqueries $Q_1, \ldots, Q_k$, in particular. On the other hand, node $\cap_3$ on the right of Figure 7 only requires subqueries $Q_1, \ldots, Q_n$ not to share nullable attributes for node $\cap_2$ to satisfy `DJN`, which is clearly less restrictive.

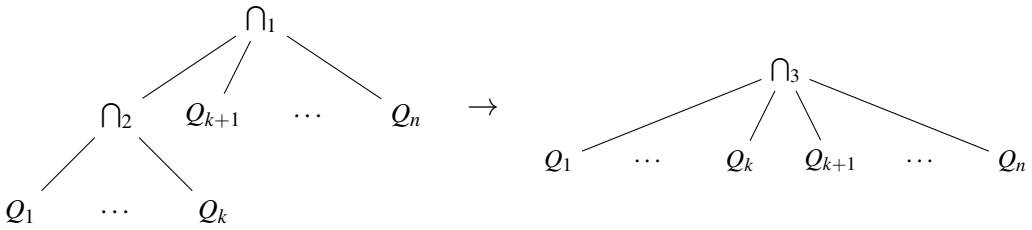


Figure 7: Intersection Merge (IM) transformation rule.

Practically, by Proposition 1, this makes a difference only when $\cap_2$ is a terminal intersection. However, this is not a problem because the goal is to merge all chained intersections nodes into a single node. The exact order in which this transformation happens does not matter. In addition, by not restricting the $\cap_2$ node to be a terminal intersection we keep the overall transformation rule more universal.

5.1.2 Bringing disconnected intersections together

Sometimes a few disconnected intersections can be represented as a single n-ary intersection. For example, queries $\bigcap(Q_1, \sigma_\theta(\bigcap(Q_2, Q_3)))$ and $\bigcap(Q_1, \sigma_\theta(Q_2), \sigma_\theta(Q_3))$ are equivalent, but the latter is more likely to satisfy the conditions of Theorem 3. Moreover, as demonstrated in chapter 3, even the order of other operations in some query Q can impact the satisfiability of the `DJN` condition by the intersection nodes. Thus, in this section, we present transformations that "normalize" the order of intersection, selection, and renaming operations in a query. Those transformations are based on distributive properties of selection and renaming operations over intersections.

For arbitrary queries Q_1 and Q_2, the selection and renaming operators distribute over binary intersections as follows:

$$\rho_{A\to B}(Q_1 \cap Q_2) = \rho_{A\to B}(Q_1) \cap \rho_{A\to B}(Q_2)$$

$$\sigma_\theta(Q_1 \cap Q_2) = \sigma_\theta(Q_1) \cap \sigma_\theta(Q_2) = \sigma_\theta(Q_1) \cap Q_2 = Q_1 \cap \sigma_\theta(Q_2)$$

The distributive properties over binary intersection naturally extend to the n-ary intersection operator:

$$\rho_{A\to B}(\bigcap(Q_1, \ldots, Q_n)) = \bigcap(\rho_{A\to B}(Q_1), \ldots, \rho_{A\to B}(Q_n)) \tag{6}$$

$$\sigma_\theta(\bigcap(Q_1, \ldots, Q_n)) = \bigcap(Q'_1, \ldots, Q'_n), \quad \text{where } Q'_i = \sigma_\theta(Q_i) \text{ or } Q_i \text{ for } i \in \{1, \ldots, n\} \text{ and at least one } Q'_i \text{ equals } \sigma_\theta(Q_i) \tag{7}$$

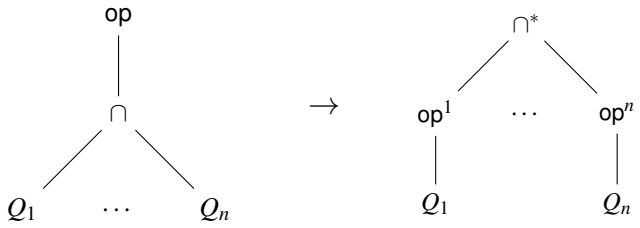
The outline of the proof for $\rho_{A\to B}(\bigcap(Q_1, \ldots, Q_n)) = \bigcap(\rho_{A\to B}(Q_1), \ldots, \rho_{A\to B}(Q_n))$ proceeds as follows:

By Proposition 4 we know that we can rewrite $\bigcap(Q_1, \ldots, Q_n)$ as $n-1$ binary intersections over $Q_1, \ldots, Q_n$. Then, we repetitively apply the distributive property over the binary intersections until we end up with $n-1$ intersections applied to $\rho_{A\to B}(Q_1), \ldots, \rho_{A\to B}(Q_n)$. Finally, due to Proposition 4, we know that we

can represent this chain of $n-1$ binary intersections using the n-ary operator as $\bigcap(\rho_{A\to B}(Q_1),\ldots,\rho_{A\to B}(Q_n))$ which completes the proof. The same reasoning holds for the distributive property of the selection over the n-ary intersection, but when propagating the selections over the binary intersections, at each step, we might propagate it to only one of the operands rather than to both of them.

Figure 8 presents the Renaming and Selection Propagation rules (RP / SP) that are used to push those operators down the syntax tree. Those transformations help to detect queries preserving Codd semantics in two ways. In the first place, by distributing selections over intersections, we can potentially restrict the set of nullable attributes in operands of the intersection (as shown in Figure 9). Moreover, after applying the transformation rules, the intersections which were previously interleaved with selections and renamings are now stacked all together in the syntax tree. As a result, these rules constitute a good normalization step before applying the IM transformation because they can increase the number of intersections that get merged (example in Figure 10). Each of these outcomes increases the chance that the intersection nodes will satisfy the DJN condition and, consequently, helps to create a query that is more likely to be recognised as Codd semantics preserving.

Now, even though the RP and SP transformations simply represent the properties (6) and (7), their exact form is carefully motivated. The distributive properties allow us to push the renaming and selection operations both up and down the syntax tree. In combination with the IM transformation, moving them in either direction enables us to capture more Codd semantics preserving queries. However, in order to maximize this number, we chose to propagate them toward the leaves of the syntax tree. There are two



for $\text{op} \in \{\rho, \sigma\}$

Figure 8: Renaming/Selection Propagation (RP / SP) transformation rules.

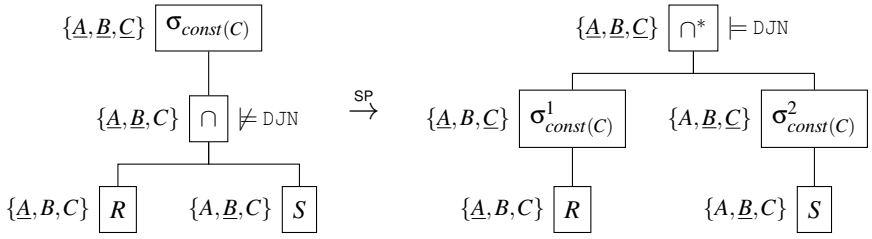


Figure 9: $\cap$ node in the LHS query does not satisfy the DJN condition. By distributing the selection over the intersection, the attribute C becomes non-nullable before the intersection is applied and $\cap^*$ satisfies the DJN in the transformed query. Consequently, the query after transformation can be recognized as Codd semantics preserving.

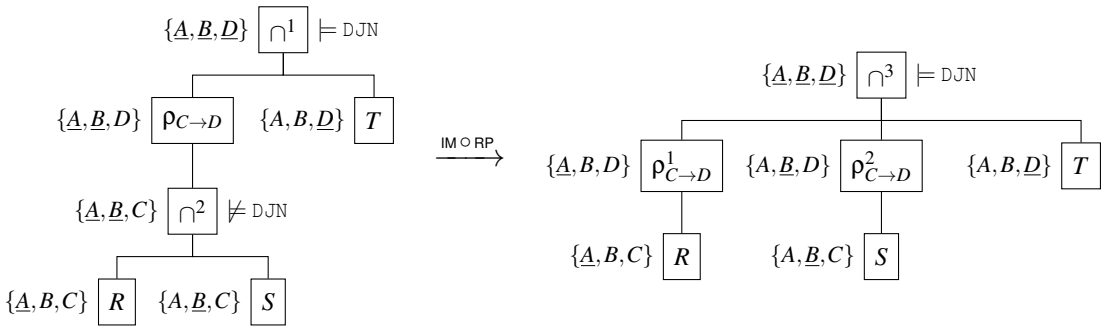


Figure 10: $\cap^2$ node in the LHS query does not satisfy the DJN condition. By distributing the renaming over the intersection, the two intersection operators can be merged producing the RHS query. Now, the original query can be recognised as Codd semantics preserving because the $\cap^3$ node in the transformed query satisfies the DJN condition.

reasons for that:

- To propagate the renaming operator towards the root, the same renaming operation must be applied to all operands of the intersection. On the other hand, there are no constraints on distributing the renaming over an intersection to all of its operands. Consequently, by pushing all selection and renaming operators down the syntax tree, more intersection operators can be chained together.
- As shown before, propagating selection to operands of an intersection increases the chances of satisfying the DJN condition by the intersection node. Doing the opposite decreases this chance.

5.1.3 Removing redundant intersections

In previous sections, the transformations aimed to increase the chances of satisfying the DJN condition by normalizing the query. Transformation rules presented in this section attempt to go a step further. In some cases, when operands of an intersection are identical, the operation can be removed altogether. The Intersection Simplification (IS) and Intersection Reduction (IR) transformation rules that capture this idea are presented

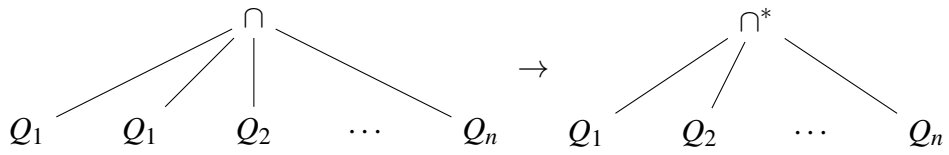


Figure 11: Intersection Simplification (IS) transformation rule.

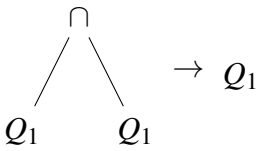


Figure 12: Intersection Reduction (IR) transformation rule.

in Figure 11 and Figure 12. The equivalence of the matched and target trees follows directly from Theorem 2(b) and (c).

One of the simplest examples that shows the usefulness of this transformation is a query $Q = \bigcap(R,R,R)$ which will be marked as Codd semantics preserving only when R is non-nullable. As soon as any of the attributes of R is nullable, the analysis of the query Q alone will fail to recognize it as Codd semantic preserving query. However, by transforming Q, using IS and IR rules, into an equivalent query $Q = \bigcap(R,R,R) \equiv \bigcap(R,R) \equiv R = Q'$ and then analysing Q' instead, we can determine that Q is indeed Codd semantics preserving regardless of what constraint are put on the attributes of R.

From the theoretical point of view, these transformation rules can be further generalised. Namely, we do not need to require that the repeating operands of an intersection are syntactically the same. In fact, we can perform either transformation as long as one of operands of the intersection in contained by the other. Specifically, if for some queries Q_1 to Q_n it is true that $Q_i \subseteq Q_j$ for some $i, j \in \{1,\ldots,n\}, i \neq j$, then it is the case that $\bigcap(Q_1,\ldots,Q_j,\ldots,Q_n) \equiv \bigcap(Q_1,\ldots,Q_{j-1},Q_{j+1},\ldots,Q_n)$ and $\bigcap(Q_i,Q_j) \equiv Q_i$. An example of the application of such generalized rule is presented in Figure 13.

As one could expect, proving query containment is much harder than detecting syntactically the same queries. In general, the query containment problem for two arbitrary relational algebra queries is undecidable for both set and bags semantics [13]. Although, this does not mean that one cannot take a pragmatic approach to detect contained queries. Recently, several projects described and implemented practical systems proving equivalence of numerous queries (a problem directly related to query containment by the fact that $Q_1 \equiv Q_2 \iff Q_1 \subseteq Q_2 \wedge Q_2 \subseteq Q_1$). Despite the same theoretical limitations, they can prove the equivalence of many pairs of arbitrary SQL queries using set and bag semantics. Examples of such systems are Cosette [2], UDP [3], EQUITAS [16], or SPES [15]. The successes of the aforementioned tools suggest that the generalized IS and IR transformation rules can have a practical application in the process of determining the preservation of Codd semantics.

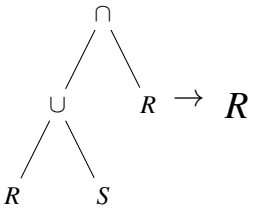


Figure 13: Example of an application of the generalized IR rule to a syntax tree rooted at $\cap$ node. Since $R \subseteq R \cup S$ under all databases, we can transform the original query, which requires the $\cap$ node to satisfy the DJN condition to be considered as Codd semantics preserving, into a much simpler query that trivially preserves the Codd semantics.

5.2 Transformations preserving sufficient conditions

Definition 8. Let ϕ be a transformation that takes a relational algebra query and produces an equivalent query. If $\phi(Q)$ satisfies the sufficient conditions for every query Q that satisfies them as well, then ϕ is *condition preserving*.

Transformations preserving sufficient conditions are of special interest to us. This is because they guarantee that we can detect all queries Q satisfying the sufficient conditions by checking the conditions in transformed queries. The relation between the sets of queries that are recognized to be Codd semantics preserving with and without the help of condition preserving transformation ϕ is following:

$$\{Q \mid \phi(Q) \text{ satisfies Theorem 3 } \} \supseteq \{Q \mid Q \text{ satisfies Theorem 3}\} \tag{8}$$

Using a transformation to detect Codd semantics preserving queries that could be recognized as such without transforming them in the first place is not that useful. However, if we show that there exists a query Q such that $\phi(Q)$ satisfies the premises of Theorem 3 while Q does not satisfy the sufficient conditions, then:

$$\{Q \mid \phi(Q) \text{ satisfies Theorem 3 } \} \supset \{Q \mid Q \text{ satisfies Theorem 3}\} \tag{9}$$

In simple words, if the relation (9) holds, then by checking the conditions in transformed queries we can detect more queries preserving Codd semantics.

Unfortunately, not all transformations are condition preserving. Even transformations producing equivalent queries are not guaranteed to be condition preserving. A simple counterexample is a transformation taking a query Q and returning the query $Q \cap Q$. Such transformation is not condition preserving as the $\cap$ node is not guaranteed to satisfy the `DJN` condition for all Codd semantics preserving queries Q. Besides, proving the condition preservation for an arbitrary transformation can be a non-trivial task. In fact, we tried it, unsuccessfully, for a more complex algorithm producing an equivalent RA query in the intersection reduced form. The main obstacle was to formally reason about conditions when the transformation affected multiple parts of the syntax tree.

This motivated us to come up with the ”atomic” transformation rules presented before. Their simplicity facilitates the analysis of condition preservation, as all the syntax tree changes are local to the transformed subtree. Furthermore, despite using only simple syntax tree rewrite rules, we are still able to express more complex transformations. For example, the transformation of an RA query to the intersection reduced form can be achieved by repeatedly applying the IM rule until there are no more neighbouring intersections. Also, composing atomic transformations makes it easier to analyse the condition preservation of the resulting complex transformations. Intuitively, if all composed transformations are condition preserving then the resulting transformation is condition preserving as well. This idea is captured in Proposition 2.

Proposition 2. *Let $\phi = \phi_n \circ \cdots \circ \phi_1$. If ϕ_1 to ϕ_n are all condition preserving then ϕ is condition preserving as well. Moreover, if any of the composed transformations satisfies the relation (9), then it holds for ϕ too.*

Proof. Both statements follow directly from the transitivity of relations (8) and (9). □

We have already shown in section 5.1 that with the help of our transformation rules we can identify new Codd semantics preserving queries. In the following sections, we prove that all of the presented transformation rules preserve the DJB, NNA, DJN, and NNC conditions.

5.3 Sufficient conditions for condition preservation

Before we prove that proposed rules are condition preserving, we want to make a few observations about what parts of a query's syntax tree are relevant for each condition and how changes to a syntax tree structure can impact the condition preservation. The purpose of this subsection is to create a unified framework that can be later used to evaluate whether a transformation rule is condition preserving.

5.3.1 Terminology

Recall that given a transformation rule ϕ we can apply it to a query Q to produce an equivalent query $\phi(Q)$. If some subtree of the syntax tree of Q matches the template tree of ϕ, we call such subtree a *matched subtree*. In such a case, $\phi(Q)$ is created by replacing the matched subtree with the target tree of the transformation rule.

To make the discussion more concise, the terms "query", "syntax tree of the query", and "node" (when referring to the specific node in a syntax tree) are used interchangeably. We allow for this ambiguity as each query is uniquely represented by its syntax tree or by a syntax tree rooted at the particular node. Moreover, on top of the normal rules for labelling nodes in the syntax tree, we require that node labels are unique in both template and target trees of a transformation rule as well as in Q and $\phi(Q)$. This way we can uniquely refer to parts of the query before and after the transformation. The exception to this rule is a situation when a syntactically identical subquery is present in a query before and after the transformation (it was unaffected by the transformation). In such a case, the corresponding nodes of this subquery can have the same labels in both trees and we say that these nodes are the same. We illustrate and explain the labelling rules in Figure 14.

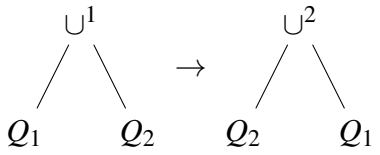


Figure 14: Example of labelling rules used in a transformation rule. Nodes Q_1 and Q_2 represent any two potentially different queries. Since they are present in both the template and the target tree, we know they represent the same subquery (i.e., Q_1 in the template tree is the same as Q_1 in the target tree). On the other hand, union $\cup^2$ in the target tree, could not be labelled as $\cup^1$ because it represents a syntactically different query (the order of operands is changed).

Also, to make the discussion about condition preservation clearer, we can identify the following elements within the matched and target trees:

Roots (of matched and target trees) - these two nodes always have different labels as they represent two different queries. From now on, we refer to them as R_m and R_t nodes respectively.

Ancestors (of transformation) - ancestors of R_m and R_t. Note that the transformation rule cannot change the structure of a syntax tree of a query Q above the root of the matched tree. Therefore, for every ancestor, A, of R_m in Q, there exists a corresponding ancestor, A_ϕ, of R_t in $\phi(Q)$ so that the path between the root of the respective query and the two ancestors is the same. Naturally, A and A_ϕ always represent the same type of operation. For notational convenience, we use subscript ϕ to represent such a corresponding node in $\phi(Q)$. Note that ancestors of transformation could not be labelled the same, as they have two different queries rooted at them.

Remark. When a transformation rule is applied to some query, roots of matched and target trees may or may not have ancestors (e.g. when the rule is applied to the root of a query). For the sake of generality, when proving the condition preservation, we will assume that roots always have some ancestors.

Leaves (of matched and target trees) - they are usually labelled Q with some subscript. They represent subqueries rooted at them (we will refer to them as ***subqueries of the transformation***). The transformation ϕ does not alter those subqueries in any way.

Inodes (of matched and target trees) - internal nodes present in matched and target trees respectively.

Inserted inodes - inodes present in a target tree but not in a matched tree.

Removed inodes - inodes present in a matched tree but not in a target tree.

Remark. Due to the way we label/identify nodes in a transformation rule, inodes of a target tree consist only of inserted nodes and non-removed inodes.

Unrelated nodes - nodes that are neither descendants nor ancestors of R_m or R_t. Queries rooted at them are not modified by the transformation in any way. Thus, two corresponding unrelated nodes N in Q and N_ϕ in $\phi(Q)$ must represent the same subquery.

5.3.2 Transformation's impact on satisfiability of the conditions

Sufficient conditions for Codd semantics preservation can be split into two categories concerned with:

- a base of a query - `DJB`; and
- nullable attributes of a query - `DJN`, `NNC`, `NNA`.

To prove that the transformation is condition preserving, we need to show that all nodes in the transformed query satisfy one of the required conditions, regardless of conditions satisfied by each node in the initial query. To facilitate this task, we split the nodes in $\phi(Q)$ into five groups depending on their position in the transformed syntax tree. Table 1 introduces the node groups as well as summarizes the impact a transformation could have on nodes' properties. In the following sections, we explain how each property is (or is not) affected by transformation rules presented in this report and outline what suffices

Type of nodes	Can be altered by a transformation?			
	n-sig	base	path to an ancestor	n-sig of an ancestor
ancestors of the transformation	yes	yes	no	yes
non-removed inodes	no	no	yes	yes
inserted inodes	n/a	n/a	n/a	n/a
nodes in subqueries of the transformation	no	no	yes	yes
unrelated nodes	no	no	no	yes

Table 1: Nodes in a transformed query can be split into five groups based on their position in a syntax tree. Table presents the impact a transformation could have on the various node properties and related to them conditions (n-sig: `DJN`, `NNC`, `NNA`; base: `DJB`; path to an ancestor: `NNA`; n-sig of an ancestor: `NNA`)

to be shown to ensure that a rule is condition preserving. We will omit the inserted nodes in our considerations as it is impossible to talk about condition preservation in their case. Instead, their satisfiability of the sufficient conditions needs to be proven separately for each transformation rule.

5.3.3 Preservation of the DJB condition

Since the base of a query is defined inductively with respect to its children, the bases of all subqueries not modified by the transformation remain the same. Such subqueries are represented by nodes that have the same label in the original and transformed query. That is: non-removed nodes, unrelated nodes, and nodes in subqueries of the transformation. Therefore, if one of these nodes satisfied the `DJB` condition before the transformation, it will continue to do so after it. For that reason, as shown in Table 1, we only need to ensure that ancestors of the transformation preserve the `DJB` condition, as their base can be affected by the transformation. The condition which guarantees this is following:

$$\mathsf{base}(R_t) \subseteq \mathsf{base}(R_m) \tag{10}$$

Proof. Let A be the ancestor of R_m such that A satisfies the `DJB` condition. Also, let A^l and A^r be the children of A. Following our notation, A^l_ϕ and A^r_ϕ are the corresponding children of A_ϕ. Without the loss of generality, we can assume that R_m is a descendant of A^l. In such a case, A^r is not modified by the transformation and is the same as A^r_ϕ.

Next, let V be the set of relation names that appear in the subtree rooted at A^l excluding the subtree rooted at R_m. Note that V is equal to the set of relation names that appear in the subtree rooted at A^l_ϕ excluding the subtree rooted at R_t. This is because A^l_ϕ can be created by replacing R_m with R_t in A^l. Therefore, we can express bases of A^l and A^l_ϕ as:

$$\mathsf{base}(A^l) = V \cup \mathsf{base}(R_m) \tag{11}$$

$$\mathsf{base}(A^l_\phi) = V \cup \mathsf{base}(R_t) \tag{12}$$

Finally, we can use the above observations to proof that A_ϕ satisfies the DJB condition as well:

$$\begin{aligned}
\mathsf{base}(A^l_\phi) \cap \mathsf{base}(A^r_\phi) &= \mathsf{base}(A^l_\phi) \cap \mathsf{base}(A^r) && \text{(because } A^r = A^r_\phi\text{)} \\
&= (V \cup \mathsf{base}(R_t)) \cap \mathsf{base}(A^r) && \text{(by 12)} \\
&\subseteq (V \cup \mathsf{base}(R_m)) \cap \mathsf{base}(A^r) && \text{(condition 10)} \\
&\subseteq \mathsf{base}(A^l) \cap \mathsf{base}(A^r) && \text{(by 11)} \\
&= \varnothing && \text{(because } A \models \texttt{DJB}\text{)}
\end{aligned}$$

□

Remark. We prove the preservation of the DJB condition only for nodes with two children as it is only required from the binary union operation.

5.3.4 Preservation of the NNA, DJN, and NNC conditions

Similarly to the base property, changes in nullable attributes of the transformed subquery can propagate only towards the root of the input query, as the nullable signature is defined recursively with respect to children of a given node. Using this observation, we formulate the following lemma which we will then use to prove the preservation of NNA, DJN, and NNC conditions.

Lemma 2. *Let ϕ be a transformation rule ϕ with R_m and R_t being the roots of the matched and target tree respectively. For any two corresponding ancestors of the transformation, A in Q and A_ϕ in $\phi(Q)$, the following statement holds:*

$$\mathsf{n\text{-}sig}(R_t) \subseteq \mathsf{n\text{-}sig}(R_m) \implies \mathsf{n\text{-}sig}(A_\phi) \subseteq \mathsf{n\text{-}sig}(A)$$

Proof. In chapter 2, we showed the recursive definition of the nullable signature of a RA expression. Using those formulas, we can represent nullable attributes of a complex query as a function of a nullable signature of its sub-expression. Assuming that a query Q' is a child of Q and that the nullable attributes of all the other children of Q are fixed, let function $f_{Q' \to Q}(N)$ represent the nullable signature of Q if $\mathsf{n\text{-}sig}(Q')$ was equal to N. For example, let $Q = Q_1 \cap Q_2$, with $\mathsf{n\text{-}sig}(Q_1) = \{A, B\}$. Then, $f_{Q_2 \to Q}(N) = N \cap \{A, B\}$. What is more, observe that by definition: $f_{Q' \to Q}(\mathsf{n\text{-}sig}(Q')) = \mathsf{n\text{-}sig}(Q)$. That is, if we assume that the nullable signature of Q' is equal to the actual nullable signature of Q', then $f_{Q' \to Q}(\mathsf{n\text{-}sig}(Q'))$ is equal to the actual nullable signature of Q.

The analysis of the propagation of nullable attributes naturally extends beyond the scope of a direct parent-child relation. Given nodes Q_1, Q_2, Q_3 such that $Q_1 \prec Q_2 \prec Q_3$, we can represent the impact of nullable attributes of Q_3 on the nullable signature of Q_1 as a function $f_{Q_2 \to Q_1}(f_{Q_3 \to Q_2}(N))$. For notational convenience, we will write:

$$f_{Q_n \to Q_1}(N) = (f_{Q_2 \to Q_1} \circ f_{Q_3 \to Q_2} \circ \cdots \circ f_{Q_n \to Q_{n-1}})(N) \tag{13}$$

for nodes Q_1 to Q_n such that $Q_1 \prec \cdots \prec Q_n$.

The last bit needed to prove the lemma is the fact that for any ancestor node A of R_m and its child A', such that $A' \prec \cdots \prec R_m$ or $A' = R_m$, it is the case that

$$f_{A' \to A}(N) = f_{A'_\phi \to A_\phi}(N) \tag{14}$$

This follows from the fact that a transformation can only impact the nullable attributes of nodes on the path from the root of the query to the node R_m. Other children of A are equal to their corresponding nodes in $\phi(Q)$, so their nullable signature is unchanged. Hence, the nullable attributes fixed by $f_{A' \to A}$ are the same as those fixed by $f_{A'_\phi \to A_\phi}$ and the two functions are equal.

Finally, we can assemble all the insights and proceed with the final proof. Let:

$$X = \mathsf{n\text{-}sig}(R_m) \qquad Y = \mathsf{n\text{-}sig}(R_t) \qquad \text{and} \qquad Z = \mathsf{n\text{-}sig}(R_m) - \mathsf{n\text{-}sig}(R_t),$$

then the set of nullable attributes of A can be expressed as follows:

$$\begin{aligned}
\mathsf{n\text{-}sig}(A) &= f_{R_m \to A}(\mathsf{n\text{-}sig}(R_m)) && \text{(by definition of } f_{R_m \to A}) \\
&= f_{R_m \to A}(X) \\
&= f_{R_m \to A}(Y \cup Z) \\
&= f_{R_m \to A}(Y) \cup f_{R_m \to A}(Z) && \text{(property of a map on sets)} \\
&= f_{R_t \to A_\phi}(Y) \cup f_{R_m \to A}(Z) && \text{(by relation 14)} \\
&= f_{R_t \to A_\phi}(\mathsf{n\text{-}sig}(R_t)) \cup f_{R_m \to A}(Z) \\
&= \mathsf{n\text{-}sig}(A_\phi) \cup f_{R_m \to A}(Z) && \text{(by definition of } f_{R_t \to A_\phi})
\end{aligned}$$

which implies that $\mathsf{n\text{-}sig}(A_\phi) \subseteq \mathsf{n\text{-}sig}(A)$ and completes the proof. □

Equipped in Lemma 2, we can find sufficient conditions for the preservation of `DJN`, `NNC`, and `NNA` conditions.

5.3.4.1 Preservation of the DJN and NNC conditions

Similarly to the base of a query, the query's nullable attributes are defined inductively with respect to its children. Therefore, nullable signatures of all nodes that are not modified by the transformation will remain unchanged. As a consequence, non-removed nodes, unrelated nodes, and nodes in subqueries of the transformation which satisfied the `DJN` or `NNC` condition in the original query will continue to do so in the transformed one. Therefore, to prove that a transformation rule preserves the `DJN` and `NNC` conditions, we only need to show that those conditions are preserved among the ancestors of the transformed tree. The following condition ensures that this is indeed the case:

$$\mathsf{n\text{-}sig}(R_t) \subseteq \mathsf{n\text{-}sig}(R_m) \tag{15}$$

Proof. Let A be an ancestor of R_m such that A satisfies the `DJN` condition. Also, let A^1 to A^n be the children nodes of A. Since A satisfies the `DJN`, we know that

$\bigcap_{i=1}^{n} \text{n-sig}(A^i) = \varnothing$. Now, without the loss of generality we can assume that R_m is a descendant of A^1. Assuming that $\text{n-sig}(R_t) \subseteq \text{n-sig}(R_m)$, we proceed as follows:

$$\begin{aligned}
\bigcap_{i=1}^{n} \text{n-sig}(A_i^{\phi}) &= \text{n-sig}(A_1^{\phi}) \cap \bigcap_{i=2}^{n} \text{n-sig}(A_i^{\phi}) \\
&= \text{n-sig}(A_1^{\phi}) \cap \bigcap_{i=2}^{n} \text{n-sig}(A_i) && \text{(other children are unaffected)} \\
&\subseteq \text{n-sig}(A_1) \cap \bigcap_{i=2}^{n} \text{n-sig}(A_i) && \text{(by Lemma 2)} \\
&\subseteq \bigcap_{i=1}^{n} \text{n-sig}(A_i) = \varnothing && \text{(because } A \models \texttt{DJN}\text{)}
\end{aligned}$$

Hence, the DJN condition is satisfied for all ancestors of R_t that satisfied the DJN in the input query. The proof for NNC is analogous. □

5.3.4.2 Preservation of the NNA condition

Recall that NNA can be satisfied in two ways. Either the node is non-nullable or one of its ancestors is. We will prove both cases in one go. We start by assuming, without the loss of generality, that:

- NNA condition is satisfied by a node N in a syntax tree of a query Q.
- M is the non-nullable ancestor of N in Q or M is N itself.
- N is a leaf node in the syntax tree of Q. If the NNA condition is preserved for a leaf, then it must be preserved for all nodes between the node M and the leaf N.

At this point, there are two scenarios. Either N is a node in a subquery of the transformation or not. We start by assuming the former. In such a case, let Q_i be a leaf node of the matched tree such that $Q_i = N$ or $Q_i \prec \cdots \prec N$. At this point, we assume that Q_i is a leaf of the target tree as well. If it was not the case, then N would not be a part of the transformed query and there would be nothing to preserve.

Then, we have three cases:

1. M is in the subquery rooted at Q_i. Under such circumstances, the NNA condition is trivially preserved as the subquery rooted at Q_i is not modified by the transformation.
2. M is an inode of the matched tree. Then we need to show that there exists another non-nullable inode in the target tree, call it M', that is an ancestor of Q_i. By the transitivity of the ancestor relation, M' will be an ancestor of N and the NNA condition will be preserved.
3. M is the ancestor of R_m. Consequently, M_ϕ is an ancestor of R_t, which is an ancestor of N. In this case, it is sufficient to show that $\text{n-sig}(R_t) \subseteq \text{n-sig}(R_m)$. If this condition is met, then, by Lemma 2, $\text{n-sig}(M_\phi) \subseteq \text{n-sig}(M) = \varnothing$ as M is non-nullable. By the transitivity of the ancestor relation, M_ϕ is a non-nullable ancestor of N in $\phi(Q)$ and the NNA condition is preserved.

Now, if N is not in any subquery of the transformation, then we have two more cases:

4. M is not an ancestor of R_m (M is a unrelated node). Under such circumstances, the corresponding node M_ϕ in $\phi(Q)$ is equal to M and the NNA condition is trivially preserved. This is because the transformation could not alter the nullable attributes of M, so M is still the non-nullable ancestor of N.
5. M is an ancestor of R_m. Again, by requiring that $\mathsf{n\text{-}sig}(R_t) \subseteq \mathsf{n\text{-}sig}(R_m)$, we ensure that $\mathsf{n\text{-}sig}(M_\phi) \subseteq \mathsf{n\text{-}sig}(M) = \varnothing$ as M is non-nullable. Consequently, the M_ϕ node is a non-nullable ancestor of N and the NNA condition is preserved.

In the first and fourth cases, the NNA is trivially preserved. Therefore, to show that a transformation rule preserves the NNA condition it is sufficient to prove that:

- $\mathsf{n\text{-}sig}(R_t) \subseteq \mathsf{n\text{-}sig}(R_m)$; and
- for every inode x in the matched tree we need to show that: if we assume x to be non-nullable, then, for every leaf Q_d of the matched tree that has x as an ancestor, there exists a non-nullable inode in the target tree that is an ancestor of Q_d.

5.3.5 Sufficient conditions for condition preservation - summary

All things considered, we just showed that when proving the condition preservation of a transformation rule, we only need to focus on the requirements related to inodes of matched and target subtrees of the rule. Specifically:

- for DJB we need to show that: $\mathsf{base}(R_t) \subseteq \mathsf{base}(R_m)$
- for DJN & NNC we need to show that: $\mathsf{n\text{-}sig}(R_t) \subseteq \mathsf{n\text{-}sig}(R_m)$
- for NNA we need to show that:
  - $\mathsf{n\text{-}sig}(R_t) \subseteq \mathsf{n\text{-}sig}(R_m)$; and
  - for every inode x in the matched tree we need to show that: if we assume x to be non-nullable, then, for every leaf Q_d of the matched tree that has x as an ancestor, there exists a non-nullable inode in the target tree that is an ancestor of Q_d.
- for every inserted node we need to show that it satisfies at least one condition required by its expression type, regardless of what conditions are satisfied by nodes in the matched tree

Remark. None of the transformation rules presented in this report adds a base relation that is not already present in $\mathsf{base}(R_m)$ to the target tree, thus we consider all presented transformations to be DJB preserving.

5.4 Proofs of condition preservation

In the next subsections, we formally prove that all transformation rules presented in this report are condition preserving.

5.4.1 Intersection Merge rule

Keeping in mind our findings from previous sections, to prove that the IM transformation rule is condition preserving we will show that:

- $\cap_3$ satisfies the DJN condition - because $\cap_3$ is an inserted node.
- $\cap_3$ is non-nullable - for the preservation of the NNA condition (so all nodes in a transformed subtree can refer to it as their non-nullable ancestor).
- $\text{n-sig}(\cap_3) \subseteq \text{n-sig}(\cap_1)$ - for the preservation of the DJN and NNA conditions.

As a matter of fact, all of these conditions come down to showing that $\text{n-sig}(\cap_3) = \varnothing$. This is because for any intersection operation $Q = \bigcap(Q_1, \ldots, Q_n)$ it is the case that $\text{n-sig}(Q) = \varnothing \iff \bigcap_{i=1}^{n} \text{n-sig}(Q_i) = \varnothing$ and the empty set is a subset of every set so the last condition is satisfied as well.

We proceed by noticing that $\text{n-sig}(\cap_2) = \bigcap_{i=1}^{k} \text{n-sig}(Q_i) = \varnothing$ since $\cap_2$ must satisfy the DJN condition. We use this observation to prove that $\cap_3$ is non-nullable as follows:

$$\text{n-sig}(\cap_3) = \bigcap_{i=1}^{n} \text{n-sig}(Q_i) = \bigcap_{i=1}^{k} \text{n-sig}(Q_i) \cap \bigcap_{i=k+1}^{n} \text{n-sig}(Q_i) = \varnothing \cap \bigcap_{i=k+1}^{n} \text{n-sig}(Q_i) = \varnothing$$

Thus, the IM transformation rule is condition preserving.

5.4.2 Selection/Renaming Propagation rules

To prove that SP/RP rules are condition preserving we will show that:

- $\cap^*$ node satisfies the DJN condition - because $\cap^*$ is an inserted node.
- $\cap^*$ node is non-nullable - for the preservation of the NNA condition.
- $\text{n-sig}(\cap^*) \subseteq \text{n-sig}(\text{op})$ - for the preservation of the DJN and NNA conditions.

Again, all of these come down to showing that $\text{n-sig}(\cap^*) = \varnothing$.

We start the proof by noticing that $\text{n-sig}(\cap) = \bigcap_{i=1}^{n} \text{n-sig}(Q_i) = \varnothing$ as the $\cap$ node must satisfy the DJN condition. Using this observation we prove that $\text{n-sig}(\cap^*) = \varnothing$ for both operations.

For the selection operation, $\text{n-sig}(\sigma^i_\theta(Q_i)) \subseteq \text{n-sig}(Q_i)$ since selection can only remove attributes from the nullable signature. Hence, we get that:

$$\text{n-sig}(\cap^*) = \bigcap_{i=1}^{n} \text{n-sig}(\sigma^i_\theta(Q_i)) \subseteq \bigcap_{i=1}^{n} \text{n-sig}(Q_i) = \varnothing$$

Similarly, for the renaming operation:

$$\begin{aligned}\text{n-sig}(\cap^*) &= \bigcap_{i=1}^{n} \text{n-sig}(\rho^i_{A\to B}(Q_i)) = \bigcap_{i=1}^{n} \text{n-sig}(Q_i)[A/B] = \Big(\bigcap_{i=1}^{n} \text{n-sig}(Q_i)\Big)[A/B] \\ &= \varnothing[A/B] = \varnothing\end{aligned}$$

Therefore, SP and RP rules are condition preserving.

5.4.3 Intersection Simplification rule

To prove that the IS rule is condition preserving we will show that:

- $\cap^*$ satisfies the DJN condition - because $\bigcap^*$ is an inserted node
- $\cap^*$ is non-nullable - for the preservation of the NNA condition
- $\text{n-sig}(\cap^*) \subseteq \text{n-sig}(\cap)$ - for the preservation of the DJN and NNA conditions

Once again, all of these reduce to showing that $\text{n-sig}(\cap^*) = \varnothing$, which is the case as:

$$\text{n-sig}(\cap^*) = \bigcap_{i=1}^{n} \text{n-sig}(Q_i) = \text{n-sig}(Q_1) \cap \bigcap_{i=1}^{n} \text{n-sig}(Q_i) = \text{n-sig}(\cap) = \varnothing \qquad (\text{as } \cap \models \texttt{DJN})$$

which proves that IS is a condition preserving transformation rule.

5.4.4 Intersection Reduction rule

The IR rule is unusual because its subquery of transformation Q_1 becomes the root of the transformed subtree. This, however, does not change the way in which we prove the condition preservation for the IR rule. The things we will show this time are the following:

- $\text{n-sig}(Q_1) \subseteq \text{n-sig}(\cap)$ - for the preservation of the DJN and NNA conditions
- Q_1 is non-nullable - for the preservation of the NNA condition (by nodes in the query rooted at Q_1)

Once more, both of these conditions are equivalent because $\text{n-sig}(\cap) = \varnothing$ (as the intersection node must satisfy the DJN condition) requiring $\text{n-sig}(Q_1)$ to be an empty set for IR to be condition preserving. We proceed with the proof as follows:

Since $\cap$ node satisfies the DJN condition, we know that:

$$\text{n-sig}(\cap) = \text{n-sig}(Q_1) \cap \text{n-sig}(Q_1) = \varnothing$$

However:

$$\text{n-sig}(Q_1) \cap \text{n-sig}(Q_1) = \varnothing \iff \text{n-sig}(Q_1) = \varnothing$$

Hence, Q_1 must be non-nullable, which proves that the IR transformation rule is condition preserving.

5.5 Proposed query preprocessing step

Finally, we present the transformation that can be used to capture more queries preserving Codd semantics:

Definition 9. ϕ_{IRF} is a transformation which takes as an input a relational algebra query Q and outputs an equivalent RA query in the intersection reduced form. It transforms the query in three phases:

1. **Distribution phase**: all selections and renamings are distributed over intersections according to the SP and RP transformation rules.
2. **Merging phase**: each chain of intersections is reduced to a single n-ary intersection according to the IM rule.
3. **Reduction phase**: First, all duplicates of intersections' operands are removed according to the IS rule. Then, any intersection that has the same subquery as its operands is replaced by that subquery according to the IR rule. Simplifying/reducing all children of a node before the node itself is processed guarantees that the syntax tree is simplified as much as possible.

It should be clear from the definition of ϕ_{IRF} that every phase of the transformation is equivalent to repeated applications of respective transformation rules. As each of these rules produces an equivalent query, the result of the overall transformation is equivalent to the original query. Consequently, this allows us to arrive at the following proposition.

Proposition 3. *Let Q be a relational algebra query. If the syntax tree of $\phi_{IRF}(Q)$ satisfies the conditions of Theorem 3, then Q preserves Codd semantics.*

In order to ensure that the new method of testing queries for the preservation of Codd semantics is practical, we must require that a transformation of the original query terminates in the polynomial time with respect to the number of nodes in the syntax tree of the input query. N.B. deciding whether a query satisfies the premises of Theorem 1 and 3 can be performed in the linear time with respect to the size of its syntax tree [7]. As one could expect, this is indeed the case for the ϕ_{IRF} transformation (see Appendix C for the supporting argument), which brings us to the final conclusion of this report:

Theorem 4. *Proposition 3 enables us to capture more Codd semantics preserving queries than Theorem 3 alone.*

Proof. By definition, ϕ_{IRF} is always equivalent to some composition of SP, RP, IM, IS, and IR transformations, all of which are condition preserving. Moreover, in section 5.1, we have already demonstrated that there exist queries for which $\text{SP}(Q_1)$, $\text{IM}(Q_2)$, and $\text{IR}(Q_3)$ satisfy the premises of Theorem 3, while queries Q_1, Q_3, Q_3 on their own do not. Meaning that the relation (9) holds for the SP, IM, and IR transformation rules. Thus, by Proposition 2, we get that:

$$\{Q \mid \phi_{\text{IRF}}(Q) \text{ satisfies Theorem 3 } \} \supset \{Q \mid Q \text{ satisfies Theorem 3}\}$$

which is the same as:

$$\{Q \mid Q \text{ satisfies Proposition 3 } \} \supset \{Q \mid Q \text{ satisfies Theorem 3}\}$$

□

Chapter 6

Testing Codd Semantics in Java

As a part of the project, we developed an open-source Java library, *coddifier* [11], which enables its clients to verify whether an RA expression is guaranteed to preserve Codd semantics. The library exposes its API through two methods in `Coddifier` class:

- `isGuaranteedToPreserveCoddSemanticsAsIs` checks whether a query is guaranteed to preserve Codd semantics by analyzing its syntax tree.
- `isGuaranteedToPreserveCoddSemantics` first normalizes the query using ϕ_{IRF} transformation and then analyzes the syntax tree of the transformed expression.

6.1 Coddifier's architecture

The system architecture consists of three main building blocks: RA `Expressions`, `ExpressionTransformations`, and database `Schemas`. Their relation with each other is depicted in Figure 15. To use the `Coddifier` class, a client needs to represent a query as an `Expression` object and pass a `Schema` with respect to which the query is to be tested for Codd semantics preservation. We provide basic implementations of these interfaces so the library can be used independently with a minimal amount of supporting code. Moreover, the simplicity of the required interface facilities seamless integration with existing software that already defines its structures for expressions and schemas. In the following subsections, we will briefly describe the core components of the *coddifier* package.

Figure 15: The overview of the system architecture. The dashed lines represent the dependencies between the modules.

6.1.1 RA expressions

In *coddifier* library, RA expressions are represented by subclasses of the `Expression` class. The base class uses a template method pattern to test for Codd semantics preservation in which classes representing individual expression types must implement `computeSignature`, `computeNullableSignature`, `isWellDefined`, and `satisfiesSufficientConditions` functions. The verification procedure follows the algorithm described in [7], with exception that all conditions are checked for each syntax tree node.

Currently, we support testing of RA queries under bags semantics as described in Theorem 3. Atomic expressions are implemented by the `Relation` class and complex expressions by `Difference`, `Distinct`, `Intersection`, `Product`, `Projection`, `Renaming`, `Selection`, and `Union` classes. However, extending this set with new operations and their corresponding sufficient conditions can be easily achieved by inheriting from `Expression` or other expression classes. For example, the projection expression under set semantics is required to satisfy the `NNA` condition [7]. To reflect this in our system, the client can extend the `Projection` class in the following manner:

```
class SetProjection extends Projection {
    protected boolean satisfiesSufficientConditions() {
        return nna;  // where "nna" is a boolean flag set to true
                     // if the node satisfies the NNA condition
    }
}
```

6.1.2 Database schemas

Sufficient conditions can guarantee that a query will preserve Codd semantics when evaluated on a database instance, or rather, a set of instances that match some database schema. Hence, we need to be able to capture information about relations and their attributes in a database. We do this by passing to the system an implementation of the `Schema` interface:

```
public interface Schema {
    boolean hasRelation(String relation);
    Set<String> getRelationAttributeNames(String relation);
    Set<String> getRelationNullableAttributeNames(String relation);
}
```

Except for the consistent implementation of the above functions (i.e., for every relation R in database schema $\mathsf{sig}(R) \neq \varnothing$ and $\mathsf{n\text{-}sig}(R) \subseteq \mathsf{sig}(R)$), we also require that objects implementing `Schema` interface are immutable. This is because they are used as keys when caching attributes of each subexpression, which avoids unnecessary recomputation of those properties every time they are used during the condition checking process.

6.1.3 Expression transformations

`ExpressionTransformations` are used by the `Coddifier` class to transform the original expression into an equivalent expression that is more likely to satisfy the sufficient

conditions. At the moment, the input query is preprocessed using the ϕ_{IRF} transformation. As proved in section 5.5, it is guaranteed to produce an equivalent query and to preserve sufficient conditions. To maintain these correctness guarantees, we do not allow clients to specify their transformations to be used by the `Coddifier` class. However, nothing prevents users from transforming queries on their own before testing them for Codd semantics preservation. Once new transformations are shown to facilitate the detection of Codd semantics preserving queries, they will be incorporated into the verification process so that everyone can benefit from them.

6.2 Evaluation

The correctness of the implementation is verified using 171 unit and integration tests providing 100% line and branch coverage. While the unit tests focus on the individual functions ensuring proper functioning of all subcomponents, the integration tests verify the correct behaviour of the exposed API as a whole. The latter mainly includes: testing queries for the preservation of Codd semantics, checking if individual nodes satisfy appropriate conditions; ensuring that transformations produce expected results.

6.3 Integration with RA Parser

The *coddifier* package on its own enables testing of queries represented only as `Expression` objects. While in some cases it might be acceptable to manually hardcode a predetermined list of queries using appropriate classes in a Java program, ideally, we would like users to be able to specify queries using some user-friendly interface.

However, we deliberately decided not to deliver any interactive user interface, nor to provide a way to build these queries dynamically. There are several reasons for that. Most notably, we wanted to keep the *coddifier* package simple. If we added RA parsing module, we would also need to add the support for creating the database schemas, handling input errors, supporting derived operations, etc. All of a sudden, the package would have many responsibilities that are not directly related to the problem of Codd semantics preservation. By focusing on one thing only, we can: keep the code simple, maintainable, and well-tested; minimize the number of dependencies and the overall size of the package; make the library highly customisable and extendable; have a clearly defined purpose. All of the above are the desirable properties of a software library which make it more likely to be used by other software.

Moreover, there already exist open-source RA interpreters, such as *real* [4] or *RA* [14], that do much more than just parse RA queries. We believe that the *coddifier* package fits best on top of such systems extending their functionality with the ability to check queries for Codd semantics preservation. Having said that, to make the functionality of the library accessible to the broader community, we integrated the *coddifier* library with the command line RA interpreter *real*. We hope that it will officially become a part of *real* in the near future. In the meantime, the fork implementing the changes is available at [8] (branch 'develop'). Examples of how to check queries for Codd semantics preservation using *coddifier* library alone and using the *real* tool are presented in Appendix D.

Chapter 7

Conclusions and Future Work

This report set out a number of improvements that facilitate recognition of Codd semantics preserving queries to techniques described in [7].

Firstly, in chapter 3, we refined the way nullable attributes are propagated in selection by deriving information about which attributes are certain to be made non-nullable by the operation from its condition. Even though there are not any conditions put directly on the selection node, this change can facilitate satisfiability of `DJN`, `NNC`, and `NNA` constraints by other nodes in the syntax tree.

Secondly, in chapter 4, we observed that the original `DJN` condition was ill-suited to capture the Codd semantics preservation of chains of intersections. For that reason, we introduced a variadic intersection operation that makes it possible for us to express the intersection of all subqueries at once. This enabled us to apply the redefined `DJN` constraint in a way that imposes the desired state of the query answer on the overall intersection, rather than requiring it from each intermediate binary intersection. The described solution fits into the general observation that introducing derived operations to the syntax of the query language allows to relax the sufficient conditions for the underlying operations [7]. The downside of this approach is that queries must use the new constructs explicitly and properly to see any benefits. This might not always be achievable, e.g., if queries are automatically generated using the standard set of RA operators.

To overcome this problem, in chapter 5, we suggested a query normalization procedure that transforms an input query into an equivalent query that is more likely to satisfy the sufficient conditions. Thanks to this method we can incorporate the derived operations in the testing process even if the original query does not include them in the first place. Moreover, we showed that the syntax tree rewrite rules involved in the transformation are conditions preserving. As a result, by preprocessing the query using the ϕ_{IRF} transformation we can detect more queries preserving Codd semantics than using Theorem 3 alone.

Furthermore, we came up with sufficient conditions for the preservation of the conditions. Interestingly, transformation rules do not always need to preserve all the conditions. For example, if queries of interest never contained a union operation, then

we would not need to prove the preservation of the DJB condition as it would never be required by any of the syntax tree nodes. This could allow for the creation of transformation rules tailored to the problem at hand.

Finally, as described in chapter 6, we implemented the checks for the preservation of Codd semantics as a Java library - *coddifier*. To the best of our knowledge, this is the first implementation of the work described in [7] and this report. Moreover, to make the package more accessible and easier to use for the broader community, we added the functionality of the *coddifier* system as a feature in the command line RA interpreter *real*.

7.1 Plan for MInf Project (Part 2)

This project has many extensions, which we are planning to pursue in its second part. One possibility is to identify new derived operations that may result in weaker constraints. Our candidate is a variadic union operation. Namely, we observed that in the binary case, we either consider the base or the nullable attributes of the union's operands. In the n-ary case, we could look at the two properties at the same time. Intuitively, nullable records from relations in non-nullable children cannot propagate to the query answer. Therefore, we should be able to exclude the non-nullable operands of the n-ary from the DJB check - potentially relaxing the constraints. Another option is to check whether the newly derived operations (e.g., the variadic intersection) help to relax sufficient conditions for queries evaluated under set semantics.

The second direction of research is to identify other syntax tree transformations facilitating satisfiability of the sufficient conditions. We showed before that the order of operations can impact the nullable attributes of individual nodes. By distributing operations that can narrow down the nullable signature of other nodes (such as selections or projections) we should be able to detect even more queries preserving Codd semantics.

One more interesting extension is to attempt to evaluate datasets of queries written in a suitable fragment of SQL. For that, we could use translations from SQL to RA presented in [6] and test the translated queries for Codd semantics preservation. Naturally, the satisfiability of restrictions will depend on the exact translations used, but we hope that our normalization step will mitigate this problem to a certain degree. A more challenging continuation is to understand Codd semantics preservation in the context of aggregations and groupings. This would enable us to express and verify even more queries that frequently appear in real life.

Finally, we would like to investigate a database model (suggested in [7]) in which duplicates of nullable records are allowed. Note that this was not possible in Codd databases as Codd nulls cannot repeat. In this scenario, a table could be represented as a set of records, each associated with its multiplicity in the given table. Such a setup poses many interesting questions: Can we even talk about the preservation of Codd semantics in that context? If yes, how can duplicated nullable records be interpreted in SQL databases? Are the corresponding restriction weaker or stronger than those required for the preservation of Codd semantics in databases where only constant records can repeat? We will try to answer these questions in the second part of the project.

Bibliography

[1] Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[2] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1591–1594, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. *Proc. VLDB Endow.*, 11(11):1482–1495, jul 2018.

[4] Paolo Guagliardo. real. `https://git.ecdf.ed.ac.uk/pguaglia/real`, 2019. Retrieved April 6, 2022.

[5] Paolo Guagliardo and Leonid Libkin. Correctness of sql queries on databases with nulls. *SIGMOD Rec.*, 46(3):5–16, oct 2017.

[6] Paolo Guagliardo and Leonid Libkin. A formal semantics of sql queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, sep 2017.

[7] Paolo Guagliardo and Leonid Libkin. On the codd semantics of SQL nulls. *Inf. Syst.*, 86:46–60, 2019.

[8] Paolo Guagliardo and Konrad Pijanowski. real-codd-preservation. `https://git.ecdf.ed.ac.uk/s1863753/real-codd-preservation`, 2022. Retrieved April 6, 2022.

[9] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, page 233–246, New York, NY, USA, 2002. Association for Computing Machinery.

[10] Witold Lipski. On relational algebra with marked nulls preliminary version. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '84, page 201–203, New York, NY, USA, 1984. Association for Computing Machinery.

[11] Konrad Pijanowski. coddifier. `https://github.com/kopi22/coddifier`, 2022. Retrieved April 6, 2022.

[12] Peter Storeng. Implementing marked nulls in postgresql. Master's thesis, University of Edinburgh, 2016.

[13] Boris A Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70(4):569–572, 1950.

[14] Jun Yang. RA. `https://github.com/junyang/RA`, 2014. Retrieved April 6, 2022.

[15] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. A symbolic approach to proving query equivalence under bag semantics. *arXiv preprint arXiv:2004.00481*, 2020.

[16] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. Automated verification of query equivalence using satisfiability modulo theories. *Proc. VLDB Endow.*, 12(11):1276–1288, jul 2019.

Appendix A

Additional Proofs

Proof of Proposition 1. If all intersections in the chain satisfy the DJN condition, then, trivially, all terminal intersections satisfy the DJN condition as well. Thus, we focus on the proof in the other direction.

We start by noticing that each intersection node in the chain, call it N, is either a terminal intersection or one of its children contains some terminal intersection. In the former case, N satisfies the DJN by the assumption.

In the latter, let $N \prec \cdots \prec T$, where T is a terminal intersection node. Since all nodes on the path between N and T are labelled $\cap$ and $\mathsf{n\text{-}sig}(Q_1 \cap Q_2) = \mathsf{n\text{-}sig}(Q_1) \cap \mathsf{n\text{-}sig}(Q_2)$ implies that $\mathsf{n\text{-}sig}(Q_1 \cap Q_2) \subseteq \mathsf{n\text{-}sig}(Q_1)$ and $\mathsf{n\text{-}sig}(Q_1 \cap Q_2) \subseteq \mathsf{n\text{-}sig}(Q_2)$, it holds that:

$$\mathsf{n\text{-}sig}(N) \subseteq \cdots \subseteq \mathsf{n\text{-}sig}(T)$$

As T satisfies the DJN condition, $\mathsf{n\text{-}sig}(T) = \varnothing$ which in turn means that $\mathsf{n\text{-}sig}(N)$ must be an empty set. Hence, N must satisfy the DJN condition as well.

□

Proof of Theorem 2. All the properties are derived directly from the definition of the n-ary bag intersection:

(a) As the the binary intersection is commutative and associative, the n-ary intersection is equal to:

$$\bigcap(B_1,\ldots,B_n) = B_1 \cap \cdots \cap B_n = \{ \underbrace{r,\ldots,r}_{\min(k_1,\ldots,k_n)\text{ times}} \mid r \in_{k_1} B_1, \ldots,\ r \in_{k_n} B_n \}$$

Since the reordering of bags does not change the counts of elements in bags, we can conclude that the ordering of bags in the n-ary intersection operation does no matter for the final result.

(b)

$$\bigcap(B_1,B_1) = B_1 \cap B_1 = B_1$$

(c)

$$\begin{aligned}\bigcap(B_1,\ldots,B_1,B_2,\ldots,B_n) &= B_1\cap\cdots\cap B_1\cap B_2\cap\cdots\cap B_n\\ &= (B_1\cap\cdots\cap B_1)\cap B_2\cap\cdots\cap B_n\\ &= B_1\cap B_2\cap\cdots\cap B_n\\ &= \bigcap(B_1,\ldots,B_n)\end{aligned}$$

(d)

$$\begin{aligned}\bigcap\big(\bigcap(B_1,\ldots,B_m),B_{m+1},\ldots,B_n\big) &= \bigcap(B_1,\ldots,B_m)\cap B_{m+1}\cap\cdots\cap B_n\\ &= \big(B_1\cap\cdots\cap B_m\big)\cap B_{m+1}\cap\cdots\cap B_n\\ &= B_1\cap\cdots\cap B_m\cap B_{m+1}\cap\cdots\cap B_n\\ &\qquad\qquad\text{(by the associativity)}\\ &= \bigcap(B_1,\ldots,B_n)\end{aligned}$$

□

Proof of Lemma 1. We assumed that there do not exist records $r_1,\ldots,r_n$ such that, for every attribute A, $r_i\in T_i$ and $r_i(A)\in\mathsf{Null}$, for $i=1,\ldots,n$. Hence, it is impossible for some record r taking a nullable value for some attribute to be present in all the tables $T_1,\ldots,T_n$ at the same time. Thus, a result of the n-ary intersection $\bigcap(T_1,\ldots,T_n)$ must be a complete table. Hence, trivially, $\bigcap(T_1,\ldots,T_n)=\mathsf{sql}\big(\bigcap(T_1,\ldots,T_n)\big)$

Now, we can show that $\bigcap(T_1,\ldots,T_n)=\bigcap\big(\mathsf{sql}(T_1),\ldots,\mathsf{sql}(T_n)\big)$ as follows:

$$\begin{aligned}\#(r,\bigcap(T_1,\ldots,T_n)) &= \#(r,\ T_1\cap\cdots\cap T_n)\\ &= \min\big\{\#(r,T_1),\ldots,\#(r,T_n)\big\}\\ &\overset{(\bullet)}{=} \min\big\{\overbrace{\sum_{s\in\mathsf{sql}^{-1}(r)}\#(s,T_1)}^{k_1},\ldots,\overbrace{\sum_{s\in\mathsf{sql}^{-1}(r)}\#(s,T_n)}^{k_n}\big\}\\ &= \min\big\{\#(r,\mathsf{sql}(T_1)),\ldots,\#(r,\mathsf{sql}(T_n))\big\}\\ &= \#(r,\bigcap(\mathsf{sql}(T_1),\ldots,\mathsf{sql}(T_n)))\end{aligned}$$

All equalities, but $(\bullet)$, follow directly from the definition of the n-ary intersection and the sql operation. To prove that equality $(\bullet)$ holds, we have to consider two cases: (1) when r maps all attributes to constant values, (2) when r maps some attribute A to a nullable value.

For (1): the equality holds because, in such case, $\mathsf{sql}^{-1}(r)=\{r\}$.

For (2): since $\bigcap(T_1,\ldots,T_n)$ is complete and $r(A)\in\mathsf{Null}$, then $\#(r,\bigcap(T_1,\ldots,T_n))=0$. By our assumption, we know that there are not any records $r_1,\ldots,r_n$ such that $r_i\in T_i$ and $r_i(A)\in\mathsf{Null}$, for $i=1,\ldots,n$. Because of that, at least one of the sums $k_1,\ldots,k_n$ must be equal to 0 (as otherwise s would contradict the assumption of the lemma) and thus the RHS of $(\bullet)$ is also equal to 0. □

Appendix B

Equivalence of the n-ary Intersection and Chains of Intersections

The goal of this appendix is to prove the following proposition:

Proposition 4. *Let E be any complex RA expression whose syntax tree meets the following conditions:*

- *the root and all internal nodes are labelled $\bigcap$*
- *leaf nodes, $X_1, \ldots, X_n$, are roots of any well-formed expressions*

Then, E is equivalent to the expression $\bigcap(X_1, \ldots, X_n)$

To do that, we will first prove a related lemma which will greatly simplify the proof of Proposition 4.

Lemma 3. *Let E be an RA expression whose syntax tree satisfies the following conditions:*

- *all internal nodes, if any, are labelled $\bigcap$;*
- *all leaf nodes, $X_1, \ldots, X_n$ are roots of any well-formed expression*

Then E is equivalent to the expression $\bigcap(X_1, X_1, X_2, X_2, \ldots, X_n, X_n)$.

Proof. We give an inductive proof on the structure of the assumed RA expression.

Basis: The base case is when E is any expression other than an intersection, so its syntax tree has a single leaf node (the root node), meaning that $E = X_1$. In such case, $E = \bigcap(X_1, X_1)$, so the basic case holds.

Induction: The inductive step applies when the root node of the syntax tree of E is labeled with $\bigcap$. In such case, $E = \bigcap(E_1, ..., E_k)$, for some $k \geq 2$. By the induction hypothesis, each sub-expression E_i, for $i \in \{1, \ldots, k\}$, has an assumed syntax tree with n_i leaf nodes labeled $X_i^1, \ldots, X_i^{n_i}$ and can be represented as $\bigcap(X_i^1, X_i^1, \ldots, X_i^{n_i}, X_i^{n_i})$. Thus:

$$E = \bigcap(E_1, ..., E_k) \equiv \bigcap\left(\bigcap(X_1^1, X_1^1, \ldots, X_1^{n_1}, X_1^{n_1}), \ldots, \bigcap(X_k^1, X_k^1, \ldots, X_k^{n_k}, X_k^{n_k})\right) \tag{16}$$

Now, using Theorem 2(d), we can merge nested intersections in the equation (16) to get:

$$\begin{aligned} E &\equiv \bigcap \left(\bigcap (X_1^1, X_1^1, \ldots, X_1^{n_1}, X_1^{n_1}), \ldots, \bigcap (X_k^1, X_k^1, \ldots, X_k^{n_k}, X_k^{n_k}) \right) \\ &\equiv \bigcap \left(X_1^1, X_1^1, \ldots, X_1^{n_1}, X_1^{n_1}, \ldots, \bigcap (X_k^1, X_k^1, \ldots, X_k^{n_k}, X_k^{n_k}) \right) \\ &\equiv \ldots \\ &\equiv \bigcap \left(X_1^1, X_1^1, \ldots, X_1^{n_1}, X_1^{n_1}, \ldots, X_k^1, X_k^1, \ldots, X_k^{n_k}, X_k^{n_k} \right) \end{aligned}$$

Finally, note that the leaf nodes of the syntax trees of E_1 to E_k constitute all of the leaves of the syntax tree of E - as required by the induction hypothesis. This is because E_1 to E_k are all children of E. This observation completes the inductive step and concludes the proof. □

Eventually, equipped with Lemma 3, we can prove Proposition 4.

Proof of Proposition 4. Let E be an expression such that its syntax tree meets the assumption of the proposition:

- the root and all internal nodes are labelled $\bigcap$
- leaf nodes, $X_1, \ldots, X_n$, are roots of any well-formed expression

Since E satisfies the assumptions of Lemma 3, the expression $\bigcap(X_1, X_1, ..., X_n, X_n)$ is equivalent to E. Using Theorem 2(c) we can remove duplicated leaf nodes from the n-ary intersection to get:

$$E \equiv \bigcap (X_1, X_1, ..., X_n, X_n) \equiv \bigcap (X_1, ..., X_n)$$

□

Appendix C

Time Complexity of $\phi_{\mathbf{IRF}}$ Transformation

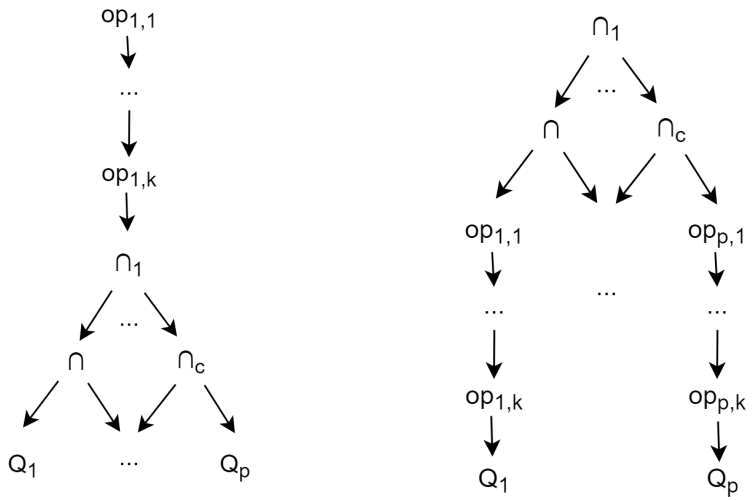
The purpose of this appendix is to present an argument that the time complexity of the ϕ_{IRF} transformation is polynomially bounded with respect to the size of the input query, i.e., the number of nodes in its syntax tree.

We do so by showing that the time complexity of each phase (distribution, merging, and reduction) is $O(n^{O(1)})$, where n is the number of nodes in the syntax tree of the original input query. As a result, we claim that the time taken by the overall transformation is:

$$O(n^{O(1)}) + O(n^{O(1)}) + O(n^{O(1)}) = O(n^{O(1)})$$

C.1 Time complexity of the distribution phase

The problem of distributing selections and renamings over intersections in an arbitrary query Q can be reduced to solving a problem of distributing selections/renamings over



(a) Before transformation (b) After transformation

Figure 16: Transformation of the local chain of intersections.

local chains of intersections in the input query. In this analysis, we consider a single intersection to be a chain of intersections as well. Now, let us assume that k ops $\in \{\sigma, \rho\}$ are to be distributed over a chain of c intersections with p leaves, for $k, c \geq 1$ and that the parent of the root of such subtree is different than a selection or renaming node. Clearly, $k + c + p \leq n$ as all the nodes making up the chain must have already existed in the input query. Figure 16a demonstrates the considered chain of intersections.

After the local transformation, we end up with a subtree that has no ops at the top. Instead, there are $k * p$ of them at the bottom of the chain (see Figure 16b). We can construct such tree by propagating each op_i, for $i = 1, \ldots, k$, to every leaf of the intersection chain. While the exact time taken to propagate an op to a single leaf depends on the tree representation and the algorithm used, it can be surely done in $O(n^{O(1)})$ time. Hence, to propagate k op nodes to p leaves, thus to transform a single local chain of transformations, it takes

$$k * p * O(n^{O(1)}) = n * n * O(n^{O(1)}) = O(n^{O(1)})$$

time, as $k * p \leq n * n$.

Moreover, we notice that each transformation of a local chain of intersections cannot produce any new chains. This is because:

- Subqueries Q_1 to Q_p cannot have an $\cap$ node as the root (by the definition of the intersection chain) so the propagated ops cannot create any new chains which would require further transformation.
- If the root of any subquery Q_i, for $i = 1, \ldots, p$, is a σ or ρ node, then the propagated ops could extend an already existing chain of intersections, not create a new one.
- If op_1 has no parent, then there is nothing left to distribute over the chain rooted at the $\cap_1$ node in the transformed chain of intersections.
- If the parent of op_1 is an intersection node then the chain of intersections rooted at $\cap_1$ in the transformed subtree extends the already existing chain of intersection.
- If the parent of op_1 is any other node (but σ and ρ which it cannot be by our initial assumption) then the new parent of $\cap_1$ cannot be propagated over the intersection so a new chain is not created.

Furthermore, we know that there must be fewer than n chains of intersections as there are less than n intersection nodes in the input query. Combining all the findings together, we get that the time complexity of the overall transformation is:

$$n * O(n^{O(1)}) + n * O(n^{O(1)}) = O(n^{O(1)})$$

where the first term represents the transformation time of at most n chains of intersections and the second term is the time needed to find each chain, which certainly can be done in polynomial time.

C.2 Time complexity of the merging phase

The merging phase can be completed in the polynomial time with respect to the size of its input simply by traversing the syntax tree once and merging the neighbouring intersection nodes.

Having said that, the input of the merging phase might be different than the input to the overall transformation. This is because the query might have already been modified by the distribution phase. However, we know that size of the new input has to be polynomially bounded with respect to the size of the original input as the distribution step terminates in the polynomial time.

As a result, the time complexity of the merging phase is also polynomially bounded with respect to the original input since:

$$O\big(O(n^{O(1)})^{O(1)}\big) = O(n^{O(1)*O(1)}) = O(n^{O(1)})$$

C.3 Time complexity of the reduction phase

Again, the size of the input to the reduction step, m, might be different than the size of the original input. However, as explained before, m must polynomially bounded with respect to n, hence $m = O(n^{O(1)})$.

To show that the reduction phase terminates in time $O(n^{O(1)})$, as all the other phases, we first consider the simplification/reduction of a single intersection node.

For that, let b be the maximum branching factor of the input syntax tree. Consequently, each intersection can have no more than b children. We can test any two subqueries for syntactical equality by traversing their syntax trees in the same order. Given that each subquery can consist of at most m nodes, the comparison can be done in $O(m)$ time. Since there are at most

$$\binom{b}{2} = \frac{b(b-1)}{2} \leq b^2 \leq m^2$$

pairs of children to be compared in each intersection node, we can simplify/reduce each intersection node in time:

$$m^2 * O(m) = O(m^3)$$

Finally to carry out the reduction phase, we can perform a postorder traversal of the syntax tree and attempt to simplify and reduce each visited intersection node. Thus, the time complexity of the reduction phase is:

$$O(m) + m * O(m^3) = O(m) + O(m^4) = O(m^4)$$

where $O(m)$ is the syntax tree traversal time and $m * O(m^3)$ is the time taken to reduce, at most, m intersection nodes. This, in turn, implies that the reduction phase terminates in polynomial time with the respect to the size of the original input, as:

$$O(m^4) = O(O(n^{O(1)})^4) = O(n^{4*O(1)}) = O(n^{O(1)})$$

Consequently, the whole ϕ_{IRF} transformation can be run in the polynomial time with respect to the number of nodes in the syntax tree of the input query.

Appendix D

Using *coddifier* Library and *real* Interpreter

Currently, there are two ways to use the functionality of the *coddifier* library:

1. using *coddifier* package directly in the Java program
2. using the command-line interpreter *real*

In this appendix, we will show how to use both methods to verify whether a query $R \cap (S \cap T)$ preserves the Codd semantics in the database with schema:

- R: A (non-nullable), B, C
- S: A, B (non-nullable), C
- T: A, B, C (non-nullable)

To use the *coddifier* library in the source code, it first needs to be added as a dependency. Then the user can access all the classes and interfaces. The code listing below presents a code which creates the schema and the RA query using provided implementations of the respective interfaces:

```
// 1. construct a schema object
var schema = new SimpleSchema.Builder()
    .addTable(
        "R",
        new Attribute("A", false),  // non-nullable
        new Attribute("B", true),   // nullable
        new Attribute("C", true)    // nullable
    )
    .addTable(
        "S",
        new Attribute("A", true),
        new Attribute("B", false),
        new Attribute("C", true)
    )
    .addTable(
        "T",
        new Attribute("A", true),
```

```
        new Attribute("B", true),
        new Attribute("C", false)
    )
    .build();

// 2. build the query
var R = new Relation("R");
var S = new Relation("S");
var T = new Relation("T");
var expression = new Intersection(R, new Intersection(S, T));

// 3. test the query
Coddifier.isGuaranteedToPreserveCoddSemanticsAsIs(
    expression, schema);  // returns false

Coddifier.isGuaranteedToPreserveCoddSemantics(
    expression, schema);  // returns true
```

To test the same query using the *real* interpreter, it must be first built from sources in the repository [8]. Having installed the tool, the following commands can be used to setup the schema and test the query for the preservation of Codd semantics:

```
# 1. create the schema:
#  - the *.csv files can be any existing files - they are not used
#  - the schema can be saved for later use
#  - '!' means that the attribute is non-nullable
&> .add R(A!, B, C) : R.csv
&> .add S(A, B!, C) : S.csv
&> .add T(A, B, C!) : T.csv

# 2. switch the evaluation mode to Codd semantics testing:
&> .eval codd

# 3. enter query using the RA grammar supported by "real":
&> R <I> (S <I> T)
Is guaranteed to preserve Codd semantics: true
```