Teorija relacijskih baza podataka u Coqu

Rajković, Marko

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni preddiplomski studij Matematika i računarstvo

Teorija relacijskih baza podataka u Coqu

Završni rad

Mentor:

doc. dr. sc. Slobodan Jelić

Autor:

Marko Rajković

Osijek, 2022

Sažetak: U ovom ćemo radu promatrati i prezentirati formalizaciju relacijskog modela podataka u Coqu koji je temelj relacijskih baza podataka. Preciznije rečeno, bavit ćemo se formalizacijom onog dijela relacijskog modela podataka koji se odnosi na strukturu podataka, zatim ćemo prezentirati modeliranje dvaju jezika za upite i optimizacije tih jezika i na kraju ćemo promatrati očuvanje integriteta baze podataka gdje ćemo se baviti funkcionalnim ovisnostima.

Ključne riječi: baza podataka, relacijski model, coq, upit, optimizacija, integritet, funkcionalna ovisnost

Relational database theory in Coq

Summary: In this paper we will study and present formalization of the relational data model which is basis of relational databases. We will study formalization of data definition part of the relational data model and we will present two query languages: relational algebra and conjunctive queries together with optimization of those. At the end we will study integrity constraints where we will be particularly focused on functional dependencies.

Keywords: database, relational data model, coq, query, optimization, integrity, functional dependency

Sadržaj

1 | Uvod

1.1 Coq

Coq[6] je formalni sustav upravljanja dokazima koji pruža formalni jezik pomoću kojeg se formaliziraju matematičke definicije, teoremi, algoritmi, specifikacije algoritama, a također omogućuje interaktivno kreiranje dokaza matematičkih teorema ili da algoritmi odgovaraju njihovoj specifikaciji. Coq se koristio u mnogo verifikacijskih projekata od kojih možemo izdvojiti projekt *CompCert*[7].

1.2 Relacijski model baze podataka

Možemo primijetiti kako se danas ogromne količine podataka spremaju i koriste. Imamo brojne sustave i aplikacije koji su zaduženi za upravljanje podacima, a jedni od najkorištenijih sustava jesu sustavi za upravljanje relacijskim bazama podataka. Problem koji se može nazirati u svim sustavima jest pitanje njihove pouzdanosti. Dobivanje čvrstih osnova pouzdanosti zahtijeva korištenje formalnih metoda i alata gdje je Coq vrlo dobar kandidat. Relacijski model koji se koristi u takvim sustavima prikazuje podatke u obliku relacija, a omogućuje i daljnje profiniranje podataka pomoću ograničenja koja čuvaju integritet baze podataka te omogućuje dohvaćanje podataka na temelju jezika upita koji su utemeljeni na algebri (relacijska algebra) ili logici prvog reda (konjuktivni upiti). Postoje dvije ekvivalentne različite verzije relacijskog modela: *nazivni* i *nenazivni*. U *nenazivnom* su okruženju imena atributa zanemareni, a ono što je jedino dostupno jezicima za upite jest ukupan broj atributa. U *nazivnom* su okruženju atributi vidljivi kao eksplicitni dio baze podataka i pri tome su korišteni jezicima za upite. Ima više razloga zašto je *nazivno* okruženje bolje. Imena nam atributa daju bolju semantiku prilikom modeliranja baze podataka, a optimizacije koje se koriste pri kreiranju i izvršavanju upita koriste u pozadini nazive atributa. Poznati sustavi kao što su Oracle, PostgreSQL i DB2 koriste *nazivno* okruženje.

1.3 Struktura projekta

U drugom ćemo poglavlju prezentirati strukturu relacijskog modela zajedno s popratnom implementacijom u Coqu, u trećem ćemo poglavlju obraditi jezike za upite gdje ćemo promatrati relacijsku algebru i konjuktivne upite. U četvrtom ćemo poglavlju prezentirati optimizacije upita gdje ćemo se susresti s važnim

algebarskim jednakostima, teoremom o homomorfizmu i principu minimalnog uvjeta. U petom ćemo poglavlju prezentirati očuvanje integriteta gdje ćemo se posebno dotaknuti funkcionalnim ovisnostima.

2 | Reprezentacija podataka

U relacijskom su modelu podaci reprezentirani pomoću tablica odnosno relacija koje se sastoje od redaka (n-torki). Možemo reći da je jedan stupac relacije pobliže određen jednim atributom. Također svaka relacija ima svoj jedinstven naziv. Na primjer, ukoliko želimo opisivati filmove, svaki film možemo reprezentirati pomoću n-torke gdje atributi mogu biti naziv filma, ime redatelja i godina izlaska. Nadalje, kako su filmovi snimani u određenim mjestima, oni se dalje mogu reprezentirati nazivom mjesta i državom u kojoj se mjesto nalazi. Kada govorimo o domeni atributa, podrazumijevamo da je to skup dopuštenih vrijednosti za određeni atribut.

2.1 Atributi, domene i vrijednosti

U našem ćemo slučaju atribute prezentirati pomoću konačnog i potpuno uređenog skupa. Inače bismo mogli za skup atributa pretpostaviti da je beskonačno prebrojiv, ali takva pretpostavka u našoj formalizaciji nije potrebna. Za prezentaciju konačnih i potpuno uređenih skupova i operacija na njima, koristit ćemo pomoćne funkcionalnosti koje se mogu naći u [5]. Nadalje, skup će vrijednosti biti potpuno uređen. Definirat ćemo svoje tipove koji će predstavljati tipove podataka u relaciji. Svakom atributu možemo pridružiti njegovu domenu ili tip pomoću funkcije `type_of_atribute`. Svakoj vrijednosti atributa također možemo pridodati njezinu domenu ili tip pomoću funkcije `type_of_value`. Možemo definirati funkciju `default_value` za određivanje unaprijed zadane vrijednosti pojedinog atributa. Konkretno, sve možemo objediniti na sljedeći način u Coqu:

```
Module Tuple.
Record Rcd : Type :=
  mk_R {
    type : Type;
    attribute : Type;
    value : Type;
    tuple : Type;
    default_value : type → value;
    type_of_attribute : attribute → type;
    type_of_value : value → type;
  }
End Tuple.
```

Vraćajući se na primjer s filmovima možemo pretpostaviti način kako bismo konkretne vrijednosti implementirali u skladu s navedenim specifikacijama:

```
Inductive attribute :=
  | Title | Director | Publication_Year | Location | Country.
Inductive type :=
  | type_string | type_nat | type_Z.
Inductive value :=
  | Value_String : string → value
  | Value_Nat : nat → value
  | Value_Z : Z → value.
Definition type_of_attribute x :=
  match x with
  | Title | Director | Location | Country ⇒ type_string
  | Publication_Year ⇒ type_nat
  end.
Definition type_of_value x :=
  match v with
  | Value_String s ⇒ type_string
  | Value_Nat n ⇒ type_nat
  | Value_Z z ⇒ type_Z
  end.
```

2.2 Redci tablice

U *nazivnom* su okruženju n-torke karakterizirane njihovim atributima. Na primjer, skup {`Title, Director, Publication_Year`} karakterizira n-torke koje pripadaju relaciji za filmove. Takav ćemo skup nazivati *podrškom* n-torke. Analogno kao kod atributa, n-torke možemo modelirati konačnim i potpuno uređenim skupovima. Koristit ćemo sljedeću oznaku $\overset{\text{set}}{=}$ za jednakost skupova gdje vrijedi $\forall s_1, s_2,\ s_1 \overset{\text{set}}{=} s_2 \Leftrightarrow (\forall e,\ e \in s_1 \Leftrightarrow e \in s_2)$. Možemo proširiti implementaciju na sljedeći način:

```
Module Tuple.
Record Rcd : Type :=
  mk_R {
    ...
    A : Fset.Rcd (Oset.Rcd attribute);
    tuple:  Type;
    OTuple : Oeset.Rcd tuple;
    support : tuple → Fset.set A;
    dot : tuple → attribute → value;
    mk_tuple : Fset.set A → (attribute → value) → tuple;
    dot_mk_tuple_ok : ∀ (a: attribute) (V: Fset.set A)
        (f: attribute → value), dot (mk_tuple V f) a = f a;
    support_mk_tuple_ok : ∀ (V: Fset.set A)
        (f: attribute → value),  support (mk_tuple V f) = V;
```

```
    tuple_eq_ok : ∀ t1 t2,
      (Oeset.compare OTuple t1 t2 = Eq) ↔ (support t1 =set= support t2
      ∧ ∀ a, a ∈ (support t1) → dot t1 a = dot t2 a)
  }
End Tuple.
```

Primijetimo da `A` modelira konačne skupove atributa i operacije nad njima. Primijetimo i razliku u implementaciji potpuno uređenih skupova atributa i n-torki. Razlika je u tome što potpuno uređeni skupovi atributa induciraju *Leibnitzovu jednakost* za razliku od potpuno uređenih skupova n-torki. Implementaciju ćemo n-torke (`tuple`) podrazumijevati apstraktnom, iako se n-torka može definirati kao par gdje je prvi element para skup atributa te n-torke (njezina *podrška*), a drugi element je funkcija koja preslikava dane atribute u vrijednosti zapisana kao `attribute` $\rightarrow$ `value`. Zatim primijetimo funkciju `mk_tuple` koja služi za izgradnju n-torke koja kao argumente prihvaća konačan skup atributa (tj. *podršku* buduće n-torke) i funkciju preslikavanja u vrijednosti. Funkcija `dot` za danu n-torku i atribut daje vrijednost toga atributa te n-torke. Zatim imamo tri specifikacije za n-torke. Specifikacija `dot_mk_tuple_ok` govori da za svaki atribut, konačan skup atributa i funkciju koja određuje vrijednost atributa vrijedi da je vrijednost atributa kreirane n-torke pomoću tog konačnog skupa i te funkcije upravo jednaka vrijednosti te funkcije primijenjene na taj atribut. Specifikacija `support_mk_tuple_ok` govori da za svaki konačni skup atributa i funkciju preslikavanja atributa u vrijednosti vrijedi da je taj konačan skup jednak *podrški* novonastale n-torke. Specifikacija `tuple_eq_ok` govori da su dvije n-torke jednake ako i samo ako imaju istu *podršku* i za svaki atribut podrške prve n-torke vrijedi da je vrijednost tog atributa jednaka vrijednosti atributa druge n-torke. Možemo uvesti definiciju da je n-torka *dobro tipizirana* ako za svaki atribut koji je u *podrški* te n-torke vrijedi da vrijednost tog atributa i taj atribut imaju iste tipove.

```
Definition well_typed_tuple (t : tuple) := ∀ a : attribute,
    a ∈ (support t) → type_of_value (dot t a) = type_of_attribute a.
```

2.3 Relacije, sheme i instance

Razlikujemo shemu baze podataka i instancu baze podataka. Shema baze podataka nam daje specifikaciju strukture baze podataka dok instanca daje konkretne podatke odnosno skup n-torki. Svaka se tablica naziva relacija i ima svoj jedinstven naziv. Skup `relname` sadrži nazive relacija te ćemo modelirati takav skup da bude potpuno uređen. Strukturu tablice opisuje njezin naziv i konačan skup atributa kojeg možemo nazvati *sortom* tablice. Naziv relacije zajedno s njezinom *sortom* nazivamo relacijska shema. Konačan skup takvih relacijskih shema nazivamo shemom baze podataka.

```
Record Rcd {attribute: Type} {OAtt: Oset.Rcd attribute} (A: Fset.Rcd OAtt)
: Type :=
```

```
mk_R {
  relname : Type;
  ORN : Oset.Rcd relname;
  basesort : relname → Fset.set A
}.
```

Funkcija `basesort` vraća za relaciju njezinu *sortu*. Kažemo da instanca ima *dobru sortu* ukoliko vrijedi da za svaku n-torku unutar instance vrijedi da je *podrška* n-torke jednaka *sorti* te relacije:

```
Definition well_sorted_instance (I : relname → Fset.set T ) :=
  ∀ (r : relname) (t : tuple), t ∈ (I r) → support t =set basesort r.
```

gdje `T` modelira konačne skupove n-torki slično kao `A`.

3 | Upiti

Upiti služe da bismo dohvaćali pohranjene informacije iz tablica gdje rezultat može biti također tablica ili kolekcija tablica. Upiti se uobičajeno definiraju jezicima od kojih je SQL najpoznatiji. Jezici se za upite kao što je relacijska algebra ili konjuktivni upiti ne koriste izravno u praksi, nego su važni u teorijskom razmatranju iako mogu biti podloga za druge jezike. U sljedećim ćemo poglavljima vidjeti kako formalizirati ta dva jezika na temelju dosadašnje formalizacije. Pretpostavimo da `T` modelira konačne skupove n-torki i operacija nad njima. Pretpostavimo da `T` koristi reprezentaciju `A` za modeliranje konačnih skupova atributa i operacija nad njima. Obilježimo sa `setA` i `setT` konačne skupove atributa i n-torki.

3.1 Relacijska algebra

Relacijska se algebra sastoji od skupa operatora pri čemu su relacije operandi. U ovom ćemo radu promatrati operatore selekcije, projekcije, prirodnog spoja, preimenovanja, unije, presjeka i razlike. Možemo primijetiti kako operatori presjeka i razlike nisu u skupu minimalne algebre, no dosta su korišteni u komercijalnim jezicima upita pa ćemo ih i mi promatrati. U kontekstu *nazivnih* okruženja prirodni je spoj intuitivan način za spajanje relacija, dok je u *nenazivnim* okruženjima to Kartezijev produkt. Možemo modelirati upite na sljedeći način:

```
Inductive query : Type :=
  | Query_Basename : relname → query
  | Query_Sigma : formula → query → query
  | Query_Pi : setA → query → query
  | Query_NaturalJoin : query → query → query
  | Query_Rename : renaming → query → query
  | Query_Union : query → query → query
  | Query_Inter : query → query → query
  | Query_Diff : query → query → query
  with term : Type :=
  | Term_Constant : value → term
  | Term_Dot : attribute → term
  with atom : Type :=
  | Atom_Eq : term → term → atom
  | Atom_Le : term → term → atom
  with formula : Type :=
  | Formula_Atom : atom → formula
  | Formula_And : formula → formula → formula
```

```
| Formula_Or : formula → formula → formula
| Formula_Not : formula → formula.
```

Za dohvaćanje cijele relacije koristimo upit `Query_basename r` gdje je `r` ime relacije. Kada promatramo selekcijski operator on uobičajeno ima formu $\sigma_{A=B}$ ili $\sigma_{A=a}$ gdje vrijedi $A, B \in$ `attribute` i $a \in$ `value`. No takav način je neprikladan za naše razmatranje i možemo ga zapisati u obliku $x.A = x.B$ $(x.A = a)$ pri čemu je x slobodna varijabla. Za dani skup n-torki I s istom *podrškom* S kažemo da je skup S *sorta* od I. Operator se selekcije primjenjuje na bilo koji skup n-torki I *sorte* S i kao rezultat daje skup iste *sorte* S. Semantiku takvog operatora možemo zapisati u obliku $\sigma_f(I) = \{t |\ t \in I \wedge f\{x \to t\}\}$ gdje se $f\{x \to t\}$ odnosi na to da t ispunjava formulu f pri čemu je x jedina slobodna varijabla. Formule možemo graditi pomoću više konstruktora kao što je i navedeno s time da se svi oni u konačnici osnivaju na uspoređivanju konstantnih vrijednosti ili atributa pojedinih n-torki. Operator se projekcije može zapisati u obliku $\pi_{\{A_1,...,A_n\}}, n \geq 0$ i primjenjuje se na skupovima I na kojima vrijedi $S \supseteq W$ gdje je $W = \{A_1, ..., A_n\}$. Operator projekcije na izlazu daje skup *sorte* W. Semantiku ovog operatora možemo zapisati u obliku $\pi_W(I) = \{t_W |\ t \in I\}$ pri čemu oznaka t_W označava n-torku dobivenu od n-torke t, ali s atributima iz W. Operator prirodnog spoja kojeg označavamo s $\bowtie$ prima ulazne skupove I_1 i I_2 koji imaju *sorte* V i W i kao rezultat daje skup *sorte* $V \cup W$. Semantiku operatora zapisujemo $I_1 \bowtie I_2 = \{t |\ \exists v \in I_1, \exists w \in I_2,\ t|_V = v \wedge t|_W = w\}$. Primijetimo da ukoliko vrijedi $sort(I_1) = sort(I_2)$ onda je $I_1 \bowtie I_2 = I_1 \cap I_2$ te ukoliko je $sort(I_1) \cap sort(I_2) = \emptyset$ onda vrijedi da je $I_1 \bowtie I_2$ Kartezijev produkt tih skupova. Asocijativnost i komutativnost su svojstva operatora prirodnog spoja. Preimenovanje atributa je injektivno preslikavanje s konačnog skupa atributa V u konačan skup atributa. Takva je funkcija g određena skupom parova $(a, g(a))$ pri čemu vrijedi $a \neq g(a)$ što je uobičajeno zapisano kao $a_1a_2...a_n \to b_1b_2...b_n$ da se označi jednakost $g(a_i) = b_i, \forall i \in [1, n], n \geq 0$. Operator imenovanja na ulaznom skupu I nad skupom atributa V je izraz ρ_g gdje je g preslikavanje na skupu V koje preslikava n-torke u izlazni skup nad skupom atributa $g[V]$ što možemo zapisati kao $\rho_g(I) = \{v | \exists u \in I, \forall a \in V, v(g(a)) = u(a)\}$. U našoj smo implementaciji uveli poseban tip `renaming` koji će biti lista parova atributa (a, b) pri čemu će funkcija g biti implementirana na način da će njezina definicija biti `attribute` → `attribute`. Dakle, funkcija g za atribut a_1 pronalazi u listi parova atributa onaj par u kojem je $a = a_1$ te vraća atribut b. Operatori na skupovima koji imaju istu *sortu* kao što je presjek, unija i razlika daju kao rezultat skup koji ima istu *sortu* kao ulazni skupovi i sadrži presjek, uniju ili razliku n-torki.

Prije nego što ćemo interpretirati evaluaciju upita i dati implementaciju u Coqu, definirat ćemo funkciju koja za upite vraća njihovu *sortu* kao što smo objasnili za pojedine operatore.

```
Fixpoint sort (q : query) : setA := match q with
  | Query_Basename r  ⇒ basesort r
  | Query_Sigma f q ⇒ sort q
  | Query_Pi W q ⇒ W ∩ (sort q)
  | Query_NaturalJoin q1 q2 ⇒ (sort q1) ∪ (sort q2)
  | Query_Rename rho q ⇒
```

```
    let sort_q := sort q in
      if one_to_one_renaming_bool (Fset.elements (A T) (sort_q)) rho then
        Fset.map (A T) (A T) (apply_renaming rho) (sort_q)
      else
        ∅
  | Query_Union q1 q2 | Query_Inter q1 q2 | Query_Diff q1 q2 ⇒
  let sort_q1 := sort q1 in
    if sort_q1 =set sort q2
      then sort_q1
    else ∅
  end.
```

Možemo primijetiti za operator projekcije da ćemo umjesto pretpostavke da je skup `W` podskup skupa `sort q` definirati da je *sorta* od `Query_Pi W q` jednaka $W \cap (sort\ q)$. Funkcija `elements` definirana na konačnim skupovima atributa vraća listu elemenata jednog skupa. Funkcija `one_to_one_renaming_bool` provjerava je li preslikavanje sa skupa na skup injektivno na način da gleda ukoliko su elementi kodomene te funkcije jednaki onda moraju biti jednaki i elementi domene. To se uspoređivanje odvija preko `compare` funkcije definirane na uređenim skupovima atributa. Funkcija `map` preslikava svaki element skupa atributa (odnosno u pozadini liste atributa) u novoimenovani atribut i pri tome vraća skup atributa. Primijetimo još parametrizaciju `(A T)` na funkcijama zbog ranijih pretpostavki o zapisima `T` i `A` te nužnosti njihovog korištenja zbog same implementacije u Coqu. Sada možemo definirati pomoćne funkcije za evaluaciju upita i samu evaluaciju upita na sljedeći način:

```
Definition eval_terms_eq (t : tuple) (t1 t2 : term) : bool :=
  match t1 with
  | Term_Dot a ⇒
    match t2 with
    | Term_Constant v ⇒
      match Oset.compare OVal (dot t a) v with
      | Eq ⇒ true
      | _ ⇒ false
      end
    | Term_Dot a2 ⇒
      match Oset.compare OVal (dot t a) (dot t a2) with
        | Eq ⇒ true
        | _ ⇒ false
      end
    end
  | Term_Constant v ⇒
    match t2 with
    | Term_Constant v2 ⇒ false
    | Term_Dot a ⇒
      match Oset.compare OVal (dot t a) v with
      | Eq ⇒ true
      | _ ⇒ false
      end
```

```
    end
  end.

Fixpoint eval_formula f (t : tuple) : bool :=
  match f with
  | Formula_Atom a ⇒
    match a with
      | Atom_Eq t1 t2 ⇒ eval_terms_eq t t1 t2
      | Atom_Le t1 t2 ⇒ eval_terms_le t t1 t2
    end
  | Formula_And f1 f2 ⇒ andb (eval_formula f1 t) (eval_formula f2 t)
  | Formula_Or f1 f2 ⇒ orb (eval_formula f1 t) (eval_formula f2 t)
  | Formula_Not f ⇒ negb (eval_formula f t)
  end.

Fixpoint eval_query (I : relname → setT) (q : query) : setT :=
  match q with
  | Query_Basename r ⇒ I r
  | Query_Sigma f q ⇒ Feset.filter (FTuple T)
    (fun t ⇒ eval_formula f t) (eval_query I q)
  | Query_Pi W q ⇒ Feset.map (FTuple T)
    (FTuple T) (fun t ⇒ mk_tuple W (fun a ⇒
    if a ∈ (support t) then dot t a
    else default_value (type_of_attribute a))) (eval_query I q)
  | Query_NaturalJoin q1 q2 ⇒
    Feset.mk_set (FTuple T) (natural_join_list
    (Feset.elements (FTuple T) (eval_query I q1))
    (Feset.elements (FTuple T) (eval_query I q2)))
  | Query_Rename rho q ⇒ Feset.map (FTuple T)
    (FTuple T) (fun t ⇒ rename_tuple rho t )
    (eval_query I q)
  | Query_Union q1 q2 ⇒ (eval_query I q1) ∪ (eval_query I q2)
  | Query_Inter q1 q2 ⇒ (eval_query I q1) ∩ (eval_query I q2)
  | Query_Diff q1 q2 ⇒ (eval_query I q1) \ (eval_query I q2)
  end.
```

Prije nego što smo definirali evaluaciju upita, reprezentirali smo dvije funkcije: `eval_terms_eq` i `eval_formula`. Funkcija `eval_terms_eq` za danu n-torku provjerava je li vrijednost atributa jednaka zadanoj konstanti ili drugom atributu. Koristimo također i funkciju `eval_terms_le` koja je analogno definirana za provjeru nejednakosti. Funkcija `eval_formula` evaluira danu formulu nad n-torkom kako je rekurzivno i reprezentirana. Obratimo pozornost na funkciju `natural_join_list` u operatoru prirodnog spoja koja kao argumente prima dvije liste n-torki te u suštini za svaku n-torku prve liste provjerava je li kompatibilna s n-torkama druge liste na temelju njihovih zajedničkih atributa te tada stvara listu proširenih n-torki ukoliko jest.

Sve što smo definirali treba zadovoljavati svojstva koja ćemo iskazati u obliku lema (iako ih postoji i više). No idemo uvesti prije toga notacije koje ćemo koristiti u

lemama. Notacija $\overset{I}{=}$ predstavlja da evaluacije upita daju iste skupove n-torki, dok notacija $\in_I$ predstavlja da je neka n-torka unutar skupa dobivenog evaluacijom nekog upita.

```
Lemma mem_basename : ∀ (I : relname → setT) (r : relname)
  (t : tuple), t ∈_I (Query_Basename r) ⇔ t ∈ (I r).

Lemma mem_inter: ∀ (I : relname → setT) (q1 q2 : query),
  sort q1 =^set sort q2 → ∀ (t : tuple), t ∈_I (Query_Inter q1 q2) ⇔
  t ∈_I q1 ∧ t ∈_I q2.

Lemma mem_pi: ∀ (I : relname → setT), well_sorted_instance I →
  ∀ (W : setA) (q : query) (t : tuple), t ∈_I (Query_Pi W q) ⇔
  ∃ (t' : tuple), (t' ∈_I q ∩ t =^t mk_tuple (W ∪ (sort q)) (dot t')).

Lemma mem_natural_join: ∀ (I : relname → setT), well_sorted_instance I →
  ∀ (q1 q2 : query) (t : tuple), t ∈_I (Query_NaturalJoin q1 q2) ⇔
  ∃ t1 : tuple, ∃ t2 : tuple, t1 ∈_I q1 ∧ t2 ∈_I q2 ∧
  (∀ a : attribute, a ∈ (sort q1 ∩ sort q2) → dot t1 a = dot t2 a) ∧
  t =^t mk_tuple ((sort q1) ∪ (sort q2))
  (fun a : attribute => if a ∈ (sort q1) then dot t1 a else dot t2 a).

Lemma mem_rename : ∀ (I : relname → setT), well_sorted_instance I →
  ∀ (rho : list (prod attribute attribute)) (q: query),
  one_to_one_renaming_bool (Fset.elements (A T) (sort q)) rho = true →
  ∀ t, t ∈_I (Query_Rename rho q) ⇔ (∃ (t' : tuple), t' ∈_I q ∧
  t =^t rename_tuple rho t').

Lemma natural_join_inter: ∀ (I : relname → setT),
  well_sorted_instance I → ∀ q1 q2, sort q1 =^S sort q2 →
  Query_NaturalJoin q1 q2 =^I Query_Inter q1 q2.
```

Možemo napomenuti da funkcija `rename_tuple` provodi preimenovanje n-torke na način da kreira novu n-torku s preimenovanim skupom atributa i istim vrijednostima. Kako smo našu n-torku definirali preko para gdje je prvi element skup atributa, a drugi element toga para funkcija koja preslikava atribute u vrijednosti, nova n-torka sadrži kao drugi element funkciju koja za svaki novoimenovani atribut pronalazi stari atribut preko stare *podrške* n-torke i pri tome preuzima njegovu vrijednost.

3.2 Konjuktivni upiti

U ovome se kontekstu upiti osnivaju na predikatnom računu, odnosno n-torke moraju zadovoljavati dani predikat. To možemo zapisati u obliku $\{(a_1, ..., a_n) | \exists b_1, ..., \exists b_m, P_1 \wedge ... \wedge P_k\}$ gdje su a_i, b_i varijable i gdje P_i označava jednakost ili pripadnost relaciji. Na primjer, odgovor na pitanje: "Koji se film snima u Osijeku čiji je redatelj Marko?" može se iskazati u obliku relacijske algebre:

$$\pi_{\{Title,\ Director,\ Publication\_Year\}}(\sigma_{x.Director="Marko"\ \wedge\ x.Location="Osijek"}(Filmovi \bowtie Lokacije)),$$

dok se u obliku konjuktivnih upita može zapisati kao:

$$\{(t,d,p)|\exists t', \exists l, \exists c \wedge Filmovi(t,d,p) \wedge Lokacije(t',l,c) \wedge t = t' \wedge$$
$$d = "Marko" \wedge l = "Osijek"\}$$

Konjuktivne ćemo upite definirati pomoću para (T,s) pri čemu ćemo T nazivati *tabelom*, a s *sažetkom* upita. *Tabela T* sadrži *proširene n*-torke pri čemu su one opisane konstantama ili varijablama. Varijable koje se nalaze u *sažetku s* reprezentiraju oblik kakav ćemo dobiti evaluacijom upita. Možemo to prikazati vizualno:

Title	Director	Publication_Year	Location	Country	
t	"Marko"	p			Filmovi
t			"Osijek"	c	Lokacije
t	d	p			summary

Redak možemo interpretirati na sljedeći način:

```
Inductive trow : Type := Trow : relname → (attribute → tvar) → trow.
```

gdje je

```
Inductive tvar : Type := Tvar : nat → tvar | Tval : value → tvar.
```

Vidimo da retke u ovom kontekstu modeliramo tako da u sebi sadrže relaciju i funkciju koja preslikava atribute u konstante ili varijable. Primjerice, prvi redak prethodnog primjera možemo prikazati u obliku:

```
Trow Filmovi (fun a : attribute ) : trow := match a with
    | Title → Tvar 0
    | Director → -> Tval (Value_String "Marko")
    | Publication_Year → -> Tvar 1
    ...
    end.
```

Tabela T predstavlja skup redaka odnosno elemenata čiji je tip `trow`. Ako modeliramo s `Ftrow` konačne skupove redaka (`trowa`) i operacija na njima, onda možemo definirati:

```
Definition tableau := (Fset.set Ftrow).
Inductive summary : Type := Summary : setA → (attribute → tvar) →
summary.
Definition tableau_query := prod tableau summary
```

Kada bismo željeli dobiti skup *n*-torki pomoću ovakvog načina morali bismo napraviti kompoziciju *sažetka s* funkcijama koje preslikavaju retke *tabele T* u vrijednosti. Stoga bismo prvo trebali definirati preslikavanje varijable u vrijednosti. To ćemo učiniti pomoću funkcija `valuation` i `apply_valuation` na sljedeći način:

```
Definition valuation := nat → value.
Definition apply_valuation (v : valuation) (x : tvar) : value :=
  match x with
    | Tvar n ⇒ v n
    | Tval c ⇒ c
  end.
```

pri čemu smo apstraktno definirali funkciju `valuation`. Nadalje, možemo definirati preslikavanja redaka i *sažetaka*:

```
Notation v[x] := (apply_valuation v x)
Definition apply_valuation_t (v : valuation) (x : trow) : tuple :=
  match x with
    Trow r f ⇒ mk_tuple (basesort r) (fun a ⇒ v[f a])
  end.
Notation v[x]_t := (apply_valuation_t v x)
Definition apply_valuation_s (v : valuation) (x : summary) : tuple :=
  match x with
    Summary V f ⇒ mk_tuple V (fun a ⇒ v[f a])
  end.
Notation v[x]_s := (apply_valuation_s v x)
```

Rezultat za dani upit (T, s) na instanci I definiramo kao $\{t \mid \exists v,\ v(T) \subseteq I \wedge t = v(s)\}$. U našoj ćemo formulaciji taj skup karakterizirati predikatom `is_a_solution I (T,s)`:

```
Inductive is_a_solution (I : relname → setT) :
tableau_query → tuple → Prop :=
  | Extract : ∀ (ST : tableau) (s : summary) (v : valuation),
    (∀ (r : relname) (f : attribute → tvar), (Trow r f) ∈ ST →
    v[Trow r f]_t ∈_I (Query_Basename r)) → ∀ (t : tuple), t =^t v[s]_s →
    is_a_solution I (ST, s) t.
```

4 | Optimizacija

4.1 Optimizacija relacijske algebre

Upiti se optimiziraju zahvaljujući algebarskim jednakostima. Mi ćemo navesti samo klasične primjere i njihove implementacije u Coqu:

$$\sigma_{f_1 \wedge f_2}(q) \equiv \sigma_{f_2 \wedge f_1}(q)$$

```
Lemma sigma_and : ∀ (I : relname → setT), well_sorted_instance I →
  ∀ (f1 f2 : formula) (q : query),
  Query_Sigma (Formula_And f1 f2) q ≡^S Query_Sigma (Formula_And f2 f1) q.
```

$$\sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_2}(\sigma_{f_1}(q))$$

```
Lemma sigma_comm : ∀ (I : relname → setT), well_sorted_instance I →
  ∀ (f1 f2 : formula) (q : query),
  Query_Sigma f1 (Query_Sigma f2 q) ≡^I Query_Sigma f2 (Query_Sigma f1 q).
```

$$(q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3)$$

```
Lemma natural_join_assoc : ∀ (I : relname → setT),
  well_sorted_instance I →
  ∀ (q1 q2 q3 : query), Query_NaturalJoin (Query_NaturalJoin q1 q2) q3 ≡^I
  Query_NaturalJoin q1 (Query_NaturalJoin q2 q3).
```

$$q_1 \bowtie q_2 \equiv q_2 \bowtie q_1$$

```
Lemma natural_join_comm : ∀ (I : relname → setT),
  well_sorted_instance I → ∀ (q1 q2 : query),
  Query_NaturalJoin q1 q2 ≡^I Query_NaturalJoin q2 q1.
```

$$\pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q)$$

ako $W_1 \subseteq W_2$

```
Lemma projection_idempotent : ∀ (I : relname → setT),
    well_sorted_instance I → ∀ (W1 W2 : setA), W1 ⊆ W2 →
    ∀ (q : query), Query_Pi W1 (Query_Pi W2 q) ≡^I Query_Pi W1 q.
```

$$\pi_W(\sigma_f(q)) \equiv \sigma_f(\pi_W(q))$$

ako $Att(f) \subseteq W$

```
Lemma sigma_projection_comm : ∀ (I : relname → setT),
  well_sorted_instance I → ∀ (W : setA) (f : formula),
  (attributes_f f) ⊆ W → ∀ (q : query),
  Query_Pi W (Query_Sigma f q) ≡[I] Query_Sigma f (Query_Pi W q).
```

$$\sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2$$

ako $Att(f) \subseteq sort(q_1)$

```
Lemma sigma_natural_join_comm : ∀ (I : relname -> setT),
  well_sorted_instance I → ∀ (f : formula) (q1 q2 : query),
  (attributes_f f) ⊆ (sort q1) →
  Query_Sigma f (Query_NaturalJoin q1 q2) ≡[I]
  Query_NaturalJoin (Query_Sigma f q1) q2.
```

$$\sigma_f(q_1 \cup q_2) \equiv \sigma_f(q_1) \cup \sigma_f(q_2)$$

```
Lemma sigma_union_distributive : ∀ (I : relname → setT),
  well_sorted_instance I → ∀ (f : formula) (q1 q2 : query),
  Query_Sigma f (Query_Union q1 q2) ≡[I] Query_Union (Query_Sigma f q1)
  (Query_Sigma f q2).
```

$$\sigma_f(q_1 \cap q_2) \equiv \sigma_f(q_1) \cap \sigma_f(q_2)$$

```
Lemma sigma_inter_distributive : ∀ (I : relname → setT),
  well_sorted_instance I → ∀ (f : formula) (q1 q2 : query),
  Query_Sigma f (Query_Inter q1 q2) ≡[I] Query_Inter (Query_Sigma f q1)
  (Query_Sigma f q2).
```

$$\sigma_f(q_1 \backslash q_2) \equiv \sigma_f(q_1) \setminus \sigma_f(q_2)$$

```
Lemma sigma_diff_distributive : ∀ (I : relname → setT),
  well_sorted_instance I → ∀ (f : formula) (q1 q2 : query),
  Query_Sigma f (Query_Diff q1 q2) ≡[I] Query_Diff (Query_Sigma f q1)
  (Query_Sigma f q2).
```

4.2 Optimizacija konjuktivnih upita

Optimizacija je konjuktivnih upita osnovana na sljedećem razmatranju: broj je linija u tabeli T jednak broju spojeva + 1 u relacijskoj algebri. Stoga se optimizacija odnosi na smanjenju broja linija koju možemo postići kroz notaciju tabelarne ekvivalencije i u konačnici minimalnog uvjeta. Pretpostavimo da postoje dva konjuktivna upita (T_1, s_1) i (T_2, s_2).

Kažemo da je (T_1, s_1) sadržan u (T_2, s_2), to jest $(T_1, s_1) \subseteq (T_2, s_2)$ ako i samo ako (T_1, s_1) i (T_2, s_2) imaju isti skup atributa i za svaku su instancu relacije rješenja od (T_1, s_1) sadržana u rješenjima od (T_2, s_2). To nas dovodi do definicije ekvivalentnosti: $(T_1, s_1) \equiv (T_2, s_2)$ ako i samo ako $(T_1, s_1) \subseteq (T_2, s_2) \wedge (T_2, s_2) \subseteq (T_1, s_1)$.

```
Definition is_contained_instance (I : relname → setT)
  (Ts1 Ts2 : tableau_query) :=
  ∀ (t : tuple), is_a_solution I Ts1 t → is_a_solution I Ts2 t.

Definition is_contained (Ts1 Ts2 : tableau_query) :=
  ∀ (I : relname → setT), is_contained_instance I Ts1 Ts2.

Definition are_equivalent (Ts1 Ts2 : tableau_query) :=
  is_contained Ts1 Ts2 ∧ is_contained Ts2 Ts1.
```

Navedeni se semantički pojmovi mogu provjeriti sintaktički oslanjajući se na tabelarne supstitucije. Kažemo da je (tabelarna) supstitucija preslikavanje varijabli u varijable ili konstante. To nas dovodi do teorema koji govori u tabelarnom homomorfizmu.

Teorem 1. *Ako su (T_1, s_1) i (T_2, s_2) konjuktivni upiti, vrijedi da je $(T_1, s_1) \subseteq (T_2, s_2)$ ako i samo ako postoji tabelarna supstitucija θ takva da je za svaku liniju t relacije r u T_2 linija $\theta(t)$ relacije r prikazana u T_1 i vrijedi $\theta(s_2) = s_1$. θ nazivamo tabelarnim homomorfizmom s (T_2, s_2) u (T_1, s_1).*

Navedeno možemo prezentirati u Coqu na sljedeći način uz definiranje načina kako primijeniti supstitucije na varijable(`tvar`), redke(`trow`) i *sažetke*(`summary`):

```
Definition substitution := nat → tvar.

Definition apply_subst_tvar (θ : substitution) (x : tvar) : tvar :=
  match x with
  | Tvar n ⇒ θ n
  | Tval _ ⇒ x
  end.

Definition apply_subst_summary (θ : substitution) (s : summary) :
summary :=
  match s with
  | Summary setA f ⇒ Summary setA (fun a : attribute ⇒
    apply_subst_tvar θ (f a))
  end.

Definition apply_subst_trow (θ : substitution) (t : trow) : trow :=
  match t with
  | Trow relname f ⇒ Trow relname (fun a : attribute ⇒
    apply_subst_tvar θ (f a))
  end.

Parameter summary_equal: summary → summary → bool.

Notation θ[x]_v := (apply_subst_tvar θ x).
Notation θ[x]_r := (apply_subst_trow θ x).
```

```
Notation θ[x]_s := (apply_subst_summary θ x).
Notation s1 ≐^S s2 := (summary_equal s1 s2 = true)

Definition tableau_homomorphism (θ : substitution)
(Ts2 Ts1 : tableau_query) :=
  match Ts1 Ts2 with (T1,s1), (T2,s2) ⇒
    (Fset.map _ Ftrow (fun t ⇒ θ[x]_r) T2) ⊆ T1 ∧ θ[s2]_s ≐^S s1
  end.
Theorem homomorphism_theorem :
 ∀ Ts1 Ts2, (∃ θ, tableau_homomorphism θ Ts2 Ts1) ↔ is_contained Ts1 Ts2.
```

S obzirom na teorem o homomorfizmu za dani konjuktivni upit sada možemo konstruirati ekvivalentni minimalni upit. Također nam to potvrđuje jedan drugi teorem koji govori da za svaki konjuktivni upit postoji ekvivalentni upit među podupitima. Podupit od (T, s) definiramo kao (T', s) pri čemu $T' \subseteq T$. Stoga se optimizacijski proces sastoji u proučavanju svih ekvivalentnih *podtabela* i odabiranju minimalne, a u tom nam je procesu glavna stvar pronaći homomorfizam inicijalne *tabele* u danu.

```
Definition min_tableau (Ts Ms : tableau_query) :=
  are_equivalent Ts Ms ∧ (∀ (Ts': tableau_query), are_equivalent Ts Ts' →
  Fset.cardinal _ (fst Ms) ≤ Fset.cardinal _ (fst Ts')).
Lemma tableau_optimization : ∀ T s, {T' | min_tableau (T, s) (T', s)}.
```

5 | Očuvanje integriteta

Kada govorimo o očuvanju integriteta baze to znači čuvati korektnost i konzistentnost podataka. Korektnost znači da svaki podatak ima ispravnu vrijednost dok konzistentnost znači da su podaci u bazi međusobno usklađeni. U svrhu obrane od narušavanja integriteta uvode se ograničenja odnosno pravila koja moraju biti ispunjena. Jedno od takvih ograničenja jesu ograničenja za čuvanje integriteta unutar relacije kao što su funkcionalne ovisnosti.

5.1 Funkcionalne ovisnosti

Ukoliko imamo situaciju da vrijednosti pojedinih atributa jednoznačno određuju vrijednosti ostalih atributa te n-torke, onda govorimo o funkcionalnoj ovisnosti. Za danu shemu baze podataka R, instancu r od R i skupovima atributa V i W (koji se nalaze u *sorti* od R), kažemo da funkcionalna ovisnost $V \hookrightarrow W$ nad r vrijedi ako $\forall t_1, t_2,\ t_1 \in r \to t_2 \in r \to t_1|_V = t_2|_V \to t_1|_W = t_2|_W$. Neka je F skup funkcionalnih ovisnosti nad danom shemom R. Kažemo da je funkcionalna ovisnost $d = X \hookrightarrow Y$ semantički implicirana skupom F što označavamo $F \models d$ ako $\forall r : R,\ (r \models F \to r \models d)$.

```
Inductive fd : Type := FD : setA → setA → fd.
Notation V ↪ W := (FD V W).
Definition fd_sem (ST : setT) (d : fd) := match d with V ↪ W ⇒
  ∀ (t1,t2 : tuple), t1 ∈ ST → t2 ∈ ST → (∀ x : attribute, x ∈ V →
  dot t1 x = dot t2 x) → ∀ y : attribute, y ∈ W → dot t1 y = dot t2 y
end.
```

Važno pitanje koje se postavlja jest kako odrediti implicirane ovisnosti na danom skupu ovisnosti. Odgovor na to pitanje daje nam Armstrongov sustav koji sadrži tri pravila (odnosno aksioma) pomoću kojih možemo odrediti impliciranu ovisnost, a to su pravila o refleksivnosti, augmentaciji i tranzitivnosti. Primijetimo da nam pravilo `D_Ax` govori o određivanju funkcionalne ovisnosti na temelju danoga skupa ovisnosti. Taj ćemo sustav modelirati *derivacijskim* stablom te ćemo iskazati njegovu robusnost i potpunost.

```
Inductive dtree (F : setF) : fd → Type :=
  | D_Ax : ∀ (X Y : setA), (X ↪ Y) ∈ F → dtree F (X ↪ Y)
  | D_Re : ∀ (X Y : setA), Y ⊆ X -> dtree F (X ↪ Y)
  | D_Aug : ∀ (X Y Z XZ YZ : setA), XZ =S= (X ∪ Z) → YZ =S= (Y ∪ Z) →
    dtree F (X ↪ Y) → dtree F (XZ ↪ YZ)
  | D_Trans : ∀ (X Y Y' Z : setA), Y =S= Y' → dtree F (X ↪ Y) →
    dtree F (Y' ↪ Z) → dtree F (X ↪ Z).
```

```
Theorem Armstrong_soundness : ∀ (F : setF) (d : fd) (t : dtree F d)
(ST : setT),
  (∀ (f : fd), f ∈ F → fd_sem ST f) → fd_sem ST d.

Lemma Armstrong_completeness : ∀ (U X Y : setA) (F : setF),
  X ⊆ U → Y ⊆ U → (∀ (ST : setT), (∀ (t : tuple), t ∈ ST →
  support t ≡(S) U) → (∀ (f : fd), f ∈ F → fd_sem ST f) →
  fd_sem ST (X ↪ Y)) → dtree F (X ↪ Y).
```

Literatura

[1] Benzaken V., Contejean, É., Dumbrava S. (2014). *A Coq Formalization of the Relational Data Model.* In: Shao Z. (eds) Programming Languages and Systems. ESOP 2014. Lecture Notes in Computer Science, vol 8410. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-54833-8_11

[2] R. MANGER, *Baze podataka*, Element d.o.o, Zagreb, 2012

[3] P. PAPIĆ, *Uvod u teoriju skupova*, HMD, Zagreb, 2000

[4] H. Garcia-Molina, J. D. Ullman, J. Widom, *Database systems: the complete book*, Upper Saddle River: Pearson Education, 2009

[5] Web izvor dostupan na https://framagit.org/formaldata/provcert

[6] Web izvor dostupan na https://coq.inria.fr/

[7] Web izvor dostupan na https://compcert.org/

[8] Web izvor dostupan na https://en.wikipedia.org/wiki/Relational_algebra

[9] Web izvor dostupan na https://en.wikipedia.org/wiki/Functional_dependency