Data Insight
Chapter 1: Data Cleaning

- ## Descriere

Scopul proiectului nostru este de a permite utilizatorilor să realizeze şi vizualizeze graficele aferente statisticilor efectuate pe seturi mari de date.

Iniţial intenţionăm să realizăm o multitudine de funcţii care vor extrage date dintr-o bază de date oarecare şi vor crea diferite statistici pe baza acestor date. Ulterior am realizat că acest lucru nu este foarte greu de realizat, noi trebuind doar să implementăm în plsql anumite funcţii deja existente în cărţi/pe internet şi să le facem să se poată adapta pe mai multe tipuri de tabele. Pe lângă asta, în cazul în care datele dintr-un tabel n-ar fi perfect corect inserate, funcţiile noastre ar genera erori.

Astfel, scopul real al proiectului nostru a devenit realizarea primului pas ce trebuie făcut în momentul în care dorim să generăm statistici pentru o anumită bază de date, şi anume ***curăţarea datelor din tabele***.

Curăţarea datelor unei tabele implică numeroase verificări care, în mod normal, sunt extrem de dificil de făcut chiar şi într-o companie foarte mare. Acest proces durează extrem de mult, iar bazele de date nu vor fi, în general, curăţate complet. De asemenea, curăţarea manuală a datelor din baza de date a unei aplicaţii aflată în producţie este aproape imposibil de realizat, din cauza multitudinii de date. În principiu, se poate spune că se poate evita inserare greşită a datelor prin realizarea de verificări automate în momentul inserţiei, dar acoperirea tuturor cazurilor ce implică date greşite este, aproape imposibil de realizat, chiar şi în mod automat.

Aplicaţia noastră va avea următoarele funcţii de curăţare a datelor, funcţii ce se vor aplica pe orice structură de tabele:

- **Rezolvarea duplicatelor:**
  Existenţa duplicatelor este una din cele mai întâlnite probleme în bazele de date. De exemplu pot exista două (sau mai multe) înregistrări ale aceleiaşi persoane într-un tabel cu acelaşi nume, prenume şi adresă de email, dar cu numere de telefon diferite pentru că validarea la înregistrare a fost făcută după numărul de telefon. Pentru aplicaţiile unde nu este permisă utilizarea a mai mult de un cont per user acest fenomen ar fi o adevărată problemă.
  Ca şi soluţii vom aborda: merge între înregistrări, ştergerea duplicatelor.
- **Conversia datelor la tipul corespunzător:**
  Într-o coloană a unui tabel putem avea numai înregistrări de timp number, dar acea coloană să aibă ca tip varchar. Astfel se va face recunoaşterea tipurilor de date înregistrate în tabel şi validarea acestora cu tipul coloanelor, urmând modificarea corespunzătoare e tipului coloanelor în cazul în care este necesar. Totodată dacă avem o coloană de tip varchar(100), dar noi avem numai înregistrări de maxim 10 caractere în acea coloană vom reduce dimensiunea acelui varchar.

- **Rezolvarea erorilor de sintaxă:**
  Erorile de sintaxă sunt de mai multe tipuri. Vom încerca să le rezolvăm pe următoarele:
  - ➔ white spaces: spaţiile albe de la începutul şi sfârşitul înregistrărilor trebuie şterse (trim)
  - ➔ fix typos: în anumite situaţii ne dorim să facem selecţia datelor după anumite standarde, dar datele pot avea mici deviaţii de la standardele propuse; un exemplu tipic este situaţia în care dorim să extragem userii având sexul bărbat dintr-o tabelă folosind tagul “Male”, dar noi avem bărbaţii înregistraţi în tabelă sub următoarele taguri: Man, male, MALE, M., masculin, bărbat.
  - ➔ pad strings: uneori ne dorim ca anumite tipuri de date să aibă o dimensiune fixă, putem completa astfel înregistrările cu spaţii albe la final în cazul stringurilor sau cu 0-uri la început în cazul numerelor până când ajungem la dimensiunea dorită
- **Standardizarea datelor:**
  Într-o tabelă ne dorim să avem datele introduse după anumite standarde:
  -numele şi prenumele încep cu majusculă, iar restul caracterelor sunt minuscule (Andrei, NU aNDREI/andrei/ANDREI)
  -nu avem majuscule intercalate cu minuscule (maşină/MAŞINĂ, nu mAşiNĂ)
  -vrem să avem data de acelaşi format peste tot (zz.ll.aaaa, NU ll/zz/aa sau aaaa-ll-zz)
- **Scalarea/Transformarea datelor:**
  Vrem să oferim posibilitatea de a transforma toate datele sau datele ‘greşite’ în formatul dorit. De exemplu, într-un tabel noi avem notele studenţilor înregistrate, dar aceste note sunt puse în două sisteme de notare diferite: un sistem este cu note de la 1 la 5, iar celălalt de la A la F. Vom notele astfel încât acestea să fie într-un singur sistem de notare.
- **Normalizarea datelor:**
  Normalizarea datelor este un proces ce ajută extrem de mult la generarea statisticilor. Pentru realizarea anumitor statistici este nevoie ca datele să fie distribuite uniform şi la o scală cât mai mică posibil pentru eficientizarea funcţiilor generatoare de statistici.

Totuşi, crearea unei aplicaţii care să cureţe în mod automat şi complet întreagă bază de date necesită cunoştinţe mult prea avansate şi un volum extrem de mare de muncă. Aşadar aplicaţia noastră va trebui puţin configurată înainte de execuţia algoritmilor de curăţare:

-Se va selecta operaţia de curăţare care se doreşte a fi executată

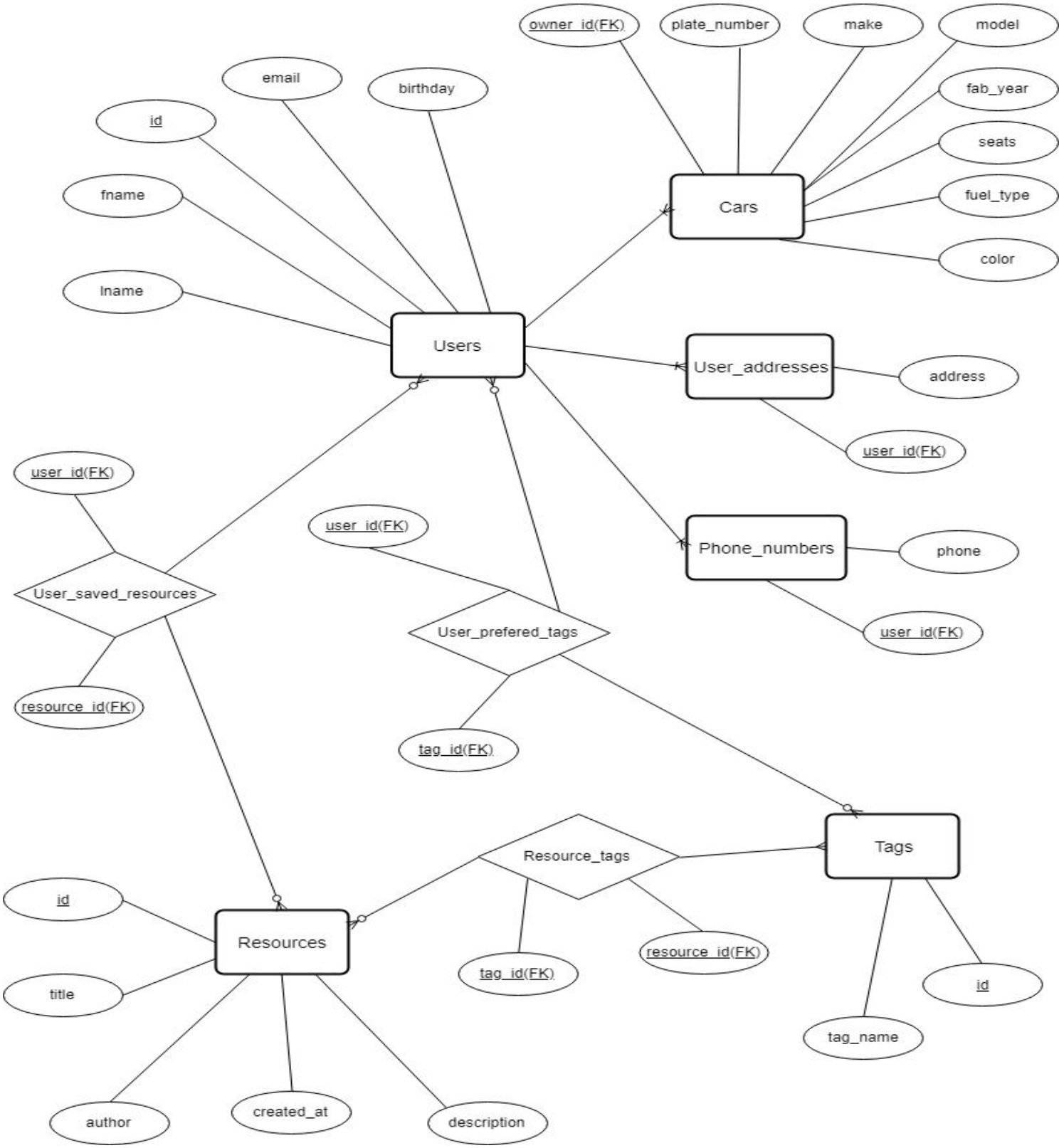
-Se va specifica ce date vor fi modificate şi în ce mod (în cazul standardizării, de exemplu)

-Se va specifica dacă se doreşte doar vizualizarea erorilor existente sau rezolvarea directă a lor cu o soluţie prestabilită/setată de utilizator.

-Va exista şi o serie de funcţii predefinite care vor recunoaşte după anumite key tipurile coloanelor, executând automat curăţarea completă a datelor din acele coloane
Exemplu:
-standardizarea numerelor de telefon în formatul +[prefix ţară]număr, prefixul de ţară fiind automat realizat în funcţie de o coloană în care a fost introdusă ţara;
-standardizarea adresei în formatul City [city], Str. [str], No. [nr]
-semnalarea erorilor în cazul unei coloane cu note, în care apar note mai mari decât 10.

Pentru testarea aplicaţiei am ales să creăm o bază de date cu o tabele ce au diverse tipuri de date şi să generăm înregistrări cu anumite tipuri de erori pentru a le putea curăţa utilizând funcţionalitatea algoritmilor noştri.

- **Schemă Entitate/Relaţie**



- ## Formă normală + Argumentare

Schemele relaţionale şi dependenţele funcţionale

Pornind de la tabelele considerate pentru a modela aplicaţia, vom încerca să ducem baza de date în forma normală 4. Așadar, inițial avem relațiile:

USERS (id, email, fname, lname, birthday, phone, address, car_plate_number, car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color, prefered_tags)

Dependenţe funcţionale USERS:
id → X, X ∈ USER
car_plate_number → Y, Y ∈ {car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color}
RESOURCES (id, author, title, created_at, description, tags)

Dependenţe funţionale RESOURCES:
id → X, X ∈ RESOURCES

1NF

Cunoscând faptul că pe viitor aplicaţia ar putea oferi suport pentru multiple numere de telefon, ceea ce ar încălca prima formă normală, extragem câmpul într-o tabela separată. Obținem astfel:

USERS (id, email, fname, lname, birthday, address, car_plate_number, car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color, prefered_tags)
PHONE_NUMBERS (user_id (fk), phone)

Acelaşi lucru se aplică şi pentru câmpul prefered_tags, acesta fiind plural. Obținem astfel:

USERS (id, email, fname, lname, birthday, address, car_plate_number, car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color)
PHONE_NUMBERS (user_id (fk), phone)
USER_PREFERED_TAGS (user_id (fk), tag)

În mod analog punctului unu, preconizăm faptul că un utilizator ar putea avea multiple adrese, iar în cazul în care aplicaţia ar avea nevoie să cunoască adresa efectivă a unui utilizator, se poate crea o tabelă de asociere, cu cheie străină către adresa considerată efectivă. Considerăm faptul că adresa este atomică, iar aceasta va fi preluată dintr-un singur câmp. După extragerea câmpului adresă din relaţia utilizatori, obținem:

USERS (id, email, fname, lname, birthday, address, car_plate_number, car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color)
id → X, X ∈ USER
car_plate_number → Y, Y ∈ {car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color}
PHONE_NUMBERS (user_id (fk), phone)
id → phone
USER_PREFERED_TAGS (user_id (fk), tag_name)
id → tag
USER_ADDRESSES (user_id (fk), address)
id → address

Analog punctului doi, resursele vor avea multiple taguri, deci trebuie sa extragem din relaţie câmpul tags şi să creem o altă relaţie numită resources tags.

RESOURCES (id, author, title, created_at, description)
Dependenţe funţionale RESOURCES:
id → X, X ∈ RESOURCES
RESOURCE_TAGS (resource_id (fk), tag_name)

Observăm faptul că avem doua relatii similare: USER_PREFERED_TAGS, respectiv RESOURCE_TAGS, având un câmp comun, tag_name. Dacă în viitor, cineva ar dori să vadă ce taguri sunt în sistem, ar trebui să interogheze doua tabele. Aşadar putem crea o noua relatie, numita Tags, la care vor face referire cheile straine din cele doua tabele. Un alt avataj al acestei acţiuni este reprezentat de faptul că dacă un utilizator preferă un tag ce aparţine in mod întâmplător unei resurse, denumirea acestui tag va fi consistentă atat la utilizator, cât şi in resursa. Continuând cu modificările de relatii obţinem:

USERS (id, email, fname, lname, birthday, car_plate_number, car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color)
id → X, X ∈ USER
car_plate_number → Y, Y ∈ {car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color}
PHONE_NUMBERS (user_id (fk), phone)
USER_PREFERED_TAGS (user_id (fk), tag_id (fk))
USER_ADDRESSES (user_id (fk), address)
RESOURCES (id, author, title, created_at, description)
id → X, X ∈ RESOURCES
RESOURCE_TAGS (resource_id (fk), tag_id (fk))
TAGS (id, tag_name)

2NF

Notăm cu k(R) mulţimea cheilor candidat a relaţiei R.

k(USERS) = {(id, car_plate_number)}
k(RESOURCES) = {id}

k(TAGS) = {id}

Observăm că în relaţia USERS exista o cheie candidat, iar din dependenţele funcţionale rezultă faptul că atributele car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color sunt dependente parţial de submulţimea cheii candidat, formată doar din car_plate_number. Aşadar, trebuie sa extragem toate atributele ne-prime în altă tabela şi să referenţiem user-ul. Dupa procedeu, obţinem:

USERS (id, email, fname, lname, birthday)
id → X, X ∈ USER
CARS (owner_id (fk), plate_number, make, model, fab_year, seats, fuel_type, color)
car_plate_number → Y, Y ∈ {car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color}

Recalculând mulţimile de chei candidat a relaţiei R, obţinem:
k(USERS) = {id}
k(CARS) = {plate_number}
k(RESOURCES) = {id}
k(TAGS) = {id}

Deoarece mulţimile de chei a relaţiei R au fiecare doar un singur element, putem să ne oprim din evaluat şi să concluzionăm că schema de relatie R este în 2NF.

3NF

Reevaluam schema de relatie R, incercând să verificăm daca toate atributele ne-prime sunt ne-transitiv dependente de orice cheie candidat a unei relaţii.

USERS (id, email, fname, lname, birthday)
id → X, X ∈ USER
CARS (owner_id (fk), plate_number, make, model, fab_year, seats, fuel_type, color)
car_plate_number → Y, Y ∈ {car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color}
PHONE_NUMBERS (user_id (fk), phone)
USER_PREFERED_TAGS (user_id (fk), tag_id (fk))
USER_ADDRESSES (user_id (fk), address)
RESOURCES (id, author, title, created_at, description)
id → X, X ∈ RESOURCES
RESOURCE_TAGS (resource_id (fk), tag_id (fk))
TAGS (id, tag_name)

Deoarece toate atributele ne-prime din relaţiile de mai sus depind funcţional direct de cheie şi nu şi de un alt atribut ne-prim, rezultă ca schema de relatie R este in 3NF.

BCNF

Calculând mulţimile de chei candidat a relaţiei R, obţinem:

k(USERS) = {id}

k(CARS) = {plate_number}
k(RESOURCES) = {id}
k(TAGS) = {id}

Datorită faptul că pentru orice dependenţă funcţională $X \rightarrow Y$ din schemele de relaţie USERS, CARS, RESOURCES, PHONE_NUMBERS, USER_PREFERED_TAGS, USER_ADDRESSES, RESOURCE_TAGS, TAGS, X este supercheia schemei R sau $X \rightarrow Y$ este o dependenţă funcţională trivială, rezultă că schema de relaţie R este în BCNF.

4NF

Schema de relaţie R şi dependenţele multivaluate:

USERS (<u>id</u>, email, fname, lname, birthday)
$id \twoheadrightarrow X, X \in USER$
CARS (owner_id (fk), <u>plate_number</u>, make, model, fab_year, seats, fuel_type, color)
$car\_plate\_number \twoheadrightarrow Y, Y \in$ {car_make, car_model, car_fab_year, car_seats, car_fuel_type, car_color}
PHONE_NUMBERS (user_id (fk), phone)
USER_PREFERED_TAGS (user_id (fk), tag_id (fk))
USER_ADDRESSES (user_id (fk), address)
RESOURCES (<u>id</u>, author, title, created_at, description)
$id \twoheadrightarrow X, X \in RESOURCES$
RESOURCE_TAGS (resource_id (fk), tag_id (fk))
TAGS (<u>id</u>, tag_name)

Calculând mulţimile de chei candidat a relaţiei R, obţinem:

k(USERS) = {id}
k(CARS) = {plate_number}
k(RESOURCES) = {id}
k(TAGS) = {id}

Datorită faptului că pentru orice dependenţă multivaluată $X \twoheadrightarrow Y$ din schemele de relaţie USERS, CARS, RESOURCES, PHONE_NUMBERS, USER_PREFERED_TAGS, USER_ADDRESSES, RESOURCE_TAGS, TAGS, X este supercheia schemei R sau $X \twoheadrightarrow Y$. Aşadar, putem spune că schema de relaţie este în 4NF.

- **Script creare db + populare**
- **Descriere şi motivare structuri folosite**

Deoarece aplicaţia noastră trebuie să funcţioneze pe diverse tipuri de baze de date şi, implicit, tabele este foarte dificil să folosim indecşi sau views şi nu ar avea nici o utilitate prea mare din moment ce parcurgerea datelor dintr-un tabel se va face doar o dată, sau, cel mult de două ori, în cazul în care utilizatorul doreşte mai întâi să vadă erorile şi apoi să cureţe tot/să aleagă ce să că cureţe. Pe lângă asta nu avem nevoie de eficienţă prea ridicată

din moment ce aplicaţia va servi câte unui singur client (nu este server/app web, ci doar un tool) care îşi va alege funcţiile de curăţare pe care va dori să le ruleze pe baza lui de date.

Pentru testarea aplicaţiei vom folosi tabelele dintr-o aplicaţie uzuală (aplicaţia de la Tehnologii Web care aduce articole de pe mai multe site-uri IT related şi le afişează în funcţie de cerinţele utilizatorului), la care am mai adăugat o tabelă Cars pentru a varia tipurile de date pentru testare. Ulterior vom mai adăuga diferite tabele/baze de date existente pe internet pentru a verifica corectitudinea algoritmilor de curăţare utilizaţi.

Tabele folosite iniţial:

USERS: va reţine datele utilizatorilor.

USER_ADDRESSES: va reţine adresele utilizatorilor (un utilizator poate avea una sau mai multe adrese)

PHONE_NUMBERS: va reţine numerele de telefon ale utilizatorilor (un utlizator poate avea unul sau mai multe numere de telefon)

RESOURCES: aici vor fi stocate temporar informaţiile despre articolele aduse din diverse surse

TAGS: aici vor fi reţinute tag-urile disponibile în aplicaţie

USER_PREFERED_TAGS: aici vor fi stocate tag-urile preferate de utilizatori. Un utilizator poate avea 0 sau mai multe tag-uri preferate, iar un tag poate fi atribuit pentru 0 sau mai mulţi utilizatori.

USER_SAVED_RESOURCES: această tabelă va reţine resursele adăugate la salvate pentru fiecare utilizator. Un utilizator poate avea 0 sau mai multe resurse salvate, iar o resursă poate fi atribuită pentru 0 sau mai mulţi utilizatori.

RESOURCES_TAGS: aici vor fi stocate tag-urile atribuite fiecărei resurse. O resursă poate avea 1 sau mai multe tag-uri atribuite, iar un tag poate fi atribuit pentru 0 sau mai multe resurse.

CARS: aceasta este tabela pe care am adăugat-o pentru a varia tipurile de date pe care să testăm aplicaţia noastră. tabela va fi legată de utilizatori, aceştia având posibilitatea de a avea o maşină şi va reţine toate informaţiile despre maşină.

● Listă funcţii + descriere

➔ *mergeDuplicates*: Funcţia va rezolva duplicatele din tabelele noastre făcând merge între ele după anumite criterii (unul dintre ele ar fi relaţia acestor date cu datele din alte tabele) -> se vor uni duplicatele într-o singură înregistrare, iar datele din alte tabele care aveau ca foreign key unul din câmpurile înregistrărilor care au făcut merge îşi vor schimba foreign key-ul (dacă e necesar).

➔ *deleteDuplicates*: Funcţia va păstra una din înregistrări, în funcţie de anumite criterii pe care le va selecta userul - cele mai puţine câmpuri null, cele mai multe aparenţe ale unui câmp ca foreign key în alte tabele, random - şi va şterge toate celelalte duplicate.

➔ *resolveTypeDimension*: Funcţia va modifica dimensiunile declarate ale tipurilor de date ale coloanelor astfel încât să nu fie alocate dimensiuni prea mari. -> de exemplu pentru cazul în car avem declarat varchar(100), dar noi avem numai înregistrări de maxim 10 caractere, dimensiunea se va modifica la 11, sau la 15-20 dacă ne dorim şi o marjă de eroare

➔ *resolveTypeConficts*: Funcţia va reface tipurile de date declarate coloanelor

astfel încât să se plieze strict pe înregistrările noastre (de exemplu dacă noi avem înregistrate numai numere, dar tipul coloanei este varchar, tipul coloanei va deveni number)

➔ *deleteUnnecessaryWhiteSpaces*: Funcţia va şterge spaţiile în plus de la începutul/sfârşitul înregistrărilor

➔ *fixTypos*: Funcţia va repara stringurile care ar trebui să fie uniformizate. De exemplu în loc de stringul “Male” avem introduse “Masc., M., male”. Funcţia va primi ca parametru coloana şi condiţiile de uniformizare.

➔ *addStringsPadding*: Funcţia va completa înregistrările pentru care ne dorim o dimensiune fixă. Pentru stringuri va adăuga spaţii la final, iar pentru numere va adăuga 0-uri la început.

➔ *dataStandardize*: Funcţia va standardiza datele după anumite standarde primite ca parametru. De exemplu va standardiza adresa după formatul City [city] Str. [str] No. [nr]

➔ *dataTransform*: Funcţia va transforma datele (toate, sau cele care specificate) dintr-un format în altul.

➔ *dataNormalize*: Funcţia va normaliza datele din tabelele/câmpurile specificate, astfel încât să fie distribuite uniform în tabel, sau să fie scalate la limite mai mici.

➔ *autoCleaning*: Funcţia va curăţa automat toată baza de date selectată după anumite criterii setate default din funcţie. Aceasta va face recunoaşterea anumitor tipuri de câmpuri (Name, Phone, Address etc) şi a tipurilor şi va încerca să cureţe cât de mult posibil baza de date.