UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milica Radojičić

AUTOMATIZACIJA PRAVLJENJA RELACIONE BAZE PODATAKA NA OSNOVU TABELARNIH PODATAKA

master rad

Beograd, 2025.

Mentor:

dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Nenad Mitić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Vesna Marinković, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Automatizacija pravljenja relacione baze podataka na osnovu tabelarnih podataka

Rezime: U ovom radu razvijen je softver koji za cilj ima da od kolekcije CSV datoteka napravi predlog projekta (dela) baze podataka u obliku SQL/DDL fajla. Predlog projekta baze podataka obuhvata: tabele, kolone i tipove, sugestije primarnih ključeva, sugestije stranih ključeva i sugestije u vezi sa normalnim formama do treće normalne forme.

Ključne reči: relaciona baza podataka, tabelarni podaci, tabele, kolone, tipovi podataka, primarni ključ, strani ključ, normalne forme, CSV, SQL, DDL

Sadržaj

Glava 1

Uvod

Konvertovanje proizvoljnih kolekcija tabelarnih podataka u tabele relacione baze podataka predstavlja osnovni deo procesa integracije podataka i skladištenja podataka. Pretvaranje tabelarnih podataka u strukturiranu bazu podataka omogućava lakše i efikasnije pravljenje skladišta podataka, ali i drugih vrsta baza podataka.

CSV datoteke su popularan i široko korišćen format za čuvanje i razmenu tabelarnih podataka. Ograničavanjem na upotrebu formata CSV ne sužava se mogućnost primene zato što praktično svi drugi formati mogu da se konvertuju u CSV format. Konvertovanje CSV podataka u relacionu bazu podataka omogućilo bi korisnicima da iskoriste velike mogućnosti i fleksibilnost relacionih baza podataka.

U ovom radu je detaljno predstavljen razvijeni softver koji za cilj ima da od kolekcije CSV datoteka napravi predlog projekta (dela) baze podataka u obliku SQL/DDL fajla. U drugom poglavlju je predstavljen pregled tehnologija korišćenih pri razvoju ovog softvera kao i arhitektura softvera. U trećem poglavlju je fokus na analizi CSV datoteka i automatskoj detekciji zaglavlja. Četvrto poglavlje se bavi analizom kolona i automatskom detekcijom tipova podataka. Peto poglavlje detaljno obrađuje detekciju primarnih ključeva, dok se šesto poglavlje fokusira na strane ključeve. Sedmo poglavlje predstavlja implementaciju algoritma za detekciju narušavanja normalizacije i generisanja sugestija za normalizaciju do treće normalne forme. Osmo poglavlje se bavi generisanjem SQL/DDL fajla nakon analize podataka. Deveto poglavlje je zaključak koji predstavlja kratak rezime urađenog i predloge unapređenja razvijenog softvera.

Implementacija softvera na GitHub-u: `https://github.com/milicar7/master-thesis`

Glava 2

Pregled tehnologija i arhitektura softvera

2.1 Pregled tehnologija

Za implementaciju softvera odabran je programski jezik Python [2], verzija 3.13, koji pruža bogatu kolekciju biblioteka za obradu i analizu podataka, što je centralno za problematiku automatskog otkrivanja strukture podataka.

Python standardna biblioteka sadrži ugrađene module za rad sa CSV datotekama (`csv` modul [1]), regularne izraze (`re` modul [3]), i statističku analizu (`statistics` modul [4]), što eliminiše potrebu za spoljašnjim zavisnostima u osnovnoj funkcionalnosti softvera.

2.2 Arhitektura softvera

Softver je dizajniran prema modularnoj arhitekturi koja jasno razdvaja različite faze obrade podataka. Ova arhitektura omogućava lakše održavanje, testiranje i proširivanje funkcionalnosti.

Slojevi softvera

Arhitektura se sastoji od četiri glavna sloja:

Sloj za obradu CSV datoteka (`csv_processing`) odgovoran je za učitavanje, parsiranje i osnovnu analizu CSV datoteka. Ovaj sloj sadrži komponente za

automatsku detekciju zaglavlja, analizu separatora i osnovnu validaciju strukture datoteka.

Sloj za otkrivanje strukture (`schema_analysis`) predstavlja centralni deo softvera koji implementira algoritme za detekciju tipova podataka, primarnih i stranih ključeva, kao i analizu normalnih formi. Ovaj sloj je podeljen na specijalizovane module za svaki tip analize.

Sloj za generisanje DDL-a (`ddl_generator`) transformiše analiziranu strukturu u konkretan SQL/DDL fajl prilagođen različitim dijalektima baza podataka. Trenutno je dodata podrška za SQLite [6], PostgreSQL [5], MySQL [12] i SQL Server [10].

Sloj za konfiguraciju (`config`) omogućava fino podešavanje algoritama kroz konfiguracione parametre i granične vrednosti.

Modularna organizacija

Svaki sloj je organizovan u logičke module koji enkapsuliraju specifičnu funkcionalnost, kao što je prikazano na slici 2.1.

Projektni obrasci i arhitekturalni principi

Implementacija sistema koristi nekoliko ključnih projektnih obrazaca koji omogućavaju modularnost, proširivost i lakše održavanje koda. Odabrani obrasci nisu samo implementacioni detalji, već omogućavaju sistemu da bude fleksibilan za buduća proširenja i prilagođavanja različitim zahtevima.

Fasada - Glavni interfejs sistema

Klasa `CSVToDDLConverter` implementira Fasada obrazac, pružajući jednostavan interfejs prema složenom sistemu koji koordiniše tri glavne komponente: CSV analizu, analizu strukture i DDL generator.

Listing 2.1: Klasa CSVToDDLConverter

```
class CSVToDDLConverter:
    def __init__(self, dialect: DatabaseDialect =
       DatabaseDialect.POSTGRESQL):
        self.csv_analyzer = CSVAnalyzer()
        self.schema_analyzer = SchemaAnalyzer()
        self.ddl_generator = DDLGenerator(dialect)
```

```
    def convert(self, input_path: Path) -> str:
        tables_headers, tables_data = self.csv_analyzer.
            process(input_path)
        tables_specs = self.schema_analyzer.analyze_tables(
            tables_headers, tables_data)
        return self.ddl_generator.generate_schema_ddl(
            tables_specs)
```

Ovaj pristup omogućava korisnicima da pozivaju celokupnu funkcionalnost kroz jedan jednostavan poziv, dok se složenost koordinacije između komponenti skriva iza čistog interfejsa.

Strategija - Fleksibilnost konfiguracije

`ConfigProvider` i `ConfigManager` definišu interfejs za različite načine učitavanja konfiguracije, omogućavajući lako dodavanje novih izvora konfiguracije (datoteka, baza podataka, ...) bez menjanja postojećeg koda. U postojećoj implementaciji se koristi `DefaultConfigProvider` koji čita vrednosti iz `constants` fajlova.

Listing 2.2: Config klase

```
class ConfigProvider(Protocol):
    def get_csv_config(self) -> CSVConfig:
        ...

    def get_header_config(self) -> HeaderConfig:
        ...

    def get_key_config(self) -> KeyConfig:
        ...

    def get_type_config(self) -> TypeConfig:
        ...

    def get_normalization_config(self) -> NormalizationConfig:
        ...
```

```
class DefaultConfigProvider:
    @staticmethod
    def get_csv_config() -> CSVConfig:
        return CSVConfig()

    @staticmethod
    def get_header_config() -> HeaderConfig:
        return HeaderConfig()

    @staticmethod
    def get_key_config() -> KeyConfig:
        return KeyConfig()

    @staticmethod
    def get_type_config() -> TypeConfig:
        return TypeConfig()

    @staticmethod
    def get_normalization_config() -> NormalizationConfig:
        return NormalizationConfig()

class ConfigManager:
    _provider: Optional[ConfigProvider] = None

    @classmethod
    def initialize(cls, provider: ConfigProvider):
        cls._provider = provider

    @classmethod
    def get_csv_config(cls) -> CSVConfig:
        if cls._provider is None:
            cls._provider = DefaultConfigProvider()
        return cls._provider.get_csv_config()

    @classmethod
    def get_header_config(cls) -> HeaderConfig:
        if cls._provider is None:
```

```
            cls._provider = DefaultConfigProvider()
        return cls._provider.get_header_config()

    @classmethod
    def get_key_config(cls) -> KeyConfig:
        if cls._provider is None:
            cls._provider = DefaultConfigProvider()
        return cls._provider.get_key_config()

    @classmethod
    def get_type_config(cls) -> TypeConfig:
        if cls._provider is None:
            cls._provider = DefaultConfigProvider()
        return cls._provider.get_type_config()

    @classmethod
    def get_normalization_config(cls) -> NormalizationConfig:
        if cls._provider is None:
            cls._provider = DefaultConfigProvider()
        return cls._provider.get_normalization_config()
```

Registar - Dijalekti baza podataka

Softver koristi Registar obrazac za podršku različitih SQL dijalekata, omogućavajući lako dodavanje novih baza podataka kroz konfiguraciju.

Listing 2.3: Konfiguracije za dijalekte

```
DIALECT_CONFIGS = {
    DatabaseDialect.SQLITE: DialectConfig(
        type_mappings={
            DataType.INTEGER: "INTEGER",
            DataType.BIGINT: "INTEGER",
            DataType.DECIMAL: "REAL",
            DataType.FLOAT: "REAL",
            DataType.BOOLEAN: "INTEGER",
            DataType.DATE: "TEXT",
            DataType.TIME: "TEXT",
            DataType.DATETIME: "TEXT",
```

```
            DataType.TIMESTAMP: "TEXT",
            DataType.UUID: "TEXT",
            DataType.EMAIL: "TEXT",
            DataType.URL: "TEXT",
            DataType.JSON: "TEXT",
            DataType.TEXT: "TEXT",
            DataType.VARCHAR: "TEXT",
            DataType.CHAR: "TEXT",
        },
        auto_increment_syntax="AUTOINCREMENT",
        supports_uuid=False,
        supports_json=False,
    ),
    DatabaseDialect.POSTGRESQL: DialectConfig(
        type_mappings={
            DataType.INTEGER: "INTEGER",
            DataType.BIGINT: "BIGINT",
            DataType.DECIMAL: "DECIMAL",
            DataType.FLOAT: "DOUBLE PRECISION",
            DataType.BOOLEAN: "BOOLEAN",
            DataType.DATE: "DATE",
            DataType.TIME: "TIME",
            DataType.DATETIME: "TIMESTAMP",
            DataType.TIMESTAMP: "TIMESTAMP",
            DataType.UUID: "UUID",
            DataType.EMAIL: "VARCHAR",
            DataType.URL: "TEXT",
            DataType.JSON: "JSONB",
            DataType.TEXT: "TEXT",
            DataType.VARCHAR: "VARCHAR",
            DataType.CHAR: "CHAR",
        },
        auto_increment_syntax="SERIAL"
    ),
    DatabaseDialect.MYSQL: DialectConfig(
        type_mappings={
            DataType.INTEGER: "INT",
```

```
            DataType.BIGINT: "BIGINT",
            DataType.DECIMAL: "DECIMAL",
            DataType.FLOAT: "DOUBLE",
            DataType.BOOLEAN: "TINYINT",
            DataType.DATE: "DATE",
            DataType.TIME: "TIME",
            DataType.DATETIME: "DATETIME",
            DataType.TIMESTAMP: "TIMESTAMP",
            DataType.UUID: "CHAR",
            DataType.EMAIL: "VARCHAR",
            DataType.URL: "TEXT",
            DataType.JSON: "JSON",
            DataType.TEXT: "TEXT",
            DataType.VARCHAR: "VARCHAR",
            DataType.CHAR: "CHAR",
        },
        quote_char="`",
        auto_increment_syntax="AUTO_INCREMENT",
        supports_uuid=False,
        boolean_needs_size=True,
    ),
    DatabaseDialect.SQL_SERVER: DialectConfig(
        type_mappings={
            DataType.INTEGER: "INT",
            DataType.BIGINT: "BIGINT",
            DataType.DECIMAL: "DECIMAL",
            DataType.FLOAT: "FLOAT",
            DataType.BOOLEAN: "BIT",
            DataType.DATE: "DATE",
            DataType.TIME: "TIME",
            DataType.DATETIME: "DATETIME2",
            DataType.TIMESTAMP: "DATETIME2",
            DataType.UUID: "UNIQUEIDENTIFIER",
            DataType.EMAIL: "NVARCHAR",
            DataType.URL: "NVARCHAR(MAX)",
            DataType.JSON: "NVARCHAR(MAX)",
            DataType.TEXT: "NVARCHAR(MAX)",
```

```
            DataType.VARCHAR: "NVARCHAR",
            DataType.CHAR: "NCHAR",
        },
        quote_char="[",
        auto_increment_syntax="IDENTITY(1,1)",
        supports_json=False,
    ),
}
```

Šablonski Metod - Analiza normalizacije

Obrazac Šablonski Metod je implementiran u hijerarhiji klasa normalizacije, gde svaki nivo normalizacije nasleđuje zajedničku strukturu ali implementira specifične provere.

Listing 2.4: Klasa NormalForm

```
class NormalForm(ABC):
    @abstractmethod
    def check(self, table_name: str, header: List[str],
              rows: List[List[str]], table_spec: TableSpec) ->
                  List[NormalizationSuggestion]:
        pass

class FirstNormalForm(NormalForm):
    def check(self, table_name: str, header: List[str],
              rows: List[List[str]], table_spec: TableSpec) ->
                  List[NormalizationSuggestion]:
        check_atomicity()

class SecondNormalForm(NormalForm):
    def check(self, table_name: str, header: List[str],
              rows: List[List[str]], table_spec: TableSpec) ->
                  List[NormalizationSuggestion]:
        check_partial_dependency()

class ThirdNormalForm(NormalForm):
    def check(self, table_name: str, header: List[str],
```

```
            rows: List[List[str]], table_spec: TableSpec) ->
                List[NormalizationSuggestion]:
        check_transitive_dependencies()
```

Ovaj pristup omogućava lako dodavanje ostalih normalnih formi bez menjanja postojeće logike.

Lanac Odgovornosti - Detekcija tipova podataka

Detekcija tipova podataka koristi obrazac Lanac Odgovornosti, gde se različiti detektori tipova pozivaju sekvencijalno dok jedan od njih ne postigne dovoljnu pouzdanost.

Listing 2.5: Funkcija detect_column_type

```
def detect_column_type(values: List[str], config: TypeConfig,
   statistics: Optional[ColumnStatistics] = None) -> DataType:
    type_tests = [
        (is_boolean, DataType.BOOLEAN),
        (is_integer, DataType.INTEGER),
        (is_bigint, DataType.BIGINT),
        (is_decimal, DataType.DECIMAL),
        (is_float, DataType.FLOAT),
        (is_uuid, DataType.UUID),
        (is_email, DataType.EMAIL),
        (is_url, DataType.URL),
        (is_datetime, DataType.DATETIME),
        (is_date, DataType.DATE),
        (is_time, DataType.TIME),
        (is_json, DataType.JSON),
    ]


    for test_func, data_type in type_tests:
        match_ratio = test_func(sample_values)
        if match_ratio >= config.
           type_detection_confidence_threshold:
            return data_type
```

Prednosti primenjenih obrazaca

Kombinacija ovih projektnih obrazaca omogućava sistemu da bude:

- **Proširiv** - Novi dijalekti, tipovi podataka i nivoi normalizacije se dodaju kroz konfiguraciju ili nasleđivanje
- **Održiv** - Jasno razdvojene odgovornosti smanjuju složenost i omogućavaju nezavisno menjanje komponenti
- **Fleksibilan** - Obrasci Strategija i Šablonski Metod omogućavaju prilagođavanje različitim zahtevima bez strukturnih izmena

Ovi obrasci nisu odabrani proizvoljno, već predstavljaju odgovor na konkretne izazove domenskog problema konverzije CSV datoteka u SQL strukture, gde je potrebna fleksibilnost za različite formate podataka, baze podataka i analitičke zahteve.

```
csv_to_ddl/
├── main.py (Ulazna tačka aplikacije)
├── argument_parser.py (Parsiranje argumenata komandne linije)
├── csv_to_ddl_converter.py (Glavni orkestrator konverzije)
├── csv_processing/ (Obrada CSV datoteka)
│   ├── csv_analyzer.py (Glavna klasa za CSV analizu)
│   ├── csv_helpers.py (Pomoćne funkcije za pronalaženje i čitanje CSV fajlova)
│   └── header_detection.py (Detekcija zaglavlja CSV fajlova)
├── schema_analysis/ (Analiza strukture)
│   ├── columns_and_types/ (Analiza kolona i tipova)
│   ├── primary_key/ (Detekcija primarnih ključeva)
│   ├── foreign_key/ (Detekcija stranih ključeva)
│   ├── normalization/ (Analiza normalnih formi)
│   └── models/ (Modeli podataka)
├── config/ (Konfiguracija za algoritme detetkcije)
│   ├── config_manager.py (Upravljanje konfiguracijama)
│   ├── config_provider.py (Pružalac konfiguracija)
│   ├── default_config.py (Podrazumevana konfiguracija)
│   └── constants/ (Konstante i pragovi)
│       ├── primary_key.py (Konstante za primarne ključeve)
│       ├── foreign_key.py (Konstante za strane ključeve)
│       ├── type_detection.py (Konstante za detekciju tipova)
│       ├── normalization.py (Konstante za normalizaciju)
│       └── csv.py (CSV konstante)
└── ddl_generator.py (Generisanje SQL/DDL fajla)
```

Slika 2.1: Struktura projekta

Glava 3

Procesiranje CSV fajlova i detekcija zaglavlja

CSV format je jedan od najčešće korišćenih načina za razmenu tabelarnih podataka. Međutim, CSV format nije striktno standardizovan, što dovodi do različitih varijanti u pogledu delimitera, enkodiranja i rukovanja specijalnim karakterima. Ovo poglavlje opisuje implementaciju CSV analizatora koji može da rukuje različitim varijantama CSV formata i automatski detektuje postojanje zaglavlja u CSV fajlovima.

3.1 Procesiranje CSV fajlova

Arhitektura CSV analize

CSV analizator je implementiran kroz klasu `CSVAnalyzer` koja koordiniše čitanje datoteka, detekciju formata i pripremu podataka za dalju analizu. Glavne komponente CSV procesora su:

- **Pronalaženje CSV datoteka** - Rekurzivno skeniranje direktorijuma i identifikacija CSV fajlova
- **Detekcija enkodiranja** - Automatsko prepoznavanje enkodiranja
- **Detekcija delimitera** - Identifikacija separatora kolona
- **Detekcija zaglavlja** - Identifikacija naziva kolona ili generičko generisanje istih

Pronalaženje i čitanje CSV datoteka

Softver podržava fleksibilno učitavanje CSV datoteka iz različitih izvora:

Listing 3.1: Funkcija find_csv_files

```
def find_csv_files(path: Path) -> List[Path]:
    if path.is_file():
        if path.suffix.lower() == '.csv' or path.name.endswith
           ('.csv.gz'):
            return [path]
        return []

    csv_files = []
    try:
        csv_files.extend(path.rglob("*.csv"))
        csv_files.extend(path.rglob("*.CSV"))
        csv_files.extend(path.rglob("*.csv.gz"))
    except PermissionError as e:
        logger.warning(f"Permission denied accessing {path}: {
           e}")

    return csv_files
```

Analizator automatski:

- Prepoznaje pojedinačne datoteke ili direktorijume
- Podržava kompresovane CSV datoteke (.csv.gz)
- Rekurzivno pretražuje poddirektorijume
- Rukuje greškama prilikom pristupa datotekama

Detekcija enkodiranja i delimitera

Jedan od glavnih izazova u procesiranju CSV datoteka je raznovrsnost enkodiranja teksta i delimitera. Analizator implementira detekciju koja pokušava nekoliko čestih enkodiranja:

Listing 3.2: Funkcija read_csv_file

```
def read_csv_file(file_path: Path, sample_size: int,
                  delimiter_detection_sample_size: int) ->
                     List[List[str]]:
    rows = []
    encoding = None
    encodings_to_try = ['utf-8', 'latin-1', 'cp1252', 'iso
       -8859-1']
    sniffer = csv.Sniffer()

    for enc in encodings_to_try:
        try:
            with open_csv_file(file_path, encoding=enc) as f:
                all_lines = [line for line in (f.readline()
                   for _ in range(sample_size)) if line]
                if all_lines:
                    sample_text = ''.join(all_lines[:
                       delimiter_detection_sample_size])
                    delimiter = sniffer.sniff(sample_text).
                       delimiter
                    rows = list(csv.reader(all_lines,
                       delimiter=delimiter))
                    encoding = enc
                    break

        except UnicodeDecodeError:
            continue
        except Exception as e:
            logger.error(f"Error sampling {file_path} with
               encoding {enc}: {e}")
            continue

    if encoding is None:
        logger.warning(f"Could not detect encoding for {
           file_path}, using utf-8")

    return rows
```

Ključne karakteristike:

- **Detekcija delimitera** - Koristi Python biblioteku `csv.Sniffer`
- **Detekcija enkodiranja** - Pokušava UTF-8, zatim Latin-1, CP1252 i ISO-8859-1
- **Optimizovano čitanje** - Čita samo uzorke datoteka, ne cele datoteke

Integracija sa analizom strukture

CSV analizator priprema podatke u standardizovanom formatu koji omogućava dalje procesiranje:

Listing 3.3: Funkcija process

```
def process(self, input_path: Path) -> Tuple[Dict[str, List[
    str]],
                                          Dict[str, List[List[
                                              str]]]]:
    files = find_csv_files(input_path)
    files_data = {}

    for file_path in files:
        rows = read_csv_file(file_path, self.csv_config.
            sample_size,
                             self.csv_config.
                                 delimiter_detection_sample_size)
        if rows:
            csv_data = self.get_header_and_data(rows)
            if csv_data:
                files_data[str(file_path)] = csv_data

    return self.create_individual_tables(files_data)
```

Izlaz CSV procesora čine dva rečnika:

- `tables_headers` - Mapiranje naziva tabela na njihova zaglavlja
- `tables_data` - Mapiranje naziva tabela na redove podataka

Ova struktura omogućava sledećem sloju (analizi strukture) da radi sa standardizovanim formatom podataka nezavisno od originalnog formata CSV datoteka.

3.2 Detekcija zaglavlja

Problem identifikacije zaglavlja

Detekcija zaglavlja je kritičan korak u procesiranju CSV datoteka jer utiče na celokupnu dalju analizu strukture. CSV datoteke mogu:

- Imati zaglavlje u prvom redu
- Počinjati direktno podacima bez zaglavlja

Pogrešna identifikacija zaglavlja može dovesti do neispravne analize tipova podataka i generisanja neadekvatne SQL strukture.

Algoritam za detekciju zaglavlja

Softver implementira algoritam koji kombinuje nekoliko heuristika za detekciju zaglavlja:

Listing 3.4: Funkcija has_headers

```
def has_header(self, rows) -> bool:
    first_row = rows[0]
    data_rows = rows[1:min(len(rows), self.header_config.
       header_data_rows_sample_size)]
    header_score = 0.0

    for i, cell in enumerate(first_row):
        col_score = 0.0
        data_values = [row[i] for row in data_rows[:
           detection_sample_size]]

        col_score += self._calculate_header_indicators_score(
           cell)
        col_score -= self._calculate_anti_header_penalties(
           cell)
        col_score += self._calculate_type_comparison_score(
           cell, data_values)

        header_score += max(0.0, min(1.0, col_score))
```

```
    confidence = header_score / total_cols
    confidence *= self._calculate_uniqueness_penalty(first_row
       )
    confidence += self._calculate_common_pattern_boost(
       first_row)

    return confidence >= self.header_config.
       header_confidence_threshold
```

Komponente algoritma detekcije

Pozitivni indikatori zaglavlja

Algoritam dodeljuje pozitivne poene ćelijama koje imaju karakteristike tipične za nazive kolona:

Listing 3.5: Funkcija calculate_header_indicators_score

```
def calculate_header_indicators_score(self, cell: str) ->
   float:
    col_score = 0.0

    if re.match(r'^[a-zA-Z][a-zA-Z0-9_\s]*$', cell):
        col_score += self.header_config.
           header_letter_pattern_bonus

    if '_' in cell or ' ' in cell:
        col_score += self.header_config.
           header_underscore_space_bonus

    if cell.isupper() or cell.islower():
        col_score += self.header_config.
           header_case_consistency_bonus

    if len(cell) <= self.header_config.
       header_max_length_threshold:
        col_score += self.header_config.header_length_bonus
```

```
    return col_score
```

Negativni indikatori (penali)

Ćelije koje sadrže numeričke ili datum vrednosti su manje verovatno zaglavlja:

Listing 3.6: Funkcija calculate_anti_header_penalties

```
def calculate_anti_header_penalties(self, cell: str) -> float:
    penalty = 0.0

    if (is_integer([cell]) > self.header_config.
       header_type_penalty_threshold or
        is_decimal([cell]) > self.header_config.
           header_type_penalty_threshold):
        penalty += self.header_config.header_numeric_penalty

    if (is_date([cell]) > self.header_config.
       header_type_penalty_threshold or
        is_datetime([cell]) > self.header_config.
           header_type_penalty_threshold):
        penalty += self.header_config.header_date_penalty

    return penalty
```

Poređenje tipova podataka

Kritičan deo algoritma poredi tipove podataka između potencijalnog zaglavlja i podataka:

Listing 3.7: Funkcija calculate_type_comparison_score

```
def calculate_type_comparison_score(self, cell: str,
   data_values: List[str]) -> float:
    header_type = detect_column_type([cell], self.type_config)
    data_type = detect_column_type(data_values, self.
       type_config)

    col_score = 0.0
```

```
    if header_type != data_type:
        col_score += self.header_config.
           header_type_difference_bonus

        # Poseban bonus za tekst vs strukturirani podaci
        if (header_type in [DataType.TEXT, DataType.VARCHAR]
           and
            data_type in [DataType.INTEGER, DataType.FLOAT,
               DataType.DATE]):
            col_score += self.header_config.
               header_text_vs_structured_bonus

    return col_score
```

Globalna prilagođavanja poverenja

Nakon analize individualnih ćelija, algoritam primenjuje globalna prilagođavanja:

- **Kazna za duplikate** - Zaglavlja bi trebalo da budu jedinstvena
- **Bonus za česte obrasce** - Prepoznavanje uobičajenih sufiksa za nazive kolona
- **Prag poverenja** - Finalno poređenje sa konfigurisanim pragom

Listing 3.8: Funkcije calculate_uniqueness_penalty, calculate_common_pattern_boost

```
def calculate_uniqueness_penalty(first_row: List[str]) ->
   float:
    unique_names = len(set(str(cell).strip().lower() for cell
       in first_row))
    return unique_names / max(1, len(first_row))

def calculate_common_pattern_boost(first_row: List[str]) ->
   float:
    common_suffixes = ['_id', '_key', '_date', '_time', '_at',
                       '_count', '_total', '_name', '_code']
```

```
    suffix_matches = sum(1 for cell in first_row
                         if any(str(cell).lower().endswith(
                            suffix)
                                for suffix in common_suffixes))
    return (suffix_matches / len(first_row)) * 0.1
```

Rukovanje slučajevima bez zaglavlja

Kada algoritam zaključi da CSV datoteka nema zaglavlje, softver automatski generiše nazive kolona:

Listing 3.9: Funkcija get_header_and_data

```
def get_header_and_data(self, rows: List[List[str]]) -> Dict[
   str, List[str]]:
    if self.header_detection.has_headers(rows):
        header = [str(h).strip() for h in rows[0]]
        data_rows = rows[1:]
    else:
        header = [f"column_{i + 1}" for i in range(len(rows
           [0]) if rows else 0)]
        data_rows = rows

    return {'header': header, 'rows': data_rows}
```

Glava 4

Kolone i tipovi

Analiza kolona i detekcija tipova podataka predstavlja jedan od centralnih delova procesa konverzije CSV datoteka u bazu podataka. Ovaj sloj softvera odgovoran je za prepoznavanje prirode podataka u svakoj koloni, određivanje odgovarajućeg SQL tipa podataka, specifikaciju veličina i ograničenja, kao i računanje statističkih metrika koje se koriste u kasnijim fazama analize.

4.1 Glavni tok analize kolona

Softver implementira koordinisan pristup analizi kroz ColumnAnalyzer koji povezuje sve komponente u jedinstven proces. Glavni tok analize prolazi kroz četiri ključne faze koje rade sekvencijalno nad podacima svake kolone.

Listing 4.1: Funkcija analyze_column

```
def analyze_column(self, name: str, values: List[str]) ->
   ColumnSpec:
    statistics_obj = self._calculate_statistics(values)
    data_type = detect_column_type(values, self.type_config,
       statistics_obj)
    size_spec = calculate_size_spec(data_type, statistics_obj.
       max_length, self.type_config)
    nullable = statistics_obj.null_count > 0

    return ColumnSpec(
        name=name,
        data_type=data_type,
```

```
        nullable=nullable,
        size_spec=size_spec,
        statistics=statistics_obj
    )
```

Proces analize kolona se zasniva na sledećem redosledu operacija. Prva faza računa osnovne statističke metrike o podacima u koloni - broj nedostajućih vrednosti, broj jedinstvenih vrednosti, udeo jedinstvenih vrednosti, maksimalnu dužinu i prosečnu dužinu stringova. Druga faza koristi algoritme detekcije tipova koji testiraju vrednosti na različitim obrascima i vraćaju SQL tip sa najboljim rezultatom. Treća faza određuje optimalnu specifikaciju veličine na osnovu detektovanog tipa i stvarnih karakteristika podataka. Finalna faza kombinuje sve rezultate u ColumnSpec objekat koji predstavlja kompletnu specifikaciju kolone.

Proces koristi TypeConfig za konfigurisanje različitih parametara detekcije, omogućavajući prilagođavanje različitim tipovima podataka i zahtevima preciznosti. Rezultujući ColumnSpec objekti se prosleđuju algoritmima za detekciju ključeva i analizu normalizacije.

4.2 Detekcija tipova podataka

Algoritam detekcije tipova koristi hijerarhijski pristup testiranja gde se specifični tipovi testiraju pre generičnih. Ovo omogućava precizno prepoznavanje složenih tipova poput UUID, email adresa ili JSON struktura pre nego što se podaci klasifikuju kao obični stringovi.

Listing 4.2: Funkcija detect_column_type

```
def detect_column_type(values: List[str], config: TypeConfig,
   statistics: Optional[ColumnStatistics] = None) -> DataType:
    non_empty_values = [str(v).strip() for v in values
                        if v is not None and str(v).strip()]

    type_tests = [
        (is_boolean, DataType.BOOLEAN),
        (is_integer, DataType.INTEGER),
        (is_bigint, DataType.BIGINT),
        (is_decimal, DataType.DECIMAL),
        (is_float, DataType.FLOAT),
```

```
        (is_uuid, DataType.UUID),
        (is_email, DataType.EMAIL),
        (is_url, DataType.URL),
        (is_datetime, DataType.DATETIME),
        (is_date, DataType.DATE),
        (is_time, DataType.TIME),
        (is_json, DataType.JSON),
    ]

    sample_values = non_empty_values[:min(len(non_empty_values
       ),
                                          config.
                                             type_detection_sample_size
                                             )]

    for test_func, data_type in type_tests:
        match_ratio = test_func(sample_values)
        if match_ratio >= config.
           type_detection_confidence_threshold:
            return data_type

    if statistics and statistics.max_length is not None:
        max_length = statistics.max_length
    else:
        lengths = [len(v) for v in sample_values]
        max_length = max(lengths) if lengths else 0

    if max_length == 1 and all(len(v) == 1 for v in
       sample_values):
        return DataType.CHAR
    elif max_length <= config.max_varchar_length:
        return DataType.VARCHAR
    else:
        return DataType.TEXT
```

Algoritam prvo filtrira prazne vrednosti jer one ne doprinose detekciji tipa. Zatim se iterativno testiraju specijalizovani detektori tipova po redosledu od najspecifičnijih ka najopštijim. Svaki detektor vraća procenat poklapanja između 0.0 i 1.0, a

prvi tip koji dostigne konfigurisan prag poverenja se bira kao finalni tip kolone.

Specijalizovani detektori

Svaki tip podataka ima namenski detektor koji implementira specifičnu logiku prepoznavanja baziranu na obrascima, formatima ili numeričkim opsezima.

BOOLEAN detektor prepoznaje uobičajene reprezentacije logičkih vrednosti kroz skup predefinisanih stringova:

Listing 4.3: Funkcija is_boolean

```
def is_boolean(values: List[str]) -> float:
    boolean_values = {
        'true', 'false', '1', '0', 'yes', 'no', 'y', 'n',
        't', 'f', 'on', 'off', 'enabled', 'disabled'
    }

    matches = sum(1 for v in values if v.lower() in
       boolean_values)
    return matches / len(values) if values else 0
```

INTEGER detektor koristi regularni izraz za prepoznavanje numeričkih vrednosti i prati minimalne i maksimalne vrednosti za određivanje odgovarajućeg opsega:

Listing 4.4: Funkcija is_integer

```
def is_integer(values: List[str]) -> float:
    matches = 0
    min_val, max_val = float('inf'), float('-inf')

    for v in values:
        if re.match(r'^-?\d+$', v):
            matches += 1
            val = int(v)
            min_val = min(min_val, val)
            max_val = max(max_val, val)

    return matches / len(values) if values else 0
```

BIGINT detektor identifikuje cele brojeve koji premašuju standardni 32-bitni opseg:

Listing 4.5: Funkcija is_bigint

```
def is_bigint(values: List[str]) -> float:
    matches = 0
    int_range = 2 ** 31

    for v in values:
        if re.match(r'^-?\\d+$', v):
            try:
                val = int(v)
                if abs(val) > int_range:
                    matches += 1
            except ValueError:
                continue

    return matches / len(values) if values else 0
```

DECIMAL detektor prepoznaje brojeve sa fiksnom tačkom:

Listing 4.6: Funkcija is_decimal

```
def is_decimal(values: List[str]) -> float:
    decimal_pattern = re.compile(r'^-?\\d+\\.\\d+$')
    matches = 0

    for v in values:
        if decimal_pattern.match(v):
            matches += 1

    return matches / len(values) if values else 0
```

FLOAT detektor identifikuje brojeve sa pokretnim zarezom:

Listing 4.7: Funkcija is_float

```
def is_float(values: List[str]) -> float:
    matches = 0

    for v in values:
        try:
            float(v)
            if '.' in v or 'e' in v.lower():
```

```
                matches += 1
            except ValueError:
                continue

    return matches / len(values) if values else 0
```

UUID detektor prepoznaje standardne formate univerzalno jedinstvenih identifikatora:

Listing 4.8: Funkcija is_uuid

```
def is_uuid(values: List[str]) -> float:
    uuid_pattern = re.compile(
        r'^[0-9a-f]{8}-[0-9a-f]{4}-[1-5][0-9a-f]{3}-[89ab][0-9
           a-f]{3}-[0-9a-f]{12}$',
        re.IGNORECASE
    )

    matches = sum(1 for v in values if uuid_pattern.match(v))
    return matches / len(values) if values else 0
```

EMAIL detektor implementira validaciju email adresa kroz regularni izraz koji pokriva osnovne formate:

Listing 4.9: Funkcija is_email

```
def is_email(values: List[str]) -> float:
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA
       -Z]{2,}$'
    matches = sum(1 for v in values if re.match(email_pattern,
        v))
    return matches / len(values) if values else 0
```

URL detektor prepoznaje HTTP i HTTPS adrese:

Listing 4.10: Funkcija is_url

```
def is_url(values: List[str]) -> float:
    url_pattern = r'^https?://[^\\s/$.?#].\\S*$'
    matches = sum(1 for v in values if re.match(url_pattern, v
       , re.IGNORECASE))
    return matches / len(values) if values else 0
```

DATETIME detektor pokušava parsiranje različitih formata datuma i vremena:

Listing 4.11: Funkcija is_datetime

```
def is_datetime(values: List[str]) -> float:
    datetime_formats = [
        '%Y-%m-%d %H:%M:%S', '%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H
            :%M:%S',
        '%Y-%m-%dT%H:%M:%S', '%Y-%m-%d %H:%M:%S.%f'
    ]

    matches = 0
    for v in values:
        for fmt in datetime_formats:
            try:
                datetime.strptime(v, fmt)
                matches += 1
                break
            except ValueError:
                continue

    return matches / len(values) if values else 0
```

DATE detektor pokušava parsiranje različitih formata datuma:

Listing 4.12: Funkcija is_date

```
def is_date(values: List[str]) -> float:
    date_formats = [
        '%Y-%m-%d', '%d/%m/%Y', '%m-%d-%Y', '%d.%m.%Y',
        '%Y/%m/%d', '%m/%d/%Y', '%d-%m-%Y'
    ]

    matches = 0
    for v in values:
        for fmt in date_formats:
            try:
                datetime.strptime(v, fmt)
                matches += 1
                break
            except ValueError:
```

```
                continue

    return matches / len(values) if values else 0
```

TIME detektor prepoznaje različite formate vremena preko regularnih izraza:

Listing 4.13: Funkcija is_time

```
def is_time(values: List[str]) -> float:
    time_patterns = [
        r'^\\d{1,2}:\\d{2}:\\d{2}$',
        r'^\\d{1,2}:\\d{2}$',
        r'^\\d{1,2}:\\d{2}:\\d{2}\\.\\d+$',
    ]

    matches = 0
    for v in values:
        for pattern in time_patterns:
            if re.match(pattern, v):
                matches += 1
                break

    return matches / len(values) if values else 0
```

JSON detektor identifikuje validne JSON strukture kroz parsiranje:

Listing 4.14: Funkcija is_json

```
def is_json(values: List[str]) -> float:
    matches = 0

    for v in values:
        v = v.strip()
        if (v.startswith('{') and v.endswith('}')) or (v.
           startswith('[') and v.endswith(']')):
            try:
                json.loads(v)
                matches += 1
            except json.JSONDecodeError:
                continue
```

```
    return matches / len(values) if values else 0
```

4.3 Detekcija veličine tipova

Nakon određivanja osnovnog tipa podataka, proces iz konfiguracije čita vrednosti za specifikaciju veličine.

Listing 4.15: Funkcija calculate_size_spec

```
def calculate_size_spec(data_type: DataType,
   max_detected_length: int,
                        config: TypeConfig) -> ColumnSizeSpec:
    if data_type in [DataType.VARCHAR, DataType.EMAIL]:
        return ColumnSizeSpec(length=config.max_varchar_length
           )
    elif data_type == DataType.CHAR:
        return ColumnSizeSpec(length=max_detected_length)
    elif data_type == DataType.DECIMAL:
        precision = config.decimal_precision_limit
        scale = config.decimal_scale_limit
        return ColumnSizeSpec(precision=precision, scale=scale
           )
    elif data_type == DataType.UUID:
        return ColumnSizeSpec(length=config.uuid_char_length)
    elif data_type == DataType.BOOLEAN:
        return ColumnSizeSpec(length=config.boolean_length)
    else:
        return ColumnSizeSpec()
```

Prilagođavanje dijalektima

Različite baze podataka imaju različite konvencije za tipove i njihove veličine. Softver implementira sistem prilagođavanja kroz mapiranje tipova specifičnih za svaki dijalekt:

Listing 4.16: Funkcija format_type_with_size

```
def format_type_with_size(data_type: DataType, size_spec:
   ColumnSizeSpec,
```

```
                          dialect: DatabaseDialect) -> str:
    config = DIALECT_CONFIGS[dialect]
    base_type = config.type_mappings.get(data_type, data_type.
       value)

    if data_type in [DataType.VARCHAR, DataType.CHAR, DataType
       .EMAIL] and size_spec.length:
        return f"{base_type}({size_spec.length})"
    elif data_type == DataType.DECIMAL:
        if size_spec.precision and size_spec.scale is not None
           :
            return f"{base_type}({size_spec.precision},{
               size_spec.scale})"
        elif size_spec.precision:
            return f"{base_type}({size_spec.precision})"
    elif data_type == DataType.UUID and not config.
       supports_uuid and size_spec.length:
        return f"{base_type}({size_spec.length})"
    elif data_type == DataType.BOOLEAN and config.
       boolean_needs_size:
        return f"{base_type}({1})"

    return base_type
```

4.4 Statistička analiza kolona

Statistička analiza predstavlja osnovu za sve dalje algoritme koji zavise od karakteristika podataka u kolonama. Ove metrike omogućavaju algoritmima za detekciju ključeva i analizu normalizacije da donose informisane odluke.

Listing 4.17: Funkcija calculate_statistics

```
def calculate_statistics(values: List[str]) ->
   ColumnStatistics:
    null_count = sum(1 for v in values if v is None or str(v).
       strip() == '')
    non_null_values = [str(v).strip() for v in values
                       if v is not None and str(v).strip()]
```

```
    distinct_count = len(set(non_null_values))
    unique_ratio = distinct_count / max(1, len(non_null_values
        ))

    lengths = [len(str(v)) for v in non_null_values]
    max_length = max(lengths) if lengths else None
    avg_length = statistics.mean(lengths) if lengths else None

    return ColumnStatistics(
        null_count=null_count,
        distinct_count=distinct_count,
        unique_ratio=unique_ratio,
        max_length=max_length,
        avg_length=avg_length
    )
```

Algoritam prvo identifikuje sve prazne ili null vrednosti računajući null_count, zatim izdvaja sve validne vrednosti i računa broj jedinstvenih vrednosti kroz konverziju u skup.

unique_ratio se računa kao odnos jedinstvenih vrednosti prema ukupnom broju validnih vrednosti, predstavljajući ključnu metriku za algoritme detekcije primarnih ključeva.

Za string tipove podataka računaju se dodatne metrike dužine: max_length predstavlja najdužu string vrednost u koloni, avg_length predstavlja prosečnu dužinu string vrednosti u koloni.

Izračunate statistike se koriste kroz ostatak softvera na sledeće načine:

null_count direktno određuje nullable ograničenje kolone - ako je null_count veći od nule, kolona se označava kao nullable=true što utiče na mogućnost da bude primarni ključ.

distinct_count i unique_ratio su centralne metrike za algoritme detekcije ključeva.

max_length direktno utiče na specifikaciju veličine za CHAR tip.

avg_length se koristi u algoritmima penalizacije dugačkih tekstova u bodovanju primarnih ključeva, gde kolone sa avg_length preko određenog praga dobijaju negativnu ocenu.

Glava 5

Primarni ključevi

Primarni ključevi predstavljaju osnovu relacionih baza podataka jer jedinstveno identifikuju svaki red u tabeli. U procesu automatskog generisanja DDL-a iz CSV fajlova, detekcija primarnih ključeva je kritična za kreiranje validnih i efikasnih tabela. [9]

Softver implementira tri pristupa za određivanje primarnih ključeva:

- Detekcija prirodnih primarnih ključeva u postojećim kolonama
- Detekcija kompozitnih ključeva kroz kombinacije dve kolone
- Automatsko generisanje surogat ključeva kada prirodni ne postoje

5.1 Glavna logika analize

Analiza primarnih ključeva sledi hijerarhijski pristup kroz PrimaryKeyAnalyzer koji implementira strategiju pokušaja u tri koraka:

1. **Pokušaj detekcije prirodnog PK** - Traže se pojedinačne kolone koje mogu biti primarni ključevi
2. **Pokušaj detekcije kompozitnog PK** - Ako prirodni PK nije pronađen, traže se kombinacije 2 kolone
3. **Generisanje surogat ključa** - Kao poslednja opcija se kreira novi identifikator

Ovaj pristup obezbeđuje da se uvek pronađe rešenje za primarni ključ:

Listing 5.1: Funkcija analyze_primary_key

```
def analyze_primary_key(table_spec, header, rows):
    natural_pk = detect_single_key(table_spec, config)
    if natural_pk:
        logger.info(f"Natural primary key detected: {
            natural_pk.columns}")
        return natural_pk, None

    composite_pk = detect_composite_primary_key(table_spec,
        rows, header, config)
    if composite_pk:
        logger.info(f"Composite primary key detected: {
            composite_pk.columns}")
        return composite_pk, None

    surrogate_pk, surrogate_column =
        generate_surrogate_primary_key(table_spec, config)
    logger.info(f"Surrogate primary key generated: {
        surrogate_pk.columns}")
    return surrogate_pk, surrogate_column
```

5.2 Detekcija prirodnih primarnih ključeva

Prirodni primarni ključevi predstavljaju kolone koje prirodno jedinstveno identifikuju redove na osnovu poslovne logike podataka. Algoritam koristi statističku analizu za identifikaciju kandidata.

Algoritam detekcije

Proces detekcije se odvija kroz tri jasno definisane faze:

1. **Obavezni uslov** - Filtriranje kolona koje ne mogu biti primarni ključevi
2. **Računanje ocene** - Svaka kolona dobija numeričku ocenu na osnovu različitih kriterijuma
3. **Izbor najboljeg kandidata** - Kolone se sortiraju po oceni i bira se najbolja

Listing 5.2: Funkcija detect_single_key

```
def detect_single_key(table_spec, config):
    candidates = []

    for col in table_spec.columns:
        if col.nullable:
            continue

        if not col.statistics or col.statistics.unique_ratio <
            config.pk_uniqueness_threshold:
            continue

        score = calculate_column_pk_score(col, config)
        candidates.append((col.name, score))

    if not candidates:
        return None

    candidates.sort(key=lambda x: x[1], reverse=True)
    col_name, score = candidates[0]

    return PrimaryKeySpec(columns=[col_name], key_type="
       natural")
```

Osnovni uslovi

Kolona mora zadovoljiti sledeće uslove da bi bila razmatrana kao kandidat:

- **Jedinstvenost** - Vrednosti u koloni moraju biti jedinstvene za svaki red (unique_ratio mora biti 1.0)
- **Nedostajuće vrednosti** - Kolona ne sme imati nedostajuće vrednosti (col.nullable = False)

Ovi uslovi su fundamentalni za primarni ključ i ne smeju biti narušeni bez obzira na druge kriterijume.

Algoritam bodovanja

Pošto su osnovni uslovi već zadovoljeni, bodovanje se fokusira na favorizovanje:

- **Semantička analiza naziva** - Prepoznavanje uobičajenih obrazaca imenovanja
- **Prikladnost tipa podataka** - INTEGER, BIGINT i UUID su preferirani, numerički tipovi imaju prednost zbog brzine poređenja i kompaktnosti u indeksima.
- **Penalizacija dugačkih tekstova** - TEXT kolone sa velikim prosečnim dužinama se kažnjavaju

Bodovanje omogućava automatsko prepoznavanje najboljih kandidata na osnovu konvencija i najboljih praksi.

Listing 5.3: Funkcija calculate_column_pk_score

```
def calculate_column_pk_score(col, config):
    score = 0.0

    col_name_lower = col.name.lower()

    if col_name_lower in ['id', 'pk', 'key']:
        score += config.pk_primary_name_bonus
    elif col_name_lower.endswith('_id') or col_name_lower.
       endswith('id'):
        score += config.pk_id_name_bonus

    if col.data_type in [DataType.INTEGER, DataType.BIGINT,
       DataType.UUID]:
        score += config.pk_type_bonus

    if col.data_type == DataType.TEXT and col.statistics and
       col.statistics.avg_length:
        if col.statistics.avg_length > config.
           pk_text_length_threshold:
            score -= config.pk_long_text_penalty
```

```
    return score
```

5.3 Detekcija kompozitnih primarnih ključeva

Kada nijedna kolona ne zadovoljava osnovne uslove za pojedinačni primarni ključ, traže se kombinacije kolona koje zajedno mogu formirati kompozitni ključ.

Algoritam detekcije kompozitnih ključeva

1. **Filtriranje kolona sa nedostajućim vrednostima** - Izdvajaju se kolone kod kojih je col.nullable = False
2. **Validacija veličine kompozitnog ključa** - Provera da li ima dovoljno kolona (pk_composite_size = 2)
3. **Generisanje kombinacija** - Itertools.combinations kreira sve parove kolona
4. **Testiranje jedinstvenosti kombinacije** - test_composite_uniqueness meri unique_ratio nad parovima kolona
5. **Filtriranje po jedinstvenosti** - Zadržavaju se jedinstvene kombinacije
6. **Bodovanje kompozitnih kombinacija** - Suma calculate_column_pk_score za sve kolone u kombinaciji; Koristi se identična funkcija bodovanja kao za pojedinačne ključeve
7. **Selekcija najbolje kombinacije** - Kombinacija sa najboljom ocenom se vraća kao PrimaryKeySpec

Listing 5.4: Funkcija detect_composite_primary_key

```
def detect_composite_primary_key(table_spec, rows, headers,
   config):
    if len(table_spec.columns) < config.pk_composite_size:
        return None

    non_nullable_columns = [col for col in table_spec.columns
       if not col.nullable]
    if len(non_nullable_columns) < config.pk_composite_size:
```

```
        return None

    best_composite = None
    best_score = 0.0

    for col_combination in itertools.combinations(
       non_nullable_columns, config.pk_composite_size):
        col_names = [col.name for col in col_combination]

        uniqueness = test_composite_uniqueness(col_names, rows
           , headers)

        if uniqueness >= config.pk_uniqueness_threshold:
            score = sum(calculate_column_pk_score(col, config)
                for col in col_combination
                       if col.statistics)

            if score > best_score:
                best_score = score
                best_composite = PrimaryKeySpec(columns=
                   col_names, key_type="composite")

    return best_composite
```

Testiranje jedinstvenosti kombinacije

1. **Mapiranje indeksa** - Pronalaženje pozicija kolona u listi zaglavlja
2. **Kreiranje parova vrednosti** - Za svaki red se kreiraju parovi vrednosti
3. **Računanje jedinstvenosti** - Meri se odnos jedinstvenih parova prema ukupnom broju

Funkcija vraća vrednost između 0.0 i 1.0, gde 1.0 označava potpunu jedinstvenost kombinacije

Listing 5.5: Funkcija test_composite_uniqueness

```
def test_composite_uniqueness(col_names, rows, headers):
```

```
    try:
        col_indices = [headers.index(col_name) for col_name in
            col_names]
    except ValueError:
        return 0.0

    combinations = []
    for row in rows:
        if len(row) > max(col_indices):
            combo = tuple(row[i] for i in col_indices)
            combinations.append(combo)

    if not combinations:
        return 0.0

    unique_combinations = len(set(combinations))
    total_combinations = len(combinations)

    return unique_combinations / total_combinations
```

5.4 Generisanje surogat ključeva

Kada prirodni ili kompozitni ključevi ne mogu biti identifikovani, sistem automatski generiše surogat ključ. Surogat ključ se implementira kao:

- **Naziv kolone** - {table_name}_id ili {table_name}_id_n gde je n inkrementalni broj u slučaju konflikta naziva
- **Tip podataka** - INTEGER
- **Ograničenja** - PRIMARY KEY, NOT NULL, AUTO_INCREMENT

Listing 5.6: Funkcija generate_surrogate_primary_key

```
def generate_surrogate_primary_key(table_spec, config):
    surrogate_name = f"{table_spec.name}_id"
    existing_names = {col.name.lower() for col in table_spec.
       columns}
```

```
    counter = 1

    while surrogate_name.lower() in existing_names:
        surrogate_name = f"{table_spec.name}_id_{counter}"
        counter += 1

    surrogate_column = ColumnSpec(
        name=surrogate_name,
        data_type=DataType.INTEGER,
        nullable=False,
        is_auto_increment=True,
        statistics=ColumnStatistics(
            null_count=0,
            distinct_count=table_spec.row_count,
            unique_ratio=1.0
        )
    )

    pk_spec = PrimaryKeySpec(
        columns=[surrogate_name],
        key_type="surrogate"
    )

    return pk_spec, surrogate_column
```

Integracija sa postojećom strukturom

Surogat ključ se dodaje kao poslednja kolona u tabeli:

- Postojeće kolone zadržavaju svoje pozicije
- Dodaje se odgovarajuća DDL sintaksa za auto-increment
- Konfiguracija se prilagođava ciljanoj bazi podataka

Glava 6

Strani ključevi

Strani ključevi uspostavljaju vezu između tabela u relacionoj bazi podataka, obezbeđujući integritet podataka kroz definisanje odnosa roditelj-dete. Automatska detekcija stranih ključeva iz CSV podataka zahteva analizu sadržaja, naziva kolona i statističkih karakteristika. [7]

Softver implementira detekciju kroz dva glavna pristupa:

- Detekcija stranih ključeva kroz pojedinačne kolone
- Detekcija kompozitnih stranih ključeva kroz kombinacije kolona

Metoda za orkestraciju

Ova metoda implementira algoritam za pronalaženje i upravljanje spoljašnjim ključevima u bazi podataka kroz četiri ključne faze:

Kreiranje mape referenci Poziva se `build_reference_keys_map` koja generiše sveobuhvatnu strukturu podataka organizovanu kao mapu sa nazivima tabela kao ključevima, gde svaka tabela sadrži informacije o kolonama koje mogu biti ciljevi spoljašnjih ključeva:

- **header:** Čuvaju se nazivi svih kolona u tabeli kao lista stringova; Potrebno za navigaciju kroz podatke
- **primary_key_columns:** Registruje se skup naziva kolona koje čine primarne ključeve, koristi se za brzu proveru da li kolona pripada primarnom ključu

- **single_keys:** Za kolone koje mogu biti referencirane od strane spoljašnjih ključeva čuva se mapa gde je ključ naziv kolone, a vrednost skup svih različitih vrednosti te kolone. Uključuje primarne ključeve od jedne kolone i kolone sa potpunom jedinstvenošću (100% različitih vrednosti)
- **composite_keys:** Isključivo za složene primarne ključeve (više kolona) čuva se mapa gde je ključ uređena lista naziva kolona, a vrednost skup svih različitih kombinacija vrednosti tih kolona

Ova struktura omogućava algoritmu da:

- Navigira kroz podatke koristeći listu kolona u tabeli (header)
- Brzo identifikuje koja kolona pripada primarnom ključu (primary_key_columns)
- Poredi vrednosti kolona iz jedne tabele sa mogućim ciljevima u drugim tabelama (single_keys za jednostavne i composite_keys za složene reference)

Detekcija spoljašnjih ključeva po tabelama Za svaku tabelu se sprovodi sistematična analiza:

- **Pronalaženje jednostavnih spoljašnjih ključeva:** Algoritam analizira svaku kolonu tabele i poredi njene vrednosti sa vrednostima potencijalnih referentnih ključeva iz drugih tabela
- **Pronalaženje složenih spoljašnjih ključeva:** Analiziraju se kombinacije od dve kolone kao mogući složeni spoljašnji ključevi koji referenciraju složene primarne ključeve
- **Spajanje kandidata:** Svi pronađeni kandidati se objedinjuju u jedinstvenu listu uređenu prema stepenu pouzdanosti

Registracija veza između tabela Implementira se mehanizam za rešavanje konflikata između konkurentnih spoljašnjih ključeva:

- **Kreiranje jedinstvenih parova:** Za svakog kandidata se formira uređeni par koji predstavlja vezu između dve tabele
- **Rešavanje konflikata:** Kada više kandidata pokušava da uspostavi vezu između istog para tabela, pobeđuje onaj sa najvišim stepenom pouzdanosti

- **Sprečavanje duplikata:** Osigurava se da između bilo kog para tabela postoji maksimalno jedna veza

Finalizacija U poslednjoj fazi se pobednički spoljašnji ključevi dodaju u specifikacije odgovarajućih tabela, čime se završava proces detekcije i registracije svih spoljašnjih ključeva u bazi podataka.

Listing 6.1: Funkcija analyze_foreign_keys

```
def analyze_foreign_keys(self, tables_specs: Dict[str,
   TableSpec],
                         tables_data: Dict[str, List[List[str
                            ]]]) -> Dict[str, TableSpec]:
    reference_keys = build_reference_keys_map(tables_specs,
       tables_data, self.config)

    claimed_relationships = {}
    for table_name, table_spec in tables_specs.items():
        table_spec.foreign_keys = []

        table_data = tables_data.get(table_name, [])
        table_header = reference_keys.get(table_name, {}).get(
           'header', [])

        single_fks = detect_single_column_foreign_keys(
                      table_name, table_header,
                      table_data,table_spec,
                      reference_keys, self.config)
        composite_fks = detect_composite_foreign_keys(
                        table_name, table_header,
                        table_data, table_spec,
                        reference_keys, self.config)
        all_fks = single_fks + composite_fks

        for fk in all_fks:
            self.claim_relationship(fk, table_name,
               claimed_relationships)
```

```
    for (source_table, winning_fk) in claimed_relationships.
        values():
         tables_specs[source_table].foreign_keys.append(
            winning_fk)

    return tables_specs
```

6.1 Detekcija stranih ključeva

Algoritam detekcije ima sledeće faze:

1. **Analiza kolona trenutne tabele** - Za svaku kolonu tabele izdvajaju se jedinstvene vrednosti iz stvarnih podataka kao potencijalni spoljašnji ključ
2. **Pretraga referentnih ciljeva** - Poredi se sa svim kolonama iz mape referentnih ključeva (single_keys) u drugim tabelama, preskačući samo sebe
3. **Računanje stepena preklapanja** - Koristi se presek skupova da se izračuna koliko vrednosti kolone postoji u ciljanoj referentnoj koloni
4. **Semantičko ocenjivanje** - Dodeljuju se bonusi na osnovu podudaranja naziva kolona, prepoznatljivih obrazaca za spoljašnje ključeve i pripadnosti primarnom ključu
5. **Selekcija najboljeg kandidata** - Za svaku kolonu bira se referentni cilj sa najvećim stepenom pouzdanosti koji prelazi minimalni prag
6. **Kreiranje specifikacije spoljašnjeg ključa** - Formira se objekat ForeignKeySpec sa nazivima kolona, referentnim ciljem i izračunatim stepenom pouzdanosti

Listing 6.2: Funkcija detect_single_column_foreign_keys

```
def detect_single_column_foreign_keys(table_name, table_header
    , table_data, table_spec, reference_keys, config):
    foreign_keys = []

    for col in table_spec.columns:
```

```
        col_values = get_single_column_values_from_data(col.
            name, table_data, table_header)
        if not col_values:
            continue

        best_match = None
        best_score = 0.0

        for ref_table, ref_data in reference_keys.items():
            if ref_table == table_name:
                continue

            for ref_col, ref_values in ref_data['single_keys'
                ].items():
                match_score, overlap_ratio =
                    calculate_match_score(
                    col_values, ref_values, col.name, ref_col,
                    ref_table, reference_keys, config
                )

                if match_score and match_score > best_score:
                    best_score = match_score
                    best_match = ForeignKeySpec(
                                columns=[col.name],
                                referenced_table=ref_table,
                                referenced_columns=[ref_col],
                                confidence=match_score)

        if best_match:
            foreign_keys.append(best_match)

    return foreign_keys
```

Računanje ocene podudaranja

Funkcija `calculate_match_score` implementira složen algoritam za ocenjivanje kvaliteta potencijalnih veza između kolone kandidata za spoljašnji ključ i referentne

kolone:

- **Validacija stepena preklapanja** - Izračunava se procenat vrednosti iz kolone kandidata koji postoji u referentnoj koloni. Proverava se da li ovaj procenat prelazi minimalni prag za valjan spoljašnji ključ odnos
- **Semantička analiza naziva** - Analiziraju se nazivi kolona i tabela radi identifikovanja očiglednih obrazaca spoljašnjih ključeva (npr. „user_id" u orders tabeli referencira „id" u users tabeli). Dodeljuju se bonusi za direktna poklapanja naziva i prepoznatljive konvencije imenovanja
- **Bonus za primarni ključ** - Ako referentna kolona predstavlja primarni ključ ciljane tabele, dodaje se značajan bonus jer su primarni ključevi najčešći ciljevi spoljašnjih ključeva u relacionim bazama
- **Posebno rukovanje generičkim kolonama** - Kolone sa automatski generisanim nazivima (poput column_1, column_2) dobijaju pojednostavljeno ocenjivanje gde se ignoriše semantička analiza naziva i oslanja se samo na stepen preklapanja vrednosti i bonus za primarni ključ

Finalna ocena predstavlja kombinaciju svih ovih faktora, omogućavajući algoritmu da izdvoji najrelevantniji referentni cilj za svaki potencijalni spoljašnji ključ.

Listing 6.3: Funkcija calculate_match_score

```
def calculate_match_score(col_values, ref_values, col_name,
    ref_col_name,
                          ref_table, reference_keys, config):
    is_valid, overlap_ratio = is_valid_fk_relationship(
        col_values, ref_values, config)
    if not is_valid:
        return 0.0, overlap_ratio

    ref_table_data = reference_keys.get(ref_table, {})
    pk_columns = ref_table_data.get('primary_key_columns', set
        ())
    pk_bonus = get_primary_key_bonus(ref_col_name, pk_columns,
        config)
```

```
    naming_score = get_naming_score(col_name, ref_col_name,
       ref_table, config)
    is_generic_column = (is_generic_column_name(col_name) and
                         is_generic_column_name(ref_col_name))

    if is_generic_column:
        base_score = overlap_ratio * config.fk_validity_bonus
        final_score = base_score + pk_bonus

        if (overlap_ratio >= config.fk_overlap_threshold and
            final_score >= config.fk_validation_threshold):
            return final_score, overlap_ratio
        else:
            return 0.0, overlap_ratio

    if naming_score == 0.0:
        return 0.0, overlap_ratio

    base_score = (overlap_ratio * config.fk_validity_bonus +
                  naming_score * config.fk_naming_bonus)
    final_score = base_score + pk_bonus

    return max(0.0, final_score), overlap_ratio
```

6.2 Detekcija kompozitnih spoljašnjih ključeva

Kompozitni spoljašnji ključevi nastaju kada kombinacija kolona referencira kompozitni primarni ključ druge tabele.

Algoritam detekcije kompozitnih spoljašnjih ključeva

Funkcija `detect_composite_foreign_keys` implementira algoritam za identifikaciju kompozitnih spoljašnjih ključeva kroz sledeće faze:

1. **Ekstrahovanje obrazaca kompozitnih primarnih ključeva** - Prolazi se kroz mapu referentnih ključeva (composite_keys) da bi se identifikovale sve kombinacije kolona koje čine kompozitne primarne ključeve u drugim tabelama

2. **Pronalaženje poklapajućih kolona po nazivu** - Za svaki kompozitni primarni ključ iz referentnih tabela, traže se kolone sa identičnim nazivima u trenutnoj tabeli. Ako se ne pronađe potpuno poklapanje svih kolona, kombinacija se odbacuje

3. **Validacija na osnovu kombinacija vrednosti** - Koriste se skupovi uređenih kombinacija da se izračuna preklapanje između vrednosti kombinacije kolona u trenutnoj tabeli i odgovarajućeg kompozitnog ključa u referentnoj tabeli

4. **Ocenjivanje stepena pouzdanosti** - Izračunava se procenat preklapanja kombinacija vrednosti kao mera pouzdanosti kompozitnog spoljašnjeg ključa. Kombinacije sa visokim stepenom preklapanja se smatraju validnim kandidatima

5. **Kreiranje specifikacije kompozitnog spoljašnjeg ključa** - Za svaki valjan kandidat kreira se ForeignKeySpec objekat koji sadrži listu kolona koje čine spoljašnji ključ, referentnu tabelu, referentne kolone i izračunatu ocenu pouzdanosti

Listing 6.4: Funkcija detect_composite_foreign_keys

```
def detect_composite_foreign_keys(table_name, table_header,
    table_data,
                                  table_spec, reference_keys,
                                      config):
    foreign_keys = []

    composite_pk_patterns = get_composite_pk_patterns(
        reference_keys, table_name)

    for ref_table, ref_columns in composite_pk_patterns:
        matching_columns = find_matching_columns(table_spec,
            ref_columns)

        if matching_columns:
            ref_composite_keys = reference_keys[ref_table]['
                composite_keys']
            best_match = evaluate_composite_match(
                matching_columns, ref_table, ref_columns,
```

```
                ref_composite_keys, table_data, table_header,
                    config
            )

            if best_match:
                foreign_keys.append(best_match)

    return foreign_keys
```

Glava 7

Normalne forme

Normalne forme predstavljaju pravila u dizajnu relacionih baza podataka koja omogućavaju eliminaciju redundantnosti i anomalija u podacima. U kontekstu automatskog generisanja DDL-a iz CSV fajlova, analiza normalnih formi pomaže u identifikaciji potrebe za dekompozicijom tabela i optimizaciji strukture podataka. [8] [11]

Softver implementira detekciju i predloge za prve tri normalne forme:

- Prva normalna forma (1NF) - Atomske vrednosti u svim kolonama
- Druga normalna forma (2NF) - Eliminacija parcijalne funkcionalne zavisnosti
- Treća normalna forma (3NF) - Eliminacija tranzitivne funkcionalne zavisnosti

Listing 7.1: Funkcija analyze_normalization

```
def analyze_normalization(self, table_name: str, table_header:
    List[str],
                          table_data: List[List[str]],
                             table_spec: TableSpec) -> List[
                             NormalizationSuggestion]:
    suggestions = []

    first_nf_violations = self.first_nf.check(table_name,
       table_header, table_data, table_spec)
    suggestions.extend(first_nf_violations)
    if first_nf_violations:
        self.logger.info(
```

```
            f"Skipping 2NF and 3NF checks for
               columns_and_types '{table_name}' due to 1NF
               violations")
        return suggestions

    second_nf_violations = self.second_nf.check(table_name,
       table_header, table_data, table_spec)
    suggestions.extend(second_nf_violations)
    if second_nf_violations:
        self.logger.info(f"Skipping 3NF check for
           columns_and_types '{table_name}' due to 2NF
           violations")
        return suggestions

    third_nf_violations = self.third_nf.check(table_name,
       table_header, table_data, table_spec)
    suggestions.extend(third_nf_violations)
    return suggestions
```

7.1 Prva normalna forma

Prva normalna forma zahteva da sve vrednosti u tabeli budu atomske (nedeljive) i da ne postoje ponavljajuće grupe podataka. CSV fajlovi često krše ovu formu kroz složene vrednosti u jednoj ćeliji.

Detekcija kršenja Prve normalne forme (1NF)

Funkcija `check` u klasi `FirstNormalForm` implementira algoritam za identifikaciju višestrukih vrednosti koje krše 1NF zahtev za atomske vrednosti:

Karakteristike detekcionog algoritma

- **Predefinisani separatori višestrukih vrednosti** - Algoritam prepoznaje standardne separatore koji ukazuju na postojanje više vrednosti u jednoj ćeliji: zarez (,), tačka-zarez (;), uspravna crta (|), novi red (\n) i tab karakter (\t)

- **Statistički pristup kvantifikacije** - Za svaku kolonu računa se procenat ćelija koje sadrže jedan ili više separatora u odnosu na ukupan broj nepraznih ćelija u toj koloni
- **Filtriranje po pragu značajnosti** - Generiše se predlog za normalizaciju samo za kolone gde je stepen kršenja 1NF veći od 10% da se izbegnu lažni pozitivni rezultati

Tok izvršavanja algoritma

Algoritam se izvršava kroz sledeće korake:

1. **Iteracija kroz kolone** - Svaka kolona tabele se analizira nezavisno
2. **Skeniranje ćelija za separatore** - Za svaku nepraznu ćeliju u koloni proverava se prisustvo bilo kog od predefinisanih separatora. Svaka ćelija koja sadrži bar jedan separator se smatra da krši 1NF
3. **Računanje stepena kršenja** - Izračunava se odnos broja ćelija sa separatorima prema ukupnom broju validnih ćelija u koloni
4. **Kreiranje predloga za normalizaciju** - Za kolone sa stepenom kršenja od 10% ili više generiše se `NormalizationSuggestion` objekat koji sadrži opis problema i preporučene rešenje (razdvajanje u zasebne kolone ili kreiranje povezane tabele)

Listing 7.2: check (1NF)

```
def check(table_name, header, rows, table_spec):
    suggestions = []

    if not rows or not header:
        return suggestions

    for col_idx, col_name in enumerate(header):
        multi_value_indicators = [',', ';', '|', '\n', '\t']
        multi_value_count = 0
        total_count = 0

        for row in rows:
```

```
            if col_idx < len(row) and row[col_idx]:
                cell_value = str(row[col_idx]).strip()
                total_count += 1

                for indicator in multi_value_indicators:
                    if indicator in cell_value:
                        multi_value_count += 1
                        break

        if total_count > 0:
            multi_value_ratio = multi_value_count /
               total_count

            if multi_value_ratio >= 0.1:
                description = (
                    f"Column '{col_name}' contains multi-
                       valued data in "
                    f"{multi_value_ratio:.1%} of rows.
                       Consider splitting into "
                    f"separate columns or creating '{col_name}
                       _values' table."
                )

                suggestion = NormalizationSuggestion(
                    table_name=table_name,
                    suggestion_type="1NF",
                    description=description,
                    confidence=multi_value_ratio
                )
                suggestions.append(suggestion)

    return suggestions
```

7.2 Druga normalna forma

Druga normalna forma eliminiše parcijalnu funkcionalnu zavisnost. Tabela je u 2NF ako je u 1NF i svaki ne-ključni atribut funkcionalno zavisi od celog primarnog ključa, a ne samo od njegovog dela.

Detekcija kršenja Druge normalne forme (2NF)

Funkcija `check` u klasi `SecondNormalForm` implementira algoritam za identifikaciju parcijalnih zavisnosti koje krše 2NF zahtev da svi ne-ključni atributi moraju u potpunosti zavisiti od kompozitnog primarnog ključa:

1. **Validacija preduslova** - Proverava da li tabela ima kompozitni primarni ključ i dovoljno podataka za analizu
2. **Priprema za analizu** - Identifikuje komponente primarnog ključa i ne-ključne kolone u tabeli
3. **Detekcija parcijalnih zavisnosti** - Poziva algoritam koji testira zavisnost ne-ključnih kolona od komponenti primarnog ključa
4. **Generisanje predloga normalizacije** - Za svaku pronađenu parcijalnu zavisnost kreira predlog izdvajanja u zasebnu tabelu sa odgovarajućim nazivom
5. **Vraćanje rezultata** - Vraća listu predloga za normalizaciju tabele radi rešavanja narušavanja 2NF

Listing 7.3: check (2NF)

```
def check(self, table_name: str, header: List[str],
          rows: List[List[str]], table_spec: TableSpec) ->
             List[NormalizationSuggestion]:
    suggestions = []

    if not table_spec or not table_spec.primary_key or len(
       table_spec.primary_key.columns) <= 1:
        return suggestions

    if not rows or not header:
        return suggestions
```

```
    pk_columns = table_spec.primary_key.columns
    pk_indices = []
    for pk_col in pk_columns:
        try:
            pk_indices.append(header.index(pk_col))
        except ValueError:
            continue

    if len(pk_indices) < 2:
        return suggestions

    non_key_columns = [col for col in table_spec.columns if
       col.name not in pk_columns]
    partial_dependencies = self._find_partial_dependencies(
       non_key_columns, rows, header, pk_columns, pk_indices)

    for dependency in partial_dependencies:
        partial_key_clean = dependency['partial_key'].lower().
           replace('_id', '').replace('id', '')
        dependent_col_clean = dependency['column'].lower().
           replace('_name', '').replace('name', '')

        if dependent_col_clean and partial_key_clean !=
           dependent_col_clean:
            proposed_table_name = f"{partial_key_clean}_{
               dependent_col_clean}"
        else:
            proposed_table_name = f"{partial_key_clean}s" if
               not partial_key_clean.endswith(
                's') else partial_key_clean

        suggestion = NormalizationSuggestion(
            table_name=table_name,
            suggestion_type="2NF",
            description=f"Column '{dependency['column']}' has
               partial dependency on "
```

```
                        f"'{dependency['partial_key']}' (part
                            of composite key {pk_columns}) "
                        f"instead of depending on the full key
                             (strength: {dependency['strength
                            ']:.1%}). "
                        f"Consider extracting to a separate '{
                            proposed_table_name}' table with "
                        f"columns: {dependency['partial_key
                            ']}, {dependency['column']}.",
                    confidence=dependency['strength']
                )
                suggestions.append(suggestion)

        return suggestions
```

Algoritam detekcije parcijalnih zavisnosti

Algoritam se izvršava kroz sledeće korake:

1. **Preduslovna validacija** - Proverava se da li tabela ima dovoljno redova za analizu prema konfiguracionom pragu (nf2_min_rows_for_analysis)
2. **Analiza svake ne-ključne kolone** - Za svaku kolonu koja nije deo primarnog ključa:
   - Pronalazi se indeks kolone u zaglavlju tabele pozivom `headers.index(col.name)`
   - Izdvajaju se jedinstvene vrednosti iz podataka iteriranjem kroz sve redove
   - Proverava se da li kolona ima dovoljno različitih vrednosti prema konfiguracionom pragu (nf2_min_column_values)
3. **Testiranje protiv svake komponente primarnog ključa** - Za svaku komponentu kompozitnog primarnog ključa:
   - Pozivaju se funkcije _build_key_groups, _calculate_determination_strength i _evaluate_dependency_strength
   - Grupišu se redovi prema vrednostima parcijalne komponente ključa i pune komponente ključa

- Izračunava se stepen determinacije kao odnos grupa sa jedinstvenim vrednostima prema ukupnom broju grupa
- Poredi se jačina parcijalne zavisnosti sa jačinom pune zavisnosti kroz konfigurisane pragove

4. **Validacija pragova zavisnosti** - Parcijalna zavisnost se smatra validnom ako:
   - Parcijalna jačina samostalno prelazi minimalni prag (nf2_min_partial_strength)
   - Jačina pune zavisnosti prelazi prag (nf2_full_key_strength_threshold)
   - Odnos parcijalne i pune jačine prelazi prag (nf2_partial_dependency_threshold)
5. **Kreiranje liste zavisnosti** - Za validne parcijalne zavisnosti dodaje se u listu rečnik koji sadrži:
   - Naziv kolone koja ima parcijalnu zavisnost
   - Naziv komponente primarnog ključa od koje zavisi
   - Numeričku vrednost jačine zavisnosti kao stepen pouzdanosti

Listing 7.4: Funkcija find_partial_dependencies

```
def find_partial_dependencies(self, non_key_columns: List,
   rows: List[List[str]],
                              headers: List[str], pk_columns:
                                 List[str],
                              pk_indices: List[int]) -> List[
                                 Dict]:
    dependencies = []

    if len(rows) < self.config.nf2_min_rows_for_analysis:
        return dependencies

    for col in non_key_columns:
        try:
            col_index = headers.index(col.name)
        except ValueError:
            continue
```

```
        col_values = set()
        for row in rows:
            if len(row) > col_index and row[col_index] and str
               (row[col_index]).strip():
                col_values.add(row[col_index])

        if len(col_values) < self.config.nf2_min_column_values
           :
            continue

        for i, pk_part_index in enumerate(pk_indices):
            partial_key_groups, full_key_groups = self.
               _build_key_groups(
                rows, col_index, [pk_part_index], pk_indices
            )

            if not partial_key_groups or not full_key_groups:
                continue

            partial_strength = self.
               _calculate_determination_strength(
               partial_key_groups)
            full_strength = self.
               _calculate_determination_strength(
               full_key_groups)

            partial_dependency_strength = self.
               _evaluate_dependency_strength(partial_strength,
                full_strength)

            if partial_dependency_strength >= self.config.
               nf2_partial_dependency_threshold:
                dependencies.append({
                    'column': col.name,
                    'partial_key': pk_columns[i],
                    'strength': partial_dependency_strength
                })
```

```
    return dependencies
```

7.3 Treća normalna forma

Treća normalna forma eliminiše tranzitivnu funkcionalnu zavisnost. Tabela je u 3NF ako je u 2NF i nijedan ne-ključni atribut ne zavisi tranzitivno od primarnog ključa.

Detekcija kršenja Treće normalne forme (3NF)

1. **Validacija preduslova** - Proverava postojanje specifikacije tabele, podataka, zaglavlja i definisanog primarnog ključa
2. **Identifikacija kolona** - Razdvaja kolone primarnog ključa od ne-ključnih kolona i proverava da li ima dovoljno ne-ključnih kolona za analizu
3. **Detekcija tranzitivnih zavisnosti** - Poziva algoritam koji testira funkcionalne zavisnosti između ne-ključnih kolona radi pronalaska tranzitivnih putanja
4. **Obrada pronađenih zavisnosti** - Za svaku tranzitivnu zavisnost priprema podatke o determinantnoj koloni, zavisnim kolonama i nazivu nove tabele
5. **Generisanje predloga normalizacije** - Kreira predloge za izdvajanje tranzitivno zavisnih kolona u zasebne tabele sa odgovarajućim opisom i stepenom pouzdanosti
6. **Vraćanje rezultata** - Vraća listu predloga za normalizaciju tabele radi rešavanja narušavanja 3NF

Listing 7.5: check (3NF)

```
def check(self, table_name: str, header: List[str],
          rows: List[List[str]], table_spec: TableSpec) ->
             List[NormalizationSuggestion]:
    suggestions = []
```

```
    if not table_spec or not rows or not header:
        self.logger.warning(f"Insufficient data to analyze 3NF
            for columns_and_types {table_name}")
        return suggestions

    if not table_spec.primary_key:
        self.logger.warning(f"No primary key defined for
           columns_and_types {table_name}")
        return suggestions

    pk_columns = set(table_spec.primary_key.columns)

    all_columns = set(header)
    non_key_columns = all_columns - pk_columns

    if len(non_key_columns) < self.config.
       nf3_min_non_key_columns:
        return suggestions

    transitive_deps = self._find_transitive_dependencies(
       header, rows, pk_columns, non_key_columns)

    for dependency in transitive_deps:
        lookup_table_columns = [dependency['determinant']] +
           dependency['dependents']

        suggestion = NormalizationSuggestion(
            table_name=table_name,
            suggestion_type="3NF",
            description=f"Transitive dependency detected: {',
               '.join(dependency['dependents'])} depends on "
                        f"{dependency['determinant']},
                           creating transitive dependency
                           through primary key. "
                        f"Consider extracting to a separate '{
                           dependency['new_table_name']}'
                           columns_and_types with "
```

```
                        f"columns: {', '.join(
                            lookup_table_columns)}.",
                confidence=dependency['confidence']
            )
            suggestions.append(suggestion)

    return suggestions
```

Algoritam za detekciju tranzitivnih zavisnosti

Funkcija `find_transitive_dependencies` implementira algoritam za identifikaciju narušavanja 3NF kroz tranzitivne funkcionalne zavisnosti:

Tok izvršavanja algoritma

1. **Kreiranje mape indeksa kolona** - Generiše se rečnik koji mapira naziv svake kolone na njen indeks u zaglavlju tabele za efikasan pristup podacima
2. **Testiranje svake ne-ključne kolone kao determinante** - Algoritam prolazi kroz sve ne-ključne kolone i svaku testira kao potencijalnu determinantu (kolonu od koje druge kolone mogu zavisiti)
3. **Testiranje funkcionalnih zavisnosti između kolona** - Za svaki par (determinanta, zavisna kolona):
    - Kreira se mapa koja vezuje vrednosti determinante za vrednosti zavisne kolone
    - Iterira se kroz sve redove tabele i za svaki red se izdvajaju vrednosti determinante i zavisne kolone
    - Proverava se konzistentnost: ako ista vrednost determinante već postoji u mapi, poredi se da li je zavisna vrednost ista
    - Ako se pronađe nekonzistentnost (ista determinanta $\rightarrow$ različite zavisne vrednosti), zavisnost se odbacuje
    - Funkcionalna zavisnost postoji ako svaka vrednost determinante uvek daje istu vrednost zavisne kolone

4. **Validacija tranzitivnosti kroz primarni ključ** - Pre prihvatanja proverava se da li determinanta zavisi od primarnog ključa pozivom `_check_pk_dependency` koji testira PK → determinanta zavisnost

5. **Računanje stepena pouzdanosti** - Poziva se `_calculate_dependency_confidence` koji kombinuje ocene konzistentnosti i jedinstvenosti

6. **Validacija praga pouzdanosti** - Tranzitivna zavisnost se uključuje u rezultate samo ako njena ocena pouzdanosti prelazi konfigurisani prag

7. **Kreiranje opisa zavisnosti** - Za validne zavisnosti kreira se rečnik koji sadrži determinantu, liste zavisnih kolona, opis problema, predloženi naziv nove tabele i numeričku ocenu pouzdanosti

Listing 7.6: Funkcija find_transitive_dependencies

```
def find_transitive_dependencies(self, headers: List[str],
   rows: List[List[str]], pk_columns: set, non_key_columns:
   set) -> List[Dict]:
    dependencies = []

    col_indices = {col: idx for idx, col in enumerate(headers)
       }

    for determinant_col in non_key_columns:
        det_idx = col_indices[determinant_col]

        dependent_cols = []
        for dependent_col in non_key_columns:
            if dependent_col == determinant_col:
                continue

            dep_idx = col_indices[dependent_col]

            dependency_map = {}
            det_values_seen = set()
            has_violation = False

            for row in rows:
```

```
                if len(row) <= max(det_idx, dep_idx):
                    continue

                det_value = row[det_idx].strip()
                dep_value = row[dep_idx].strip()

                det_values_seen.add(det_value)

                if det_value in dependency_map:
                    if dependency_map[det_value] != dep_value:
                            has_violation = True
                            break
                    else:
                        dependency_map[det_value] = dep_value

            if not has_violation:
                is_dependent = 1 < len(det_values_seen) == len
                   (dependency_map)
                self.logger.debug(f"Checking {determinant_col}
                    -> {dependent_col}: {is_dependent}")
                if is_dependent:
                        dependent_cols.append(dependent_col)
                else:
                    self.logger.debug(f"Checking {
                       determinant_col} -> {dependent_col}:
                       False")

        if dependent_cols:
            pk_dependency_exists = self._check_pk_dependency(
               determinant_col, pk_columns, rows, col_indices)

            if pk_dependency_exists:
                confidence = self.
                   _calculate_dependency_confidence(rows,
                   col_indices, determinant_col,
                   dependent_cols,
                                                        pk_columr
```

```
                                                                    )

                if confidence > self.config.
                    nf3_confidence_threshold:
                    dependencies.append({
                        'determinant': determinant_col,
                        'dependents': dependent_cols,
                        'description': f"{', '.join(
                            dependent_cols)} depends on {
                            determinant_col}, creating
                            transitive dependency through
                            primary key",
                        'new_table_name': f"{determinant_col.
                            lower()}_details",
                        'confidence': confidence
                    })

    return dependencies
```

Glava 8

Generisanje SQL/DDL fajla

DDL generator predstavlja finalni modul u procesu konverzije CSV datoteka u SQL tabele. Ovaj komponenta preuzima analizirane specifikacije tabela i generiše kompletne CREATE TABLE skripte koje mogu biti direktno izvršene na ciljanim bazama podataka. Softver implementira DDLGenerator klasu koja podržava više različitih dijalekata baza podataka kroz konfigurisano mapiranje tipova i sintaksne varijacije.

8.1 Glavni algoritam generisanja

Proces generisanja SQL/DDL fajla ima sledeće faze:

1. **Generisanje zaglavlja** - Informacije o dijalektu i vremenu kada je fajl izgenerisan
2. **Iteracija kroz tabele** - Za svaku tabelu se generiše CREATE TABLE naredba
3. **Dodavanje ograničenja** - Uključivanje primarnih i stranih ključeva
4. **Komentari za normalizaciju** - Umetanje predloga za poboljšanje dizajna u vidu komentara

Listing 8.1: Funkcija generate_schema_ddl

```
def generate_schema_ddl(self, tables_specs: Dict[str,
   TableSpec]) -> str:
    ddl_parts = [
```

```
        f"-- Generated DDL for {self.dialect.value}",
        f"-- Generated at: {datetime.now().isoformat()}",
        ""
    ]

    for table_name, table_spec in tables_specs.items():
        table_ddl = self._generate_table_ddl(table_spec)
        ddl_parts.append(table_ddl)
        ddl_parts.append("")

        if table_spec.normalization_suggestions:
            ddl_parts.append("-- NORMALIZATION SUGGESTIONS:")
            for suggestion in table_spec.
               normalization_suggestions:
                ddl_parts.append(f"-- [{suggestion.
                   suggestion_type}] {suggestion.table_name}:
                   {suggestion.description}")
                if hasattr(suggestion, 'confidence') and
                   suggestion.confidence:
                    ddl_parts.append(f"--   Confidence: {
                       suggestion.confidence:.1f}\\n")

    return "\\n".join(ddl_parts)
```

Algoritam kreira listu DDL delova koja se finalno spaja u jedinstven string. Svaka tabela se obrađuje sekvencijalno, a predlozi za normalizaciju se dodaju kao komentari koji omogućavaju da se razmotri poboljšanja dizajna.

8.2 Generisanje CREATE TABLE naredbe

Za svaku tabelu se generiše kompletna CREATE TABLE naredba:

Listing 8.2: Funkcija generate_table_ddl

```
def generate_table_ddl(self, table_spec: TableSpec) -> str:
    lines = [f"CREATE TABLE {self._quote_identifier(table_spec
       .name)} ("]

    column_lines = []
```

```
    for col in table_spec.columns:
        col_ddl = self._generate_column_ddl(col)
        column_lines.append(f"    {col_ddl}")

    pk_columns = table_spec.primary_key.columns if table_spec.
       primary_key else None
    if pk_columns:
        pk_cols = ", ".join(self._quote_identifier(col) for
           col in pk_columns)
        column_lines.append(f"    PRIMARY KEY ({pk_cols})")

    for fk in table_spec.foreign_keys:
        fk_ddl = self._generate_foreign_key_ddl(fk)
        column_lines.append(f"    {fk_ddl}")

    lines.append(",\\n".join(column_lines))
    lines.append(");")

    return "\\n".join(lines)
```

Struktura se generiše kroz sledeće korake:

1. CREATE TABLE naredba sa nazivom tabele
2. Generisanje definicija kolona
3. Dodavanje primarnog ključa
4. Iteracija kroz sve strane ključeve

Generisanje definicija kolona

Svaka kolona se transformiše u SQL definiciju na sledeći način:

Listing 8.3: Funkcija generate_column_ddl

```
def generate_column_ddl(self, col: ColumnSpec) -> str:
    parts = [self._quote_identifier(col.name)]

    sql_type = format_type_with_size(col.data_type, col.
       size_spec, self.dialect)
```

```
    parts.append(sql_type)

    if col.is_auto_increment:
        if self.config.auto_increment_syntax:
            parts.append(self.config.auto_increment_syntax)

    if not col.nullable:
        parts.append("NOT NULL")

    return " ".join(parts)
```

8.3 Dodavanje stranih ključeva

Strani ključevi se generišu kroz standardnu FOREIGN KEY sintaksu:

Listing 8.4: Funkcija generate_foreign_key_ddl

```
def generate_foreign_key_ddl(self, fk: ForeignKeySpec) -> str:
    fk_cols = ", ".join(self._quote_identifier(col) for col in
        fk.columns)
    ref_cols = ", ".join(self._quote_identifier(col) for col
       in fk.referenced_columns)

    return (f"FOREIGN KEY ({fk_cols}) "
            f"REFERENCES {self._quote_identifier(fk.
               referenced_table)} ({ref_cols})")
```

Ova sintaksa je standardizovana kroz različite dijalekte, omogućavajući portabilnost generisane DDL skripte.

8.4 Predlozi za normalizaciju

Generator automatski uključuje predloge za normalizaciju kao komentare u DDL fajlu:

Listing 8.5: Sugestije u vezi normalizacje

```
if table_spec.normalization_suggestions:
    ddl_parts.append("-- NORMALIZATION SUGGESTIONS:")
```

```
    for suggestion in table_spec.normalization_suggestions:
        ddl_parts.append(f"-- [{suggestion.suggestion_type}] {
           suggestion.table_name}: {suggestion.description}")
        if hasattr(suggestion, 'confidence') and suggestion.
           confidence:
            ddl_parts.append(f"--   Confidence: {suggestion.
               confidence:.1f}\\n")
```

Komentari sadrže:

- **Tip predloga** - 1NF, 2NF, 3NF oznake
- **Opis problema** - Konkretno objašnjenje kršenja normalne forme
- **Pouzdanost** - Numerička ocena verovatnoće problema

8.5 Primer kompletnog izlaza

DDL izlaz generiše sledeću strukturu:

Listing 8.6: Primer izlaza

```
-- Generated DDL for postgresql
-- Generated at: 2024-03-15T14:30:45.123456

CREATE TABLE "student_course" (
    "StudentID" VARCHAR(255) NOT NULL,
    "CourseID" VARCHAR(255) NOT NULL,
    "StudentName" VARCHAR(255) NOT NULL,
    "CourseName" VARCHAR(255) NOT NULL,
    "Grade" CHAR(2) NOT NULL,
    "Credits" INTEGER NOT NULL,
    PRIMARY KEY ("StudentID", "CourseID")
);

-- NORMALIZATION SUGGESTIONS:
-- [2NF] student_course: Column 'StudentName' has partial
   dependency on 'StudentID'
--   (part of composite key ['StudentID', 'CourseID']) instead
    of depending on the full key
```

```
--   (strength: 100.0%). Consider extracting to a separate '
   students' table with columns:
--   StudentID, StudentName.
--   Confidence: 100.0%
-- [2NF] student_course: Column 'CourseName' has partial
   dependency on 'CourseID'
--   (part of composite key ['StudentID', 'CourseID']) instead
    of depending on the full key
--   (strength: 100.0%). Consider extracting to a separate '
   courses' table with columns:
--   CourseID, CourseName.
--   Confidence: 100.0%
-- [2NF] student_course: Column 'Credits' has partial
   dependency on 'CourseID'
--   (part of composite key ['StudentID', 'CourseID']) instead
    of depending on the full key
--   (strength: 100.0%). Consider extracting to a separate '
   course_credits' table with
--   columns: CourseID, Credits.
--   Confidence: 100.0%
```

Glava 9

Zaključak

CSV datoteke su široko rasprostranjeni format za čuvanje i razmenu tabelarnih podataka u različitim domenima, od poslovnih aplikacija do naučnih istraživanja. Automatsko pretvaranje ovih datoteka u strukturirane relacione baze podataka predstavlja ključni korak u procesu upravljanja podacima. Razvoj softvera koji može pouzdano da generiše predlog projekta baze podataka iz proizvoljnih CSV datoteka zahteva kombinovanje znanja iz oblasti baza podataka, statistike i algoritamskih pristupa. Tokom razvoja ovog softvera korišćen je iterativni pristup testiranja na različitim tipovima CSV datoteka, što je omogućilo postepeno poboljšavanje preciznosti svih implementiranih algoritma.

9.1 Rezime urađenog posla

U ovom radu je razvijen celokupni softver za automatsko pretvaranje CSV datoteka u predlog projekat (dela) baze podataka. Softver rešava problem automatskog prepoznavanja strukture podataka kombinovanjem različitih pristupa analize podataka, statističkih metoda i heurističkih tehnika.

Glavni doprinosi rada uključuju algoritme za automatsko prepoznavanje zaglavlja CSV fajlova, prepoznavanje tipova podataka, otkrivanje primarnih ključeva (prirodni, kompozitni, surogat) i stranih ključeva, kao i implementaciju algoritma za normalizaciju koji otkriva narušavanja 1NF, 2NF i 3NF.

Razvijeni softver generiše gotov SQL/DDL fajl za ulazne CSV fajlove sa podrškom za više SQL dijalekata (SQLite, PostgreSQL, MySQL, SQL Server). Modularna arhitektura omogućava lako proširivanje i održavanje softvera.

9.2 Moguća dalja unapređenja softvera

Buduća unapređenja softvera mogu ići u nekoliko pravaca:

- **Veličina kompozitnih ključeva** može biti promenjena da uzima u obzir i više od dve kolone
- **Primena mašinskog učenja** mogla bi poboljšati preciznost detekcije ključeva i tipova podataka kroz zamenu trenutnih heurističkih pristupa naprednim algoritmima klasifikacije
- **Implementacija obrade u tokovima** omogućila bi rad sa velikim datotekama koje se ne mogu učitati u memoriju odjednom
- **Automatska optimizacija DDL-a** za bolje performanse kroz kreiranje odgovarajućih indeksa i strategija podele podataka predstavlja prirodni pravac za dalje razvijanje softvera
- **Grafički korisnički interfejs** olakšao bi korišćenje softvera krajnjim korisnicima koji nisu tehnički stručnjaci
- **Proširivanje podrške na dodatne SQL dijalekte** povećalo bi praktičnu primenu softvera

Bibliografija

[1] Python Software Foundation. CSV biblioteka, 2025. on-line at: https://docs.python.org/3/library/csv.html.

[2] Python Software Foundation. Python, 2025. on-line at: https://www.python.org/.

[3] Python Software Foundation. re biblioteka, 2025. on-line at: https://docs.python.org/3/library/re.html.

[4] Python Software Foundation. statistics biblioteka, 2025. on-line at: https://docs.python.org/3/library/statistics.html.

[5] The PostgreSQL Global Development Group. PostgreSQL, 2025. on-line at: https://www.postgresql.org/.

[6] D. Richard Hipp. SQLite, 2025. on-line at: https://www.sqlite.org/.

[7] IBM. Foreign Key Constraint, 2025. on-line at: https://www.ibm.com/docs/en/db2/12.1.0?topic=constraints-foreign-key-referential.

[8] IBM. Normal Forms, 2025. on-line at: https://www.ibm.com/docs/en/tpfdf/1.1.3?topic=database-normalization.

[9] IBM. Primary Key Constraint, 2025. on-line at: https://www.ibm.com/think/topics/primary-key.

[10] Microsoft. SQL Server, 2025. on-line at: https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver17.

[11] Leah Nguyen. Normal Forms Examples, 2025. on-line at: https://medium.com/@ndleah/a-brief-guide-to-database-normalization-5ac59f093161.

[12] Oracle. MySQL, 2025. on-line at: https://www.mysql.com/.