FD-Finder
Dokumentácia

Version 0.0.1

Richard Hvizdoš

November 2, 2025

Contents

List of Code Listings

1 Úvod

FD-Finder je jednoduchá webová aplikácia na hľadanie funkčných závislostí v dátach. Podporuje formáty CSV a JSON, obsahuje viacero algoritmov s možnosťou porovnávania ich výsledkov a taktiež opakovaného spúšťania. Samotný kód je možné stiahnuť z GitHub repozitára[1] a následne spustiť.

Dokumentácia je členená na: používateľskú dokumentáciu (sekcia 2), ktorá Vás zoznámi s jej funkciami a použitím, programátorskú dokumentáciu (sekcia 3), ktorá Vás prevedie jej kódom a členením a výsledkami a experimentami (sekcia 4), ktorá popíše modifikácie algoritmov a porovnanie s pôvodnými implementáciami.

1.1 Motivácia

Funkčné závislosti (FD) vyjadrujú vzťah medzi atribútmi datasetu. FD je v tvare X->A, čo naznačuje, že hodnoty atribútu/-ov X unikátne určujú hodnoty atribútu A. Napríklad platí funkčná závislosť: rodné číslo->dátum narodenia.

Samotné funkčné závislosti je možné využiť pri profilovaní rozsiahlych multimodelových dát, nahradení niektorých atribútov funkciou alebo pri čistení dát a vytváraní schémy [4].

Motiváciou a samotným cieľom bolo vytvoriť jednoduchý nástroj s rôznymi algoritmami, ktoré budú schopné škálovať a hľadať funkčné závislosti v BIG DATA s podporou viacerých dátových formátov.

Pretože určité nástroje a algoritmy existujú, ale majú zvyčajne nedostatky v podpore dátových formátov, škálovateľnosti alebo sú optimalizované iba na jeden parameter, rozhodol som sa, že tieto algoritmy upravím na elimináciu týchto nedostatkov a zistené výsledky budú použité na ďalší vývoj.

Príkladom je projekt Metanome[2], ktorý implementuje dané algoritmy a ponúka užívateľskú aplikáciu. Síce niektoré algoritmy sú paralelizované, ale nie sú distribuované, a preto sú menej škálovateľné. Samotná analýza algoritmov bola pripravená už v špecifikácii[3], a preto nie je súčasťou záverečného reportu.

[1] https://github.com/DonRiccardo/FD-finder

[2] https://hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html

[3]

1.2 Analýza

Analýzou potrieb a podobných nástrojov sa dospelo k nasledujúcim požiadavkám.

Funkčné požiadavky kladené na aplikáciu sú:

- selekcia vstupného súboru, v ktorom sa budú hľadať FDs so špecifikáciou formátu (napr. CSV, JSON),
- upresnenie, či súbor obsahuje hlavičku, špecifikácia oddeľovača hodnôt pre CSV súbory,
- zadanie limitu na počet spracovávaných záznamov zo vstupného súboru, počtu záznamov, ktoré budú preskočené vo vstupnom súbore a limit na veľkosť LHS,
- určenie výstupu algoritmu,
- spoľahlivá aplikácia,
- možnosť nastavenia parametrov pre algoritmy,
- zobrazenie výsledkov a štatistík po dokončení algoritmu.

Medzi **nefunkčné požiadavky** patrí:

- vysoký výkon algoritmov pri hľadaní FDs,
- stabilný modul a algoritmy s vhodným výpisom chabových hlášok,
- jednoduchá inštalácia a spustenie
- podrobná používateľská a programátorská dokumentácia,
- modulárna architektúra s rozdelenín nástroja na FE a BE.

Samotná aplikácia aktuálne cieli na akademické prostredie a použitie na ďalší výskum, preto nebude obsahovať používateľské role a ďalšie potrebné časti v SW potrebné pri komerčnom použití.

Potrebnými prvkami aplikácie sú: možnosť pridať dataset, vytvoriť nad datasetom job so špecifickými parametrami, spustiť job a prípadne ho zastaviť, zobraziť výsledky dokončeného jobu a taktiež zobraziť štatistiky jobu (napríklad dĺžka behu, počet nájdených FDs, spotreba pamäte, zaťaženie CPU). Samozrejmosťou je možnosť odstrániť job alebo dataset.

2 Používateľská dokumentácia

Používateľská dokumentácia obsahuje všetky potrebné informácie pre používateľa ako napríklad spôsob inštalácie aplikácie, jej spustenie, postup práce v aplikácii a v neposlednom rade jej dostupné funkcie pre hľadanie funkčných závislostí v datasetoch.

Dokumnetácia je členená na sekcie: Inštalácia a spustenie (2.1), Popis aplikácie (2.2) s popisom práce v aplikácii, Stránky aplikácie (2.3) s popisom jednotlivých stránok aplikácie, ďalšie sekcie Vás prevedú hlavnými funkciami aplikácie (Pridanie datasetu v 2.4, Vytvorenie jobu v 2.5, Spustenie a zastavenie jobu v 2.6, Zobrazenie a stiahnutie výsledkov v 2.7 a Skončenie a odinštalovanie v 2.8).

2.1 Inštalácia a spustenie

Ako prvé je potrebné naklonovať GitHub repozitár[4] či už prostredníctvom príkazu alebo stiahnutím .ZIP priečinku (a jeho následným extrahovaním) do vášho počítača.

Spustenie aplikácie prebieha prostredníctvom Dockeru[5], ktorý je potrebné mať nainštalovaný a spustený na zariadení.

V Dockeri sa budú vytvárať kontajnery, v ktorých pobežia jednotlivé súčasti aplikácie. Aplikáciu je nutné spustiť príkazom (zadaným v koreňovom adresári aplikácie)

```
docker compose up
```

alebo ako nútený rebuil príkazom

```
docker compose up --build
```

Následne sa začnú vytvárať kontajnery, čo potrvá istý čas.

Používateľ interaguje s frontendom vo webovom prehliadači, ktorý je dostupný na adrese `http://localhost:5173`. Ako používať aplikáciu je popísané v kapitole 2.

[4] https://github.com/DonRiccardo/FD-finder
[5] https://www.docker.com/

2.2 Popis aplikácie

Po spustení aplikácie a otvorení stránky `http://localhost:5173` sa vám zobrazí nasledujúca úvodná obrazovka zobrazená na obrázku 1. Okrem úvodnej stránky má aplikácia ďalšie tri stránky, ktoré sú popísané v uvedených skeciách:

- **Datasets** zobrazujúca uložené datasety s možnoťou ich pridania, *(sekcia 2.3.1)*
- **Jobs** zobrazujúca vytvorené joby s možnosťou vytvorenia ďalších, *(sekcia 2.3.2)*
- **Job Results** zobrazujúca výsledky a štatistiky jednotlivých jobov. *(sekcia 2.3.3)*

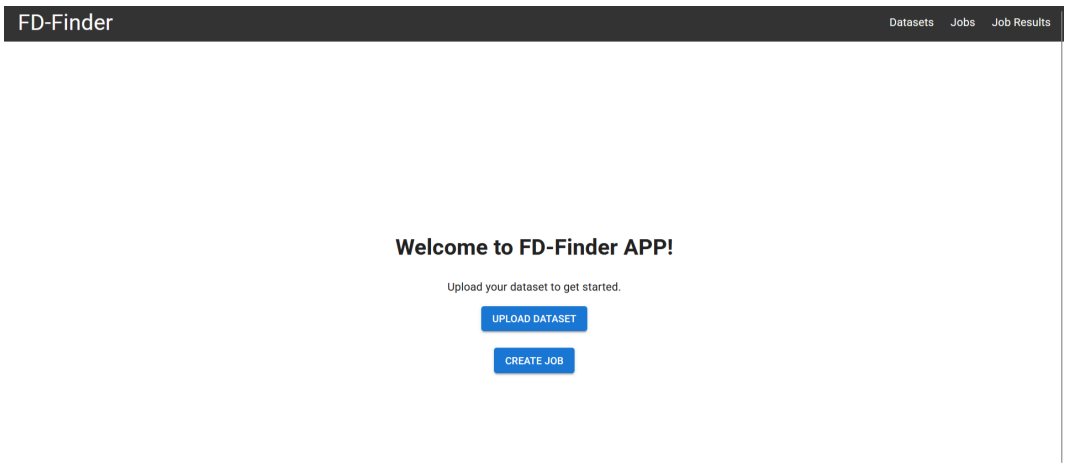

Figure 1: Úvodná obrazovka aplikácie

Nižšie je napísaný postup práce v aplikácii, potrebný k tomu, aby bolo možné zvolený dataset spracovať a určiť funkčné závislosti, ktoré sa v ňom nachádzajú. Každý krok je podrobnejšie popísaný v jemu priradenej sekcii nižšie.

Postup práce v aplikáci:

1. Pridanie požadovaného datasetu na spracovanie. *(sekcia 2.4)*
2. Vytvorenie jobu pre datý dataset na vybraných algoritmoch. *(sekcia 2.5)*
3. Spustenie jobu a čakanie na jeho skončenie. *(sekcia 2.6)*
4. Zobrazenie výsledkov a štatistík výpočtu. *(sekcia 2.7)*

2.3 Stránky aplikácie

V nasledujúcich podsekciách sú popísané jednotlivé stránky aplikácie s podrobným popisom ich častí a funkcií.

2.3.1 Stránka Datasets

Stránka **Datasets** obsahuje tabuľku so záznamami pre každý uložený dataset v aplikácii.

Záznam obsahuje ID, unikátny názov datasetu, formát súboru uloženého datasetu, popis datasetu, počet záznamov a atribútov v datasete, oddeľovač hodnôt a hlavičku datasetu (platné iba pre datasety vo formáte CSV) a nakoniec čas uloženia datasetu.

Na konci každého záznamu sú tlačidlá (na obrázku 2 označené číslami 2, 3 a 4), ktoré majú nasledovné funkcie:

- tlačidlo **2** - vytvorenie jobu pre daný dataset,
- tlačidlo **3** - stiahnutie uloženého datasetu,
- tlačidlo **4** - odstránenie daného datasetu z aplikácie *(Upozornenie: po odstránení datasetu nebude možné vykonať vytvorené joby pre odstránený dataset)*.

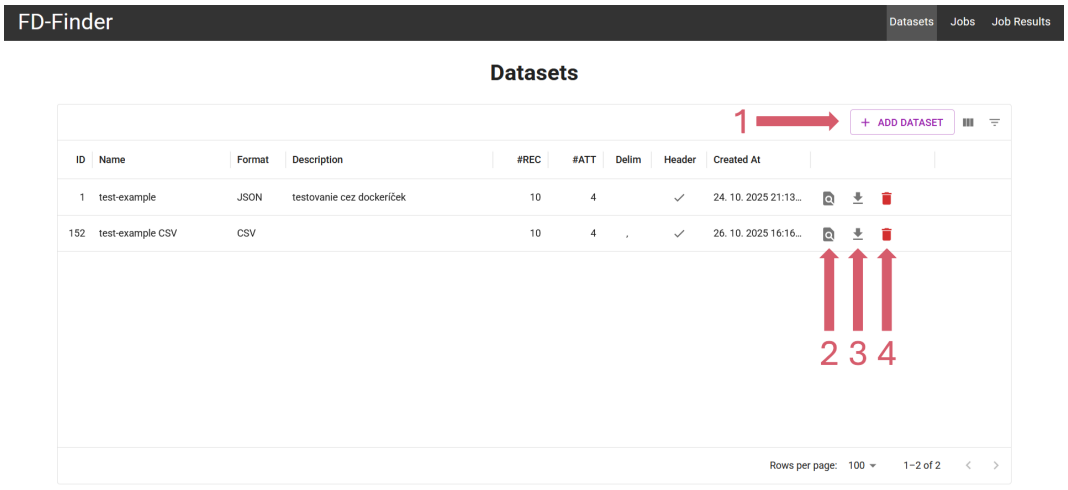

Figure 2: Stránka Datasets

Pridanie datasetu je možné vykonať po stlačení tlačidla **+ ADD DATASET** (na obrázku 2 označeného číslom 1). Popis ako pridať dataset sa nachádza v sekcii 2.4.

2.3.2 Stránka Jobs

Stránka **Jobs** obsahuje tabuľku so záznamami pre každý vytvorený job v aplikácii.

Záznam obsahuje: ID, unikátny názov jobu, popis jobu, zoznam algorimov, na ktorých bude/bol spustený, názov spacovávaného datasetu, stav výpočtu jobu, počet záznamov, ktoré budú preskočené (záznamy sa preskakujú od začiatku datasetu, bez hlavičky), maximálny počet spracovávaných záznamov (po preskočení), maximálna veľkosť ľavej strany funkčnej závislosti. Na konci je čas vytvorenia jobu a jeho poslednej aktualizácie.

Každý záznam má pridelené svoje tlačidlá (na obrázku 3 označené číslami 2, 3, 4, a 5), ktoré majú nasledovné funkcie:

- tlačidlo **2** (zelený play button) - spustenie jobu,
- tlačidlo **3** (červený krížik) - zastavenie jobu *(pozn.: na obrázku je šedé, pretože jeho funkcia bola zablokovaná)*,
- tlačidlo **4** - zobrazenie výsledkov jobu na stránke **Job Results** (sekcia 2.3.3),
- tlačidlo **5** - odstránenie jobu.

Vytvorenie jobu je možné vykonať po stlačení tlačidla **+ ADD JOB** (na obrázku 3 označeného číslom 1). Popis ako vytvoriť job sa nachádza v sekcii 2.5.

2.3.3 Stránka Job Results

Stránka **Job Results** zobrazuje výsledky jednotlivých jobov. Stránka je rozdelená na pravú časť, ktorá zobrazuje informácie o jobe (na obrázku 4 označená číslom **2**) a na ľavú časť, na ktorej je možné zobraziť štatistiky, nájdené funkčné závislosti a grafy ukazujúce zaťaženie procesoru a pamäte.

Informácie sa zobrazia po výbere jobu z ponuky (na obrázku 4 označená číslom **1**) alebo po kliknutí na tlačidlo **4** na stránke **Jobs** (sekcia 2.3.2). Aby sa výsledky mohli zobraziť, job musí byť v stave **DONE**. Ako je možné vidieť

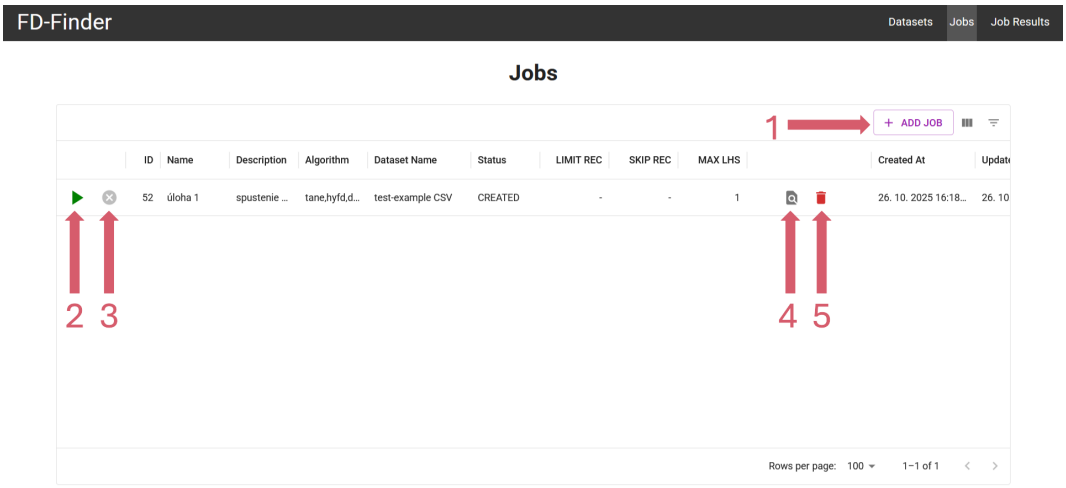

Figure 3: Stránka Jobs

na obrázku 4, stav označený číslom **2** je **RUNNING**, a teda nie sú zobrazené žiadne výsledky jobu.

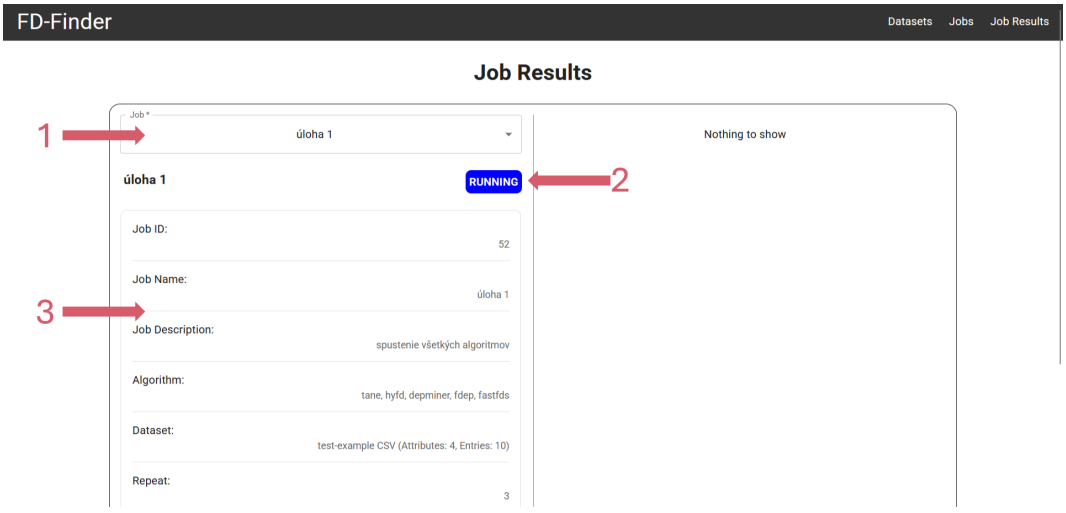

Figure 4: Stránka Job Results

Po dokončení jobu je možné zobraziť **štatistiky** behu výpočtu pre každý algoritmus samostatne (ukázané na obrázku 5), s výsledkami: dĺžka výpočtu, zaťaženie procesoru, využitie pamäte a počet nájdených funkčných závislostí. Hodnoty sú napísané v tomto poradí: *minimálna hodnota zo všetkých behov / priemerná hodnota cez všetky behy / maximálna hodnota zo všetkých behov*.

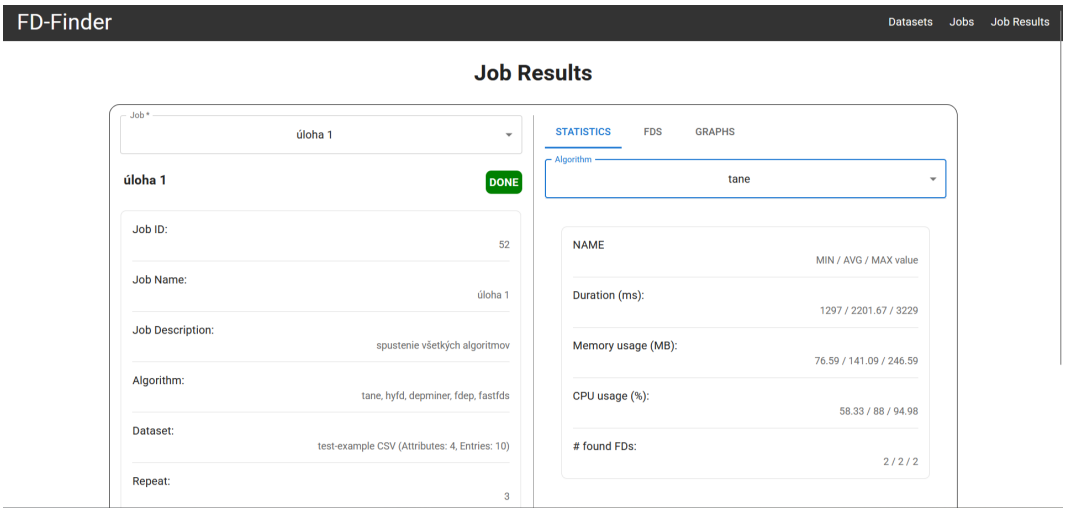

Figure 5: Stránka Job Results - štatistiky

Nájdené funkčné závislosti budú zobrazené až po výbere algoritmu a následne aj konkrétneho behu a stlačení tlačidla označeného na obrázku 6 číslom **1**. Tlačidlo označené číslom **2** na obrázku 6 slúži na stiahnutie TXT súboru s nájdenými funkčnými závislosťami.

V poslednej časti ľavej strany stránky sú zobrazené dva **grafy**, ako je možné vidieť na obrázku 7. Prvý graf ukazuje pre všetky algoritmy priemerné zaťaženie procesoru v priebehu výpočtu. Druhý graf ukazuje pre všetky algoritmy priemernú spotrebu pamäte v priebehu výpočtu.

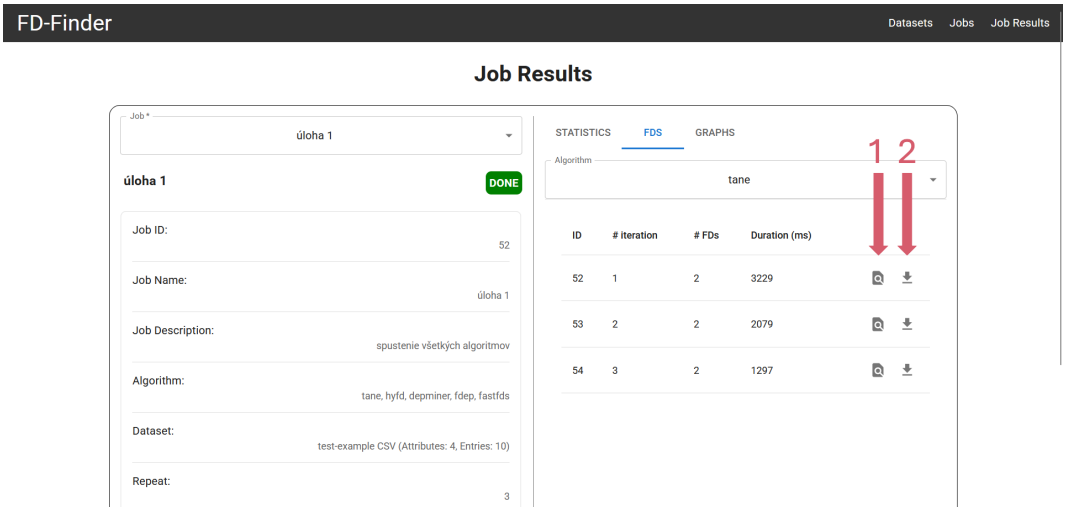

Figure 6: Stránka Job Results - funkčné závislosti

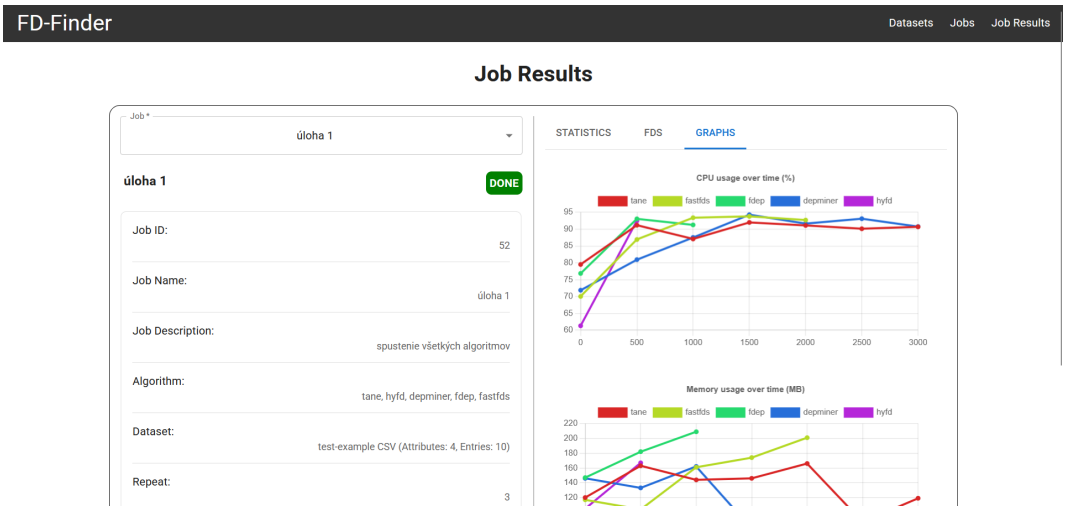

Figure 7: Stránka Job Results - grafy

2.4 Pridanie datasetu

Pridanie datasetu do aplikácie prebieha po kliku na tlačidlo **+ ADD DATASET** na stránke **Datasets** (sekcia 2.3.1) alebo po klinutí na tlačidlo **UPLOAD DATASET** na úvodnej stránke. Následne sa zobrazí stránka (zobrazená na obrázku 8) s možnosťou pridať dataset. Aplikácie podporuje pridanie datasetov iba vo formáte **CSV** alebo **JSON**.

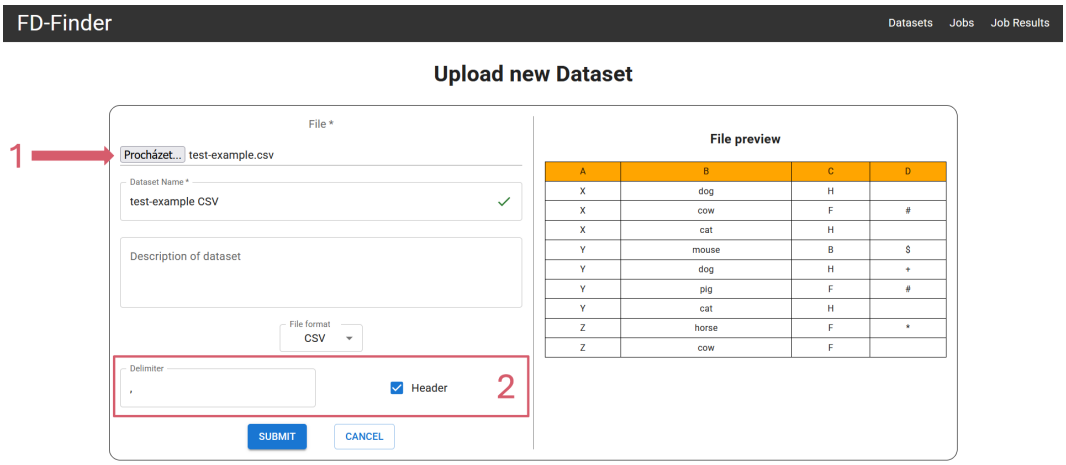

Figure 8: Stránka na pridanie datasetu

Dataset vyberiete z dialógového okna zobrazeného po stlačení poľa alebo

tlačidla označeného na obrázku 8 číslom **1**. Po výbere sa načíta dataset a na pravej strane sa zobrazí **ukážka datasetu**. Podľa prípony sa určí formát súboru, ktorý je nutné overiť.

Je potrebné vybrať pre dataset **unikátny názov**, ktorý sa bude na stránkach aplikácie používať ako jeho identifikátor. Na jeho korektnosť budete upozornený zelenou fajkou na konci riadku. Voliteľnou časťou je popis datasetu.

V prípade súborov vo formáte **CSV** budú zobrazené dve polia (na obrázku 8 označené číslom **2**) na správne určenie **oddeľovača hodnôt** (obvykle je oddeľovačom čiarka) a **hlavičky súboru** (v prípade jej prítomnosti a nezakliknutí alebo neprítomnosti a zakliknutí môže dôjsť k chybným výsledkom). Správnosť oddeľovača hodnôt zistíte tak, že po jeho výbere sa **ukážka datasetu** (File preview) zobrazí ako pekne formátovaná tabuľka. Prítomnosť hlavičky je vyznačená oranžovým riadkom v ukážke.

Pridanie datasetu dokončíte kliknutím na tačidlo **Submit**. Následne budete presmerovaný na stránku **Datasets** (sekcia 2.3.1), a pre novopridaný dataset sa dopočíta počet jeho atribútov a záznamov.

2.5 Vytvorenie jobu

Vytvorenie jobu prebieha po kliku na tlačidlo **+ ADD JOB** na stránke **Jobs** (sekcia 2.3.2) alebo po kliku na tlačidlo **2** na konci záznamu daného datasetu na stránke **Datasets** (sekcia 2.3.1) (v tomto prípade už bude vyplnené pole Dataset daným datasetom). Následne sa zobrazí stránka s možnosťou vytvoriť job zobrazená na obrázku 9.

Podobne ako pri pridávaní datasetu, každý job má svoj **unikátny názov**, ktorý sa bude na stránkach aplikácie používať ako jeho identifikátor. Na jeho korektnosť budete upozornený zelenou fajkou na konci riadku. Voliteľnou časťou je popis jobu.

Hlavnými krokmi je **výber algoritmov**, ktoré budú dataset spracovávať a hľadať funkčné závislosti, a **výber datasetu**. Počet vybraných algoritmov nie je obmedzený, dataset môže byť vybraný iba jeden. Algoritmy aj dataset sa vyberajú z ponuky dostupných algoritmov a datasetov.

Ďalšími voliteľnými položkami je určenie **Limit entries** (označuje maximálny počet záznamov, ktoré budú spracované z datasetu), **Skip entries** (označuje počet záznamov, ktoré budú preskočené od začiatku datasetu bez hlavičky),

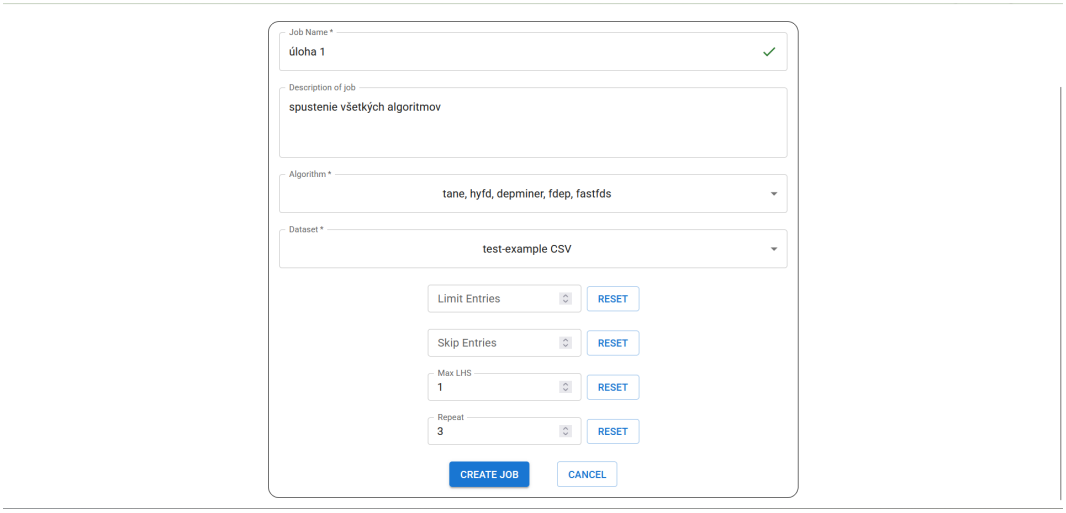

Figure 9: Stránka Job Create

Max LHS (označuje maximálnu dĺžku ľavej časti funkčných závislostí, ktoré budú výstupom výpočtu) a **Repeat** (označuje počet opakovaných spustení každého zvoleného algoritmu nad daným datasetom).

Limit entries a Skip entries musia mať hodnotu v rozmedzí 0 až #záznamov, hodnota Max LHS musí byť nezáporná a Repat musí mať hodnotu v rozmedzí 1 až 20. V prípade chybného zadanie je možný reset danej hodnoty stlačením príslušného tlačidla **RESET**.

K vytvoreniu jobu dôjde po stlačení tlačidla **CREATE JOB**, následne budete presmerovaný na stránku **Jobs** (sekcia 2.3.2).

2.6 Spustenie a zastavenie jobu

Spustenie jobu je možné na stránke **Jobs** *(sekcia 2.3.2)* kliknutím na **zelený play button naľavo**. Po štarte jobu sa zmení jeho stav na **WAITING**, a tak môžete vedieť, že job bude spustený na jednotlivých algoritmov (po dokončení predchádzajúcich jobov).

V prípade, že job beží príliš dlhú dobu, alebo výsledky už nepotrebujete, môžete **job zastaviť** kliknutím na **červený krížik naľavo**. Job bude následne zastavený a bude ho možné znovu spustiť.

Každý job má priradený jeden z nasledujúcich stavov:

- **CREATED** - job je vytvorený,
- **WAITING** - job je registrovaný na algoritmoch a čaká na spracovanie,
- **RUNNING** - job je spustený a prebieha výpočet,
- **CANCELLED** - job bol zastavený používateľom,
- **FAILED** - pri behu jobu došlo k chybe,
- **DONE** - job bol úspešne dokončený.

K stavu **FAILED** mohlo dôjsť tým, že ste odstránili dataset, ktorý mal job spracovávať. V tomto prípade bude táto skutočnosť napísaná na stránke **Job Results** *(sekcia 2.3.3)* v informáciách o datasete a takýto job môžete odstrániť. V opačnom prípade je chyba spôsobená v samotnej aplikácii a job môžete skúsiť spustiť neskôr.

2.7 Zobrazenie a stiahnutie výsledkov

Výsledky jobu je možné zobraziť na stránke **Job Results** *(sekcia 2.3.3)* a následne vybraním jobu z ponuky. Druhou možnosťou je kliknúť na tlačidlo **4** na konci záznamu pre daný job na stránke **Jobs** *(sekcia 2.3.2)*;

Nájdené funkčné závislosti je možné stiahnuť na stránke **Job Results** *(sekcia 2.3.3)* v časti **FDS** ako TXT súbor pre konkrétny beh vybraného algoritmu po kliknutí na tlačidlo **2**.

2.8 Skončenie aplikácie, odinštalovanie

Aplikáciu môžete zatvoriť v prehliadači, ale samotný server je potrebné **zastaviť** buď v aplikácii Docker alebo po stlačení kláves ctrl+C v termináli, v ktorom ste aplikáciu spustili.

Aplikáciu **odinštalujete** odstránením jej priečinku a vymazaním image v Dockeri.

3 Programátorská dokumentácia

Aplikácia **FD-Finder** umožňuje používateľom ukladať datasety a v nich následne prostredníctvom algoritmov hľadať funkčné závislosti. V tejto sekcii sa zoznámite s architektúrou aplikácie, jej členením a základným postupom práce. Ďalšie komentáre a informácie sú dostupné v zdrojovom kóde aplikácie.

Momentálne aplikácia funguje iba lokálne s DB úložiskom na lokálnom file systéme.

Aplikácia bola testovaná manuálne na vybraných datasetoch a overované správanie aplikácie a algoritmov pri jednotlivých situáciách.

3.1 Architektúra

Aplikácia FD-Finder je navrhnutá princípom microservices, teda skladá sa zo servisov, kde každý má svoju špecifickú úlohu (popísané nižšie v sekcii 3.3), je to možné vidieť na obrázku 10. Výhodou tohto prístupu je jednoduché nahradenie alebo pridanie nového service.

Používateľ pracuje vo webovom prehliadači na stránke `http://localhost:5173`, tá následne komunikuje so **Serverom**, **JobService** a **Dataservice**. Každý algoritmus je samostatný service, ktorý vykonáva joby a umožňuje odlišné nastavenia pre samotné algoritmy.

3.2 Použité technológie

Backend je vyvíjaný v jazyku Java *(verzia 21)*, použitím Spring Boot *(verzia 3.5.6)*, Apache Spark *(verzia 4.0.1)*. Dáta ukladané do H2 databáze použitím JPA (v prípade Job a Data services).

Frontend je vyvíjaný v jazyku JavaScript vo frameworku Vite použitím React.

3.3 Microservices

Aplikácia je rozdelená na **8 microservices** v backende (z toho je 5 microservices pre algoritmy). Každá microservice komunikuje s **Eureka serverom**, na ktorom sa registrujú a majú pridelený svoj port.

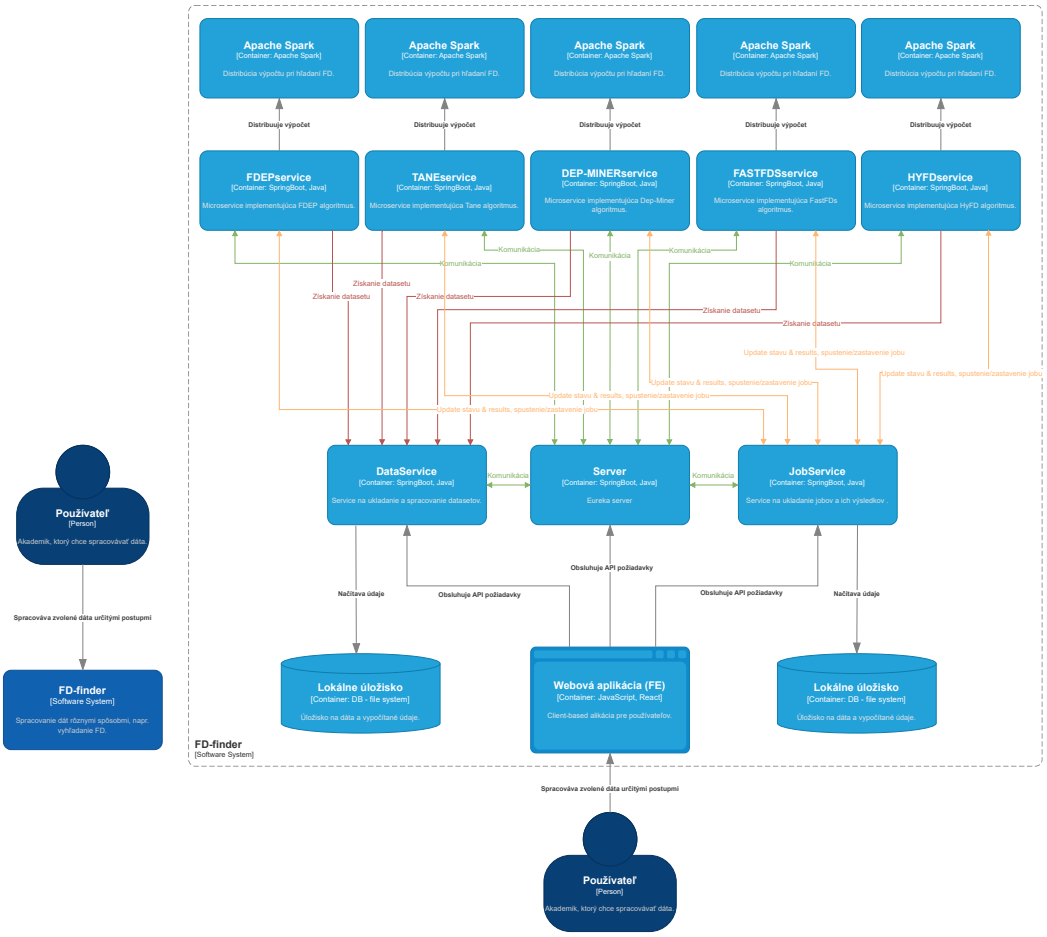

Figure 10: C4 model FD-Finder

Backend navyše obsahuje **DemoAlgService** microservice, ktorý slúži na rýchle a jednoduché vytvorenie nového microservice. Obsahuje základné triedy a metódy, ktoré taktiež rešpektujú štruktúru uložených dát. Popis ako vytvoriť a pridať nový microservice je v sekcii 3.3.5.

3.3.1 Eureka server

Je to Netflix Eureka server, na ktorý sa registrujú jednotlivé microservices a je možné získať ich názvy a adresy.

Navyše má pridaný endpoint `/algorithms`, ktorý vráti zoznam názvov aktuálne registrovaných algoritmov v sieti.

3.3.2 Data service

Microservice Data má za úlohu ukladať **metadáta** o datasetoch a samotné **súbory**. Súbory sa ukladajú do lokálneho file systému v priečinku `datasets/"formát"/timestamp_hash-súboru` v priečinku DataService, kde formát je csv alebo json. Po uploade sa spustí asynchrónna metóda na načítanie súboru a určenia počtu atribútov a záznamov. Je to buď metóda `DatasetService.processNewCSVDataset(Dataset dataset)` v prípade CSV súboru alebo `DatasetService.processNewJsonDataset(Dataset dataset)` v prípade JSON súboru.

Samotné metadáta sú uložené v H2 DB v triede `Dataset` (zobrazenej v ukážke 1), ktorá má nasledovné atribúty. **Id** je kľúč v DB, ale pre užívateľov komfort sa vo fronende používa unikátny **name**, ktorý používateľ sám vytvorí. Atribúty **delim** a **header** sú podstatné, iba ak je súbor vo formáte CSV.

3.3.3 Job service

Microservice Job má za úlohu ukladať vytvorené Joby a následne aj ich výsledky po spracovaní. Taktiež sa cez tento microservice spracovávajú výsledky na štatistiky a dáta pre grafovú vizualizáciu, spúšťajú a zastavujú jednotlivé joby.

Samotné metadáta o jobe sú uložené v H2 DB v triede `Job` (zobrazenej v ukážke 2), kde je opäť kombinácia **id** ako kľúča v DB a unikátny **name** pre používateľov komfort. Každý Job má celkový **status**, ktorý sa vyhodnotí zo všetkých statusov jednotlivých **JobResult**.

List<**JobResult**> sa vytvorí po vytvorení Jobu pomocou metódy `Job.generateJobResultsObjects()`, kde sa pre každý zvolený algoritmus a jeho iteráciu vytvorí jeden objekt **JobResult** (zobrazený v ukážke 3) s názvom daného algoritmu (**algorithm**) a číslom iterácie (**iteration**).

Pri vykonávaní výpočtu sa aktualizuje stav, nájdené funkčné závislosti apod. iba pre konkrétny beh (**JobResult**) v **jobResults**, z tejto zmeny sa prípadne upraví aj samotný **Job**. Výsledné funkčné závislosti sa ukladajú do súborov v lokálnom file systéme v priečinku `jobs/results/job-ID-ALG-run-N-foundFDs.txt` v priečinku JobService,

```java
@Entity
public class Dataset {

    @Id
    @GeneratedValue
    private Long id;
    @NotEmpty
    @Column(unique = true)
    private String name;
    private String description;
    @NotNull
    private FileFormat fileFormat;
    private Long size;

    private int numAttributtes=0;
    private Long numEntries=0L;

    private String originalFilename;
    private String hash;
    private LocalDateTime createdAt;
    private long savedAt;

    private String delim;
    private boolean header;
}
```

Listing 1: Dataset class

kde ID je id jobu, ALG označuje názov algoritmu, ktorý bol použitý na nájdenie FDs a N je poradie iterácie.

Pri výpočtoch sa zaznamenávajú **snapshots** do triedy **Snaphot** (zobrazenej v ukážke 4), z ktorých sa následne získavajú dáta a spracúvajú sa na štatistiky prostredníctvom metódy `JobService.prepareStatisticForJob(Job job)` a grafovú vizualizáciu prostredníctvom metódy `JobService.prepareSnapshotDataForVisualization(Job job)`.

```java
@Entity
public class Job {

    @Id
    @GeneratedValue
    private Long id;
    @NotEmpty
    private String jobName;
    private String jobDescription;
    @Enumerated(EnumType.STRING)
    private JobStatus status;
    @NotEmpty
    private List<String> algorithm;
    @Positive
    @Max(20)
    private int repeat;
    @NotNull
    private Long dataset;
    @NotEmpty
    @NotNull
    private String datasetName;
    private int limitEntries = -1;
    private int skipEntries = -1;
    private int maxLHS = -1;
    @JsonIgnoreProperties("job")
    @OneToMany(mappedBy = "job", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<JobResult> jobResults = new ArrayList<>();

    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}
```

Listing 2: Job class

```java
@Entity
public class JobResult {

    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "job_id")
    @JsonIgnoreProperties("jobResults")
    private Job job;
    @Enumerated(EnumType.STRING)
    private JobStatus status;
    @Positive
    private int iteration;
    @NotEmpty
    private String algorithm;
    private Long startTime;
    private Long endTime;
    private Long duration;
    @PositiveOrZero
    private int numFoundFd;
    @ElementCollection(fetch = FetchType.EAGER)
    private List<Snapshot> snapshots;
}
```

Listing 3: JobResult class

```java
@Embeddable
public class Snapshot {
    private Long timestamp;
    private Long usedMemory;
    private Double cpuLoad;
    private Integer threadCount;
}
```

Listing 4: Snapshot class

Stavy Job, JobResult

,

Každý Job a JobResult má jeden stav z nasledujúcich:

- **CREATED** - job je vytvorený,

- **WAITING** - job je registrovaný na algoritmoch a čaká na spracovanie,
- **RUNNING** - job je spustený a prebieha výpočet,
- **CANCELLED** - job bol zastavený používateľom,
- **FAILED** - pri behu jobu došlo k chybe,
- **DONE** - job bol úspešne dokončený.

Stavy **jobResults** sa menia nasledovne:

- na začiatku sú všetky stavy CREATED,
- pri spustení výpočtu sa stavy zmenia na WAITING,
- ak nejaký algoritmus začne výpočet pre nejakú iteráciu, tak konkrétnemu **JobResult** sa zmení stav na RUNNING,
- ak používateľ zastaví výpočet, tak všetky stavy **jobResults** sa zmenia na CANCELLED,
- ak nejaká iterácia dospela k chybe, zmení sa stav konkrétneho **JobResult** na FAILED,
- ak iterácia úspešne skončila, konkrétny stav **JobResult** je DONE.

Stav **Job** je získany zo všetkých stavov **jobResults** nasledovne:

- ak **aspoň jeden** zo stavov **jobResults** je RUNNING/FAILED -> **Job** bude mať nastavený daný stav,
- ak sú **všetky** stavy DONE -> **Job** bude mať stav DONE.

V opačnom prípade je stav hromadne nastavený pri vytvorení jobu (CREATED), zrušení jobu (CANCELLED) alebo pri spustení (WAITING).

3.3.4 Microservices algoritmov

Zoznam dostupných algoritmov, každý vo vlastnej microservice:

- **FDEP** [1] ako fdepService,
- **Dep-Miner** [3] ako dep-minerService,
- **FastFDs** [6] ako fastfdsService,
- **HyFD** [5] ako hyfdService,

- **TANE** [2] ako taneService.

Jednotlivé microservices pre algoritmy majú rovnakú štruktúru a líšia sa iba zvoleným algoritmom, ktorý spracováva dataset. Každá microservice má dva endoinpty (jeden na spustenie a druhý na zastavenie výpočtu), na ktoré posiela požiadavky iba microservice **Job** (sekcia 3.3.3). Odpoveď na požiadavku sa pošle okamžite, ale samotný výpočet prebieha asynchrónne.

Algotitmy v aplikácii boli distribuované pomocou Apache Spark[6] a testované manuálne na vybraných dátových sadách v GitHub repozitári[7].

(Pozn.: nižšie spomínaný Controller aj Service, majú pre každý algoritmus v názve predponu daného názvu algoritmu.)

Joby, ktoré je potrebné vykonať sa ukladajú do **queue** v metóde `Controller.registerNewJob(JobDto job)`, z ktorej sa postupne vyberajú a spracujú v metóde `Service.startNext()`. Trieda **JobDto** je okresaná verzia triedy **Job** (ukážka 2) iba na potrebné atribúty.

Samotný výpočet prebieha v metóde `Service.runJob(JobDto job)`, kde sa ako prvé stiahnu metadáta o datasete do triedy **DatasetDto** a následne sa uloží aj samotný súbor, ktorý sa po skončení jobu vymaže. Trieda **DatasetDto** je taktiež okresaná verzia triedy **Dataset** (ukážka 1) iba na potrebné atribúty. Service má pre tieto algoritmy vytvorenú inštanciu **Apache Spark**, pomocou ktorého sa distribuuje výpočet algoritmu. Počas výpočtu sa zachytávajú snapshoty s informáciami o zaťažení procesoru alebo aj použitou pamäťou prostredníctvom triedy **JobMonitorService**.

Po skončení každej iterácie algoritmu sa získané výsledky odosielajú na **JobService** v metóde

```
Service.processOneJobResult(JobResultsFDs jobResultsFDs,
long jobId)
```

a následne sa aktualizuje stav iterácie v metóde

```
Service.updateStatus(Long jobResultId, JobStatus status,
ServiceInstance serviceInstanceJob)
```

, ktorá pošle požiadavku na **JobService**.

[6] https://spark.apache.org/
[7] https://github.com/DonRiccardo/DP-algoritmy

3.3.5 Pridanie nového service

V priečinku FD-finder existuje microservice **demoAlgService**, čo je pripravený microservice na pridanie nového algoritmu. Obsahuje pripravené triedy a metódy na spracovanie algoritmov a požiadaviek. Je nutné iba **upraviť** nasledujúce **časti kódu** a **pridať implementáciu algoritmu**.

Upraviť v kóde pri pridaní nového algoritmu:

- vhodné je zmeniť názvy súborov, tried a samotného package,
- v **DemoAlgController.java** upraviť `@RequestMapping("/demoalg")` na názov nového algoritmu,
- v **DemoAlgService.java** v metóde `cancelJob(long jobId)` implementovať zrušenie už spusteného jobu *(v závislosti na implementácii vášho algoritmu)*,
- v **DemoAlgService.java** v metóde `runJob(JobDto job)` inicializovať algoritmus a následne do `List<_FunctionalDependency> foundFds` priradiť nájdené funkčné závislosti,
- v **JobDto.java** zmeniť v metóde `getJobResults()` názov algoritmu,
- v **application.yml** zmeniť číslo portu na ešte nepoužité, a názov na daný algoritmus *(Upozornenie: názov musí zostať v tvare **algservice-názovAlgoritmu**)*.

Algoritmus umiestnite do service, v ktorom je priečinok `/de/../container` s triedami na reprezentáciu funkčných závislostí, ktoré prijíma metóda na spracovanie výsledkov. V prípade potreby je možné kód doplniť o ďalšie časti ako Apache Spark, či iné frameworky a triedy.

Následne je potrebné vytvoriť pre nový algoritmus **Dockerfile** a do **docker-compose.yml** pridať nový service. Frontend nie je potreba upravovať, v ňom sa načítajú automaticky dostupné algoritmy (z tohto dôvodu je nutné dodržať názov novej service).

3.4 Frontend

Frontend pracuje na porte 5173 a je to pre používateľa jediný pristupový bod do aplikácie. Frontend komunikuje iba so serverom, JobService a DataService na získanie potrebných dát. Je rozdelený na viacero stránok:

- **Homepage()** - základná stránka s dvomi tlačidlami,

- **datasets-add.jsx** - pridanie datasetu do aplikácie,
- **datasets-all.jsx** - zobrazenie tabuľky so všetkými datasetmi,
- **jobs-add.jsx** - vytvorenie nového jobu,
- **jobs-all.jsx** - zobrazenie tabuľky so všetkými jobmi,
- **jobs-results.jsx** - zobrazenie výsledkov jobu.

3.4.1 Homepage()

Nachádza sa v súbore `App.jsx`, je to uvítacia stránka s tlačidlami na pridanie datasetu a vytvorenie nového jobu. Zobrazí sa aj po kliknití na názov FD-Finder.

3.4.2 datasets-add.jsx

Na tejto stránke je možné vyplniť formulár, po odoslaní ktorého sa na **DataService** pridá nový dataset. Keďže **name** datasetu má byť unikátne (používa sa ako identifikátor pre komfort používateľa), tak je potrebné z **DataService** načítať názvy uložených datasetov do premennej **datasetNames**.

`handleFileInputChange(event)` zabezpečuje pri výbere a zmene súboru jeho načítanie (podľa prípony) do premennej **rawRows**, na jeho neskoršiu ukážku používateľovi.
Pri tomto načítaní sa nastaví formát do premennej **fileFormat** a taktiež sa vytvorí **name** datasetu, z názvu súboru, a priradí sa do **fileName**. FileName je overované na jeho unikátnosť v `React.useEffect()` a výsledok je uložený v **isNameUnique**. Unikátnosť name sa používateľovi zobrazuje krížokom/fajkou na konci políčka a taktiež prípadným červeným orámovaním ak name nie je unikátne.

Formát súboru, by mal užívateľ overiť. Ak je formát súboru CSV, tak sa na konci formuláru zobrazia políčka na určenie oddeľovača hodnôt a špecifikáciu, či súbor obsahuje hlavičku. Pri ich zmene sa nanovo generuje ukážka súboru v premennej **preview**.

3.4.3 datasets-all.jsx

Stránka zobrazuje jednoduchú tabuľku so všetkými datasetmi uloženými v aplikácii s možnosťou zakliknúť, ktoré stĺpce budú zobrazené a taktiež

filtrovať datasety.

Ako prvé sa načítajú datasety z **DataService** a po úprave (napr. úprava formátu timestamp) sa uložia do premennej **rows**, z ktorej sa dáta načítajú do tabuľky.

Každý dataset v tabuľke má na konci záznamu tri tlačidlá s nasledujúcimi ikonami a funkciami:

- `<FindInPageIcon />` - odkaz na stránku **jobs-add.jsx** s predvyplneným datasetom, v ktorom sa budú hľadať FDs,
- `<DownloadIcon />` - stiahnutie uloženého súboru,
- `<DeleteIcon />` - odstránenie datasetu z aplikácie.

3.4.4 jobs-add.jsx

Na tejto stránke je možné vyplniť formulár, po ktorého odoslaní sa na **JobService** pridá nový Job. Keďže **jobName** má byť unikátne (používa sa ako identifikátor pre komfort používateľa), tak je potrebné z **JobService** načítať názvy uložených jobov do premennej **jobNames**. Unikátnosť **jobName** sa overuje v `React.useEffect()` a výsledok sa ukladá do premennej **isNameUnique**.

Do premennej **availableAlgorithms** sa zo serveru získajú názvy dostupných algoritmov (bez predpony **algservice-**) a do premennej **availableDatasets** sa získajú dostupné datasety z **DataService**, ktoré sa pre zjednodušenie práce uložia vo forme dictionary (kľúčom je unikátne name datasetu).

Ak je premenná **datasetId** vyplnená, to znamená, že stránka sa zobrazila s požiadavkou vyhľadať funkčné závislosti v danom datasete a premenná **dataset** sa nastaví na požadovaný názov datasetu.

Na základe vybraného datasetu sa polia na určenie **limitEntries** a **skipEntries** ohraničia maximálnou hodnotou, t.j. počet záznamov v datasete a musia byť nezáporné. Hodnota **repeat** je ohraničená 1 až 20 a **maxLhs** musí byť nezáporné.

3.4.5 jobs-all.jsx

Stránka zobrazuje jednoduchú tabuľku so všetkými jobmi uloženými v aplikácii s možnosťou zakliknúť, ktoré stĺpce budú zobrazené a taktiež filtrovať

joby.

Ako prvé sa načítajú joby z **JobService** a po úprave (napr. úprava formátu timestamp) sa uložia do premennej **rows**, z ktorej sa dáta načítajú do tabuľky.

Každý job v tabuľke má na začiatku záznamu dve tlačidlá a na konci ďalšie dve tlačidlá s nasledujúcimi ikonami a funkciami:

- `<PlayArrowIcon />` - spustenie jobu,
- `<CancelIcon />` - zastavenie jobu,
- `<FindInPageIcon />` - odkaz na stránku **job-results.jsx** aby zobrazila výsledky daného jobu,
- `<DeleteIcon />` - odstránenie jobu z aplikácie.

3.4.6 jobs-results.jsx

Stránka zobrazuje vybrané metadáta pre Job a jeho výsledky. Ako prvé sa načítajú dáta pre dostupné joby z **JobService** a uložia sa do **availableJobs**. V prípade že je nastavený vstupný parameter **jobId**, tak sa nastaví **selectedJob** na požadovaný job.

Následne je možné stiahnuť dáta o datasete a ak je job `DONE`, tak sa stiahnu štatistiky, ktoré je potrebné preformátovať, a dáta pre graf z**JobService**. Používateľ vyberá z troch podstránok na zobrazenie: štatistiky, funkčné závislosti a grafy. Štatistiky pre vybraný algoritmus sú zobrazené ako `<List>`, funkčné závislosti sú zobrazené pre vybraný algoritmus až po výbere jeho iterácie a následne sú stiahnuté, grafy sa vykreslia z už stiahnutých dát.

Grafy sú dva, jeden pre priemerné zaťaženie CPU počas iterácií algoritmu a druhý zobrazuje priemerné využitie pamäte počas iterácií algoritmu. Každý algoritmus má vlastnú krivku so svojou farbou. Grafy sú vykresľované prostredníctvom knižnice react-chartjs-2[8].

3.5 API dokumentácia

Úplná API dokumentácia je dostupná v GitHub repozitári[9], nižšie si popíšeme základy.

[8] `https://react-chartjs-2.js.org/`
[9] `https://github.com/DonRiccardo/FD-finder`

Každý zo services, server a frontend pracujú na rozdielom porte, ktoré sú vypísané nižšie a majú vlastný názov, ktorým sú identifikovaný na serveri (okrem FE). Pre services taktiež nechýba ani mapovanie napríklad v podobe `/jobs` pre všetky endpointy v JobService alebo `/datasets` pre všetky endpointy DataService.

Používané porty a názvy jednotlivých services:

- **FE** frontend pracuje na porte 5173,
- **Eureka server** pracuje na porte 8761,
- **DataService** ako `dataservice` pracuje na porte 8081,
- **JobService** ako `jobservice` pracuje na porte 8082,
- **fdepService** ako `algservice-fdep` pracuje na porte 8083,
- **dep-minerService** ako `algservice-depminer` pracuje na porte 8084,
- **fastfdsService** ako `algservice-fastfds` pracuje na porte 8085,
- **hyfdService** ako `algservice-hyfd` pracuje na porte 8086,
- **taneService** ako `algservice-tane` pracuje na porte 8087.

4 Výsledky, experimenty

Výsledkom projektu je vyššie popísaná aplikácia **FD-Finder**, ale taktiež distribuované algoritmy, ktoré sú popísané v tejto sekcii s príslušnými experimentami na vyhodnotenie ich náročnosti.

4.1 Algoritmy

Inšpiráciou algoritmov bola ich implementácia od Hasso-Plattner-Institut[10], ktorá sa následne upravovala na distribúciu výpočtu prostredníctvom frameworku Apache Spark[11].

Ďalej sú popísané zmeny a princíp výpočtu jednotlivých algoritmov. Pozmenené metódy/triedy sú v oranžových boxoch, modré objekty znamenajú, že metóda/trieda zostala v pôvodnej implementácii a v zelenom objekte sú nájdené FDs.

4.1.1 FDEP

Aloritmus FDEP má nasledujúcu implementáciu zobrazenú na obrázku 11. Trieda `FdepSpark` zabezpečuje načítanie datasetu podľa formátu súboru, nastavenie a spustenie `FdepSparkAlgorithm`, ktorá vykonáva samotný výpočet.

Distribúcia výpočtu je jednoduchá, a to iba rozdelenie vstupných dát (riadkov) do skupín a z týchto skupín sa vytvoria dvojice. Dvojice sa paralelizujú a pre kombinácie riadkov z dvojice sa vypočítajú neplatné funkčné závislosti. Následne sa s týmito neplatnými funkčnými závislosťami pracuje rovnako, ako v pôvodnej implementácii. Zjednodušený postup výpočtu je zobrazený na obrázku 12.

[10] `https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html`

[11] `https://spark.apache.org/`

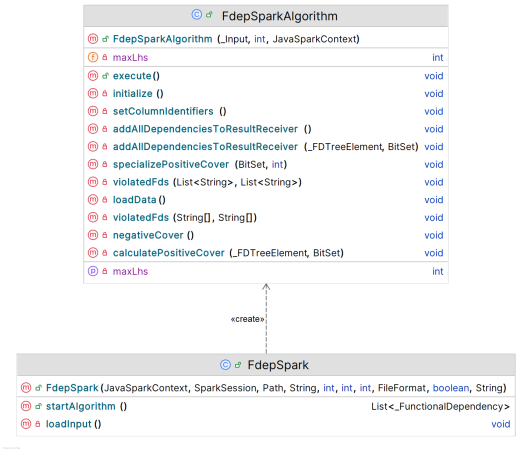

Figure 11: Fdep diagram tried

FdepSpakAlgorithm.execute()

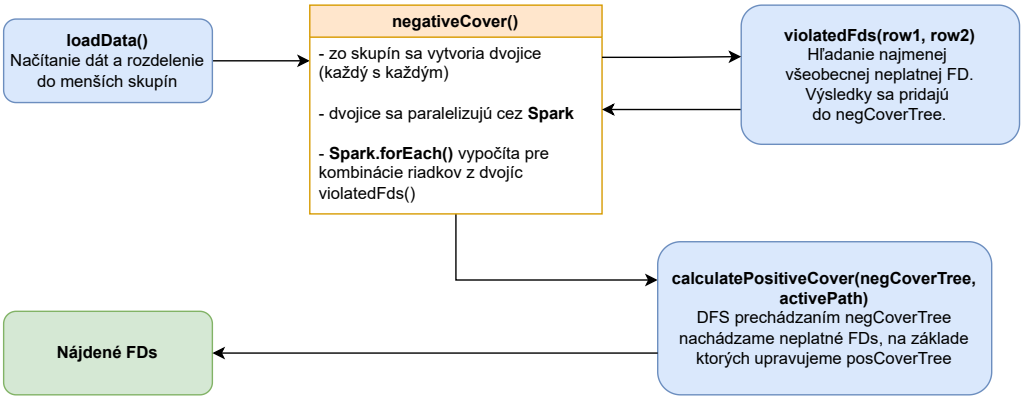

Figure 12: FdepSparkAlorithm diagram

4.1.2 Dep-Miner

Algoritmus Dep-Miner má nasledujúci implementáciu zobrazenú na obrázku 13. Trieda `DepMinerSpark` zabezpečuje načítanie datasetu podľa formátu súboru, nastavenie a spustenie `DepMinerSparkAlgorithm`, ktorá spúšťa jednotlivé generátory a predáva medzivýsledky ako parametre.

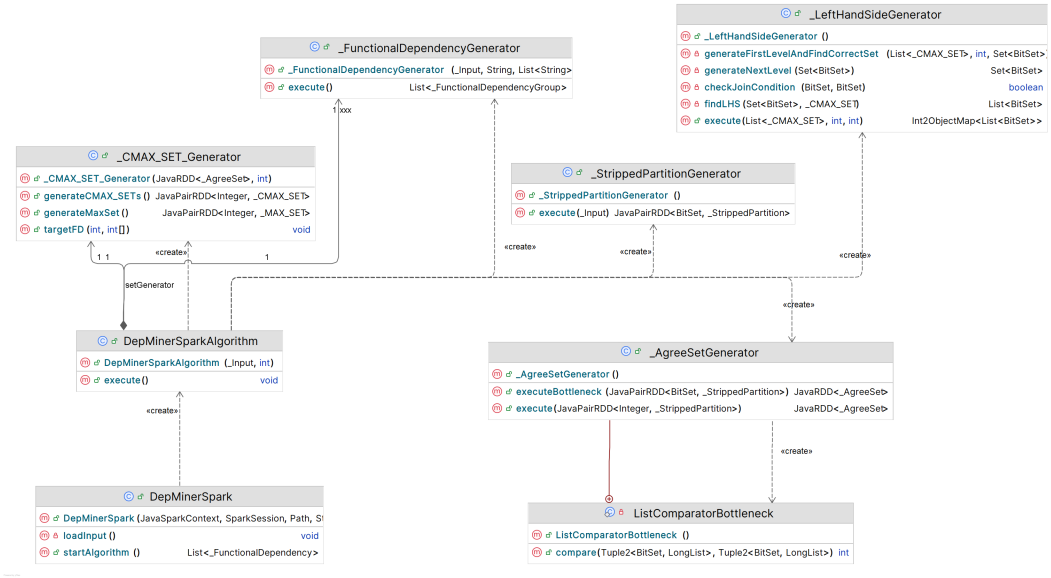

Figure 13: Dep-Miner diagram tried

Ako je možné vidieť na obrázku 14 väčšina generátorov je distribuovaná prostredníctvom frameworku Apache Spark. Ako prvý je **_StrippedPartitionGenerator**, ktorý z načítaných dát vytvára Stripped partition pre každý z atribútov.

Z vytvorených Stripped Partition sa vytvárajú v **_AgreeSetGenerator** jednotlivé Agree Set, ale cez tzv. "bottleneck", pretože sa všetky partitions spoja. Tento krok je podstatný na ich vzájomné porovnávanie a vytváranie Maximálnych tried ekvivalencie, z ktorých sa vytvoria dvojice IDčok (riadkov). Tých bude vďaka tomuto prístupu menej a z nich sa vytvárajú jednotlivé Agree Set.

Následne sa pre každý atribút vytvorí MaxSet ako množina AgreeSet, ktoré neobsahujú daný atribút a nie sú podmnožinou iných. Doplnkom MaxSet sa stane CMaxSet, z ktorých sa vypočítavajú možné LHS funkčných závislostí. Tie sa overujú na netriviálnosť a minimalitu.

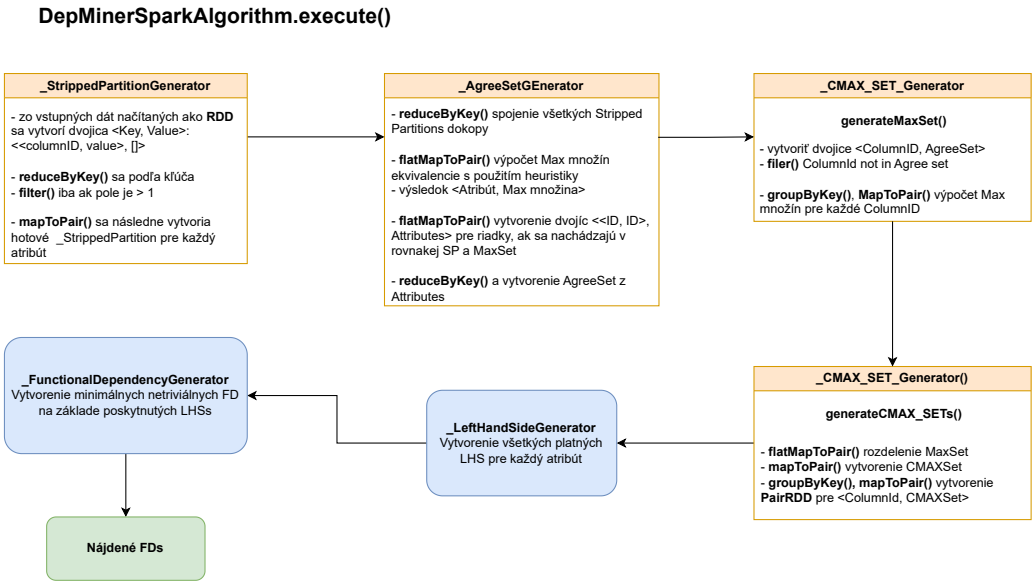

Figure 14: DepMinerSparkAlorithm diagram

4.1.3 FastFDs

Algoritmus FastFDs má nasledujúcu implementáciu zobrazenú na obrázku 15. Trieda `FastFdsSpark` zabezpečuje načítanie datasetu podľa formátu súboru, nastavenie a spustenie `FastFdsSparkALgorithm`, ktorá spúšťa jednotlivé generátory a predáva medzivýsledky ako parametre.

Na obrázku 16 môžete vidieť, že generátory sú opäť distribuované prostredníctvom frameworku Apache Spark.

Ako v predošlom prípade, keďže FastFDs je upravený DepMiner, tak sa ako prvé načítajú dáta a vytvoria **StrippedPartitions**, ale počas ich vytvárania sa do **relationships** pridávajú záznamy. Relationships obsahuje pre každé rowID záznam (index SP, atribút), pre ktorú **StrippedPartion**, v ktorom atribúte sa nachádzal.

Podobne začína výpočet ako v DepMiner **AgreeSet**, cez tzv. "bottleneck", ale s rozdielom, že sa samotné **AgreeSet** nevytvárajú a rovno sa vytvoria **DifferenceSets** použitím vytvorených **relationships**.

Následne sa vytvárajú **DifferenceSet** DwithoutA bez atribútu A a overuje sa, či takáto DwithoutA nie je prázdna. Ak by bola, tak by to značilo, že funkčná závislosť bude triviálna. Pre takto vytvorené množiny DwithoutA sa spustí

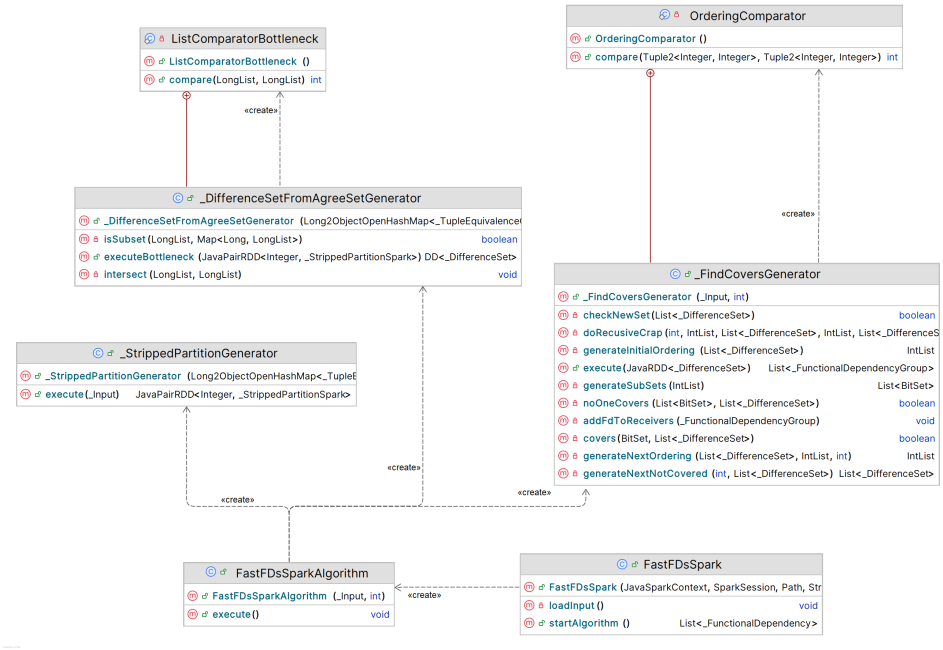

Figure 15: FastFds diagram tried

DFS priechod stromom podľa atribútov, ktoré sa usporadúvajú podľa počtu DwithoutA, ktoré pokrývajú. Takýmto spôsobom sa hľadajú výsledné funkčné závislosti.

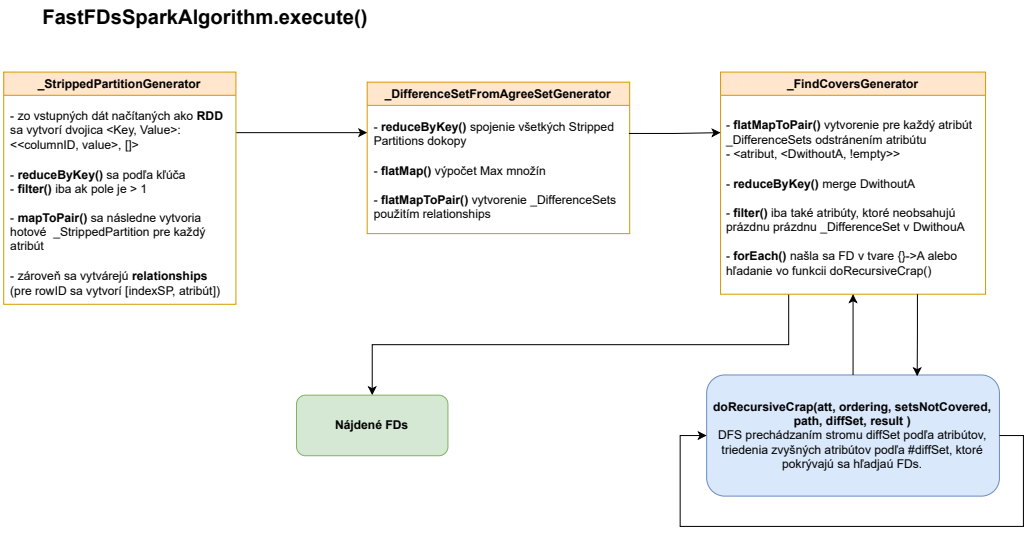

Figure 16: FastFdsSparkAlgorithm diagram

4.1.4 HyFD

Algoritmus HyFD má nasledujúci implementáciu zobrazenú na obrázku 17. Trieda `HyFDSpark` zabezpečuje načítanie datasetu podľa formátu súboru, nastavenie a spustenie `HyFDSparkAlgorithm`, ktorá spúšťa jednotlivé fázy a procesy výpočtu.

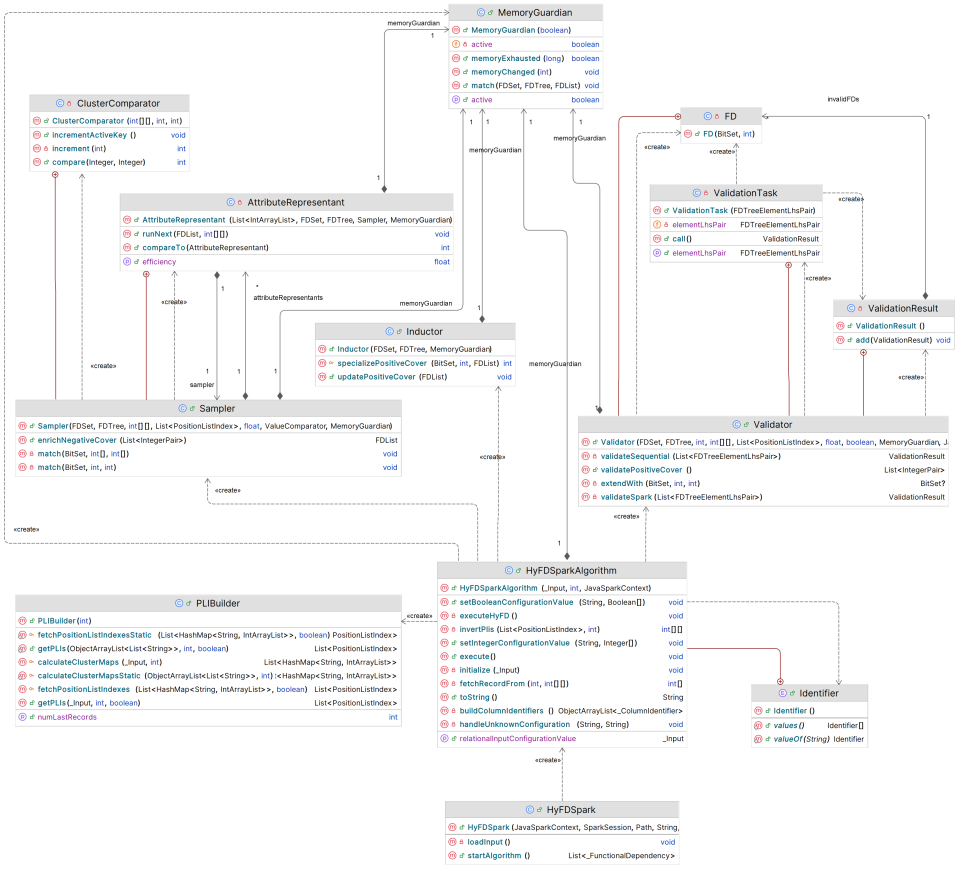

Figure 17: HyFD diagram tried

Postup výpočtu algoritmu kombinuje viaceré prístupy, a to: riadkovo efektívny (lattice traversal) a stĺpcovo efektívny (dependency induction). Preto má algoritmus iný prístup k hľadaniu funkčných závislostí, čo môžeme vidieť na obrázku 18.

Podobne ako u predchádzajúcich, ako prvé sa vytvoria stripped partitions, z ktorých sa vypočítajú **invertedPlis** a z nich následne **compressedRecords**, ktoré sa budú používať pri výpočte. Spočívajú dvoch indexoch: podľa rowID a atribútu sa získa index partition, v ktorej sa daný row nachádza.

Prvá fáza výpočtu je **Sampler**, ktorý prostredníctvom focused sampling vyberá dáta a hľadá neplatné funkčné závislosti. Z týchto neplatných funkčných závislostí **Inductor** generuje nových kandidátov na platné FDs.

Tie už overuje **Validator**, ktorý má distribuovanú metódu na overenie validity FDs. Proces validácie prebieha po leveloch (zdola hore). Ak sa nájde nekorektná FD, tak sa v ďalšom leveli nahradí všetkými jej minimálnymi a netriviálnymi špecializáciami použitím štandardných pruning rules pre lattice traversal algoritmy. Výpočet sa zastaví vo Validatorovi ak sa dosiahne MaxLHS alebo nový level je prázdny.

O vhodné uloženie a občasné prerezávanie sa stará nepovinná komponenta **Guardian**, ktorá v prípade veľkého využitia pamäť prereže a zmenší uložený prefixový strom.

Na prechod medzi fázami Sampler+Inductor a Validator sa používa **Sampling efficiency** (počet nových objavených neplatných FDs na porovnávania) a **Validation efficiency** (počet objavených korektných FDs na validáciu). Ak nejaká hodnota spadne pod zadaný treshold, tak sa algoritmus presunie do opačnej fázy.

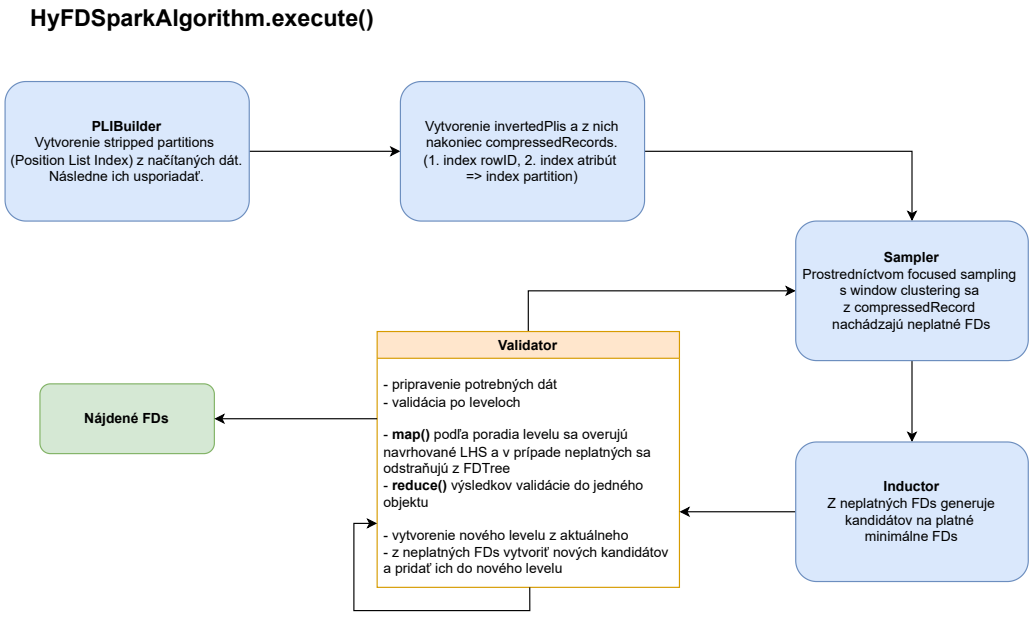

Figure 18: HyFDSparkAlgorithm diagram

4.1.5 TANE

Algoritmus TANE má nasledujúci implementáciu zobrazenú na obrázku 19. Trieda `TaneSpark` zabezpečuje načítanie datasetu podľa formátu súboru, nastavenie a spustenie `TaneSparkAlgorithm`, ktorá spúšťa jednotlivé metódy výpočtu.

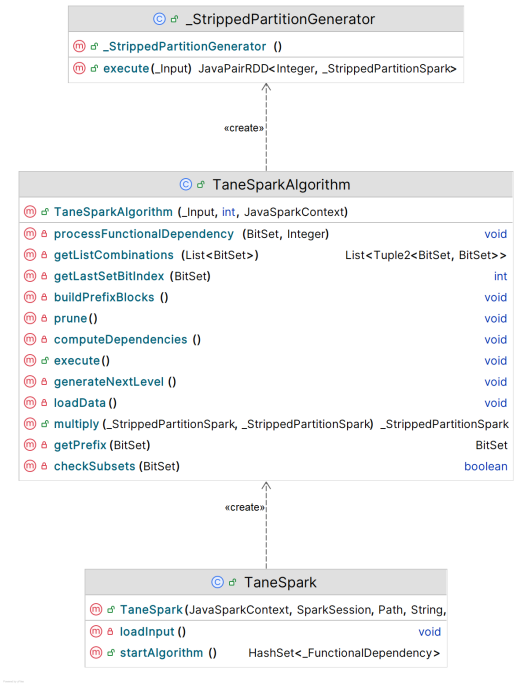

Figure 19: Tane diagram tried

TANE je hlavným predstaviteľom lattice traversal algoritmu, t.j. vyhľadávací priestor prechádza po vrstvách (leveloch) a z aktuálneho generuje ďalší. To je možné vidieť na obrázku 20, kde po vytvorení **Stripped Partitions** sa inicializujú **level0** a **level1**. Levely obsahujú kombináciu atribútov, vždy o 1 väčšiu ako na predchádzajúcom leveli. Začína sa od kombinácie veľkosti 0.

Pre validné kombinácie z **level1** sa vypočítajú z RHS kandidátov intersection a ak má rovnaký **error** (je to celkový počet rowIDs v SP - počet partitions), tak sa našla funkčná závislosť.

Ďalším krokom je orezanie lattice, ktorá sa ale v skutočnosti neorezáva, aby bolo možné vytvoriť všetky kombinácie. Orezané prvky sú označené `false` v `isValid()` a s ich výpočtom sa nepokračuje. V opačnom prípade sa z RHS kandidátov odstráni kombinácia K, postupne sa do nej pridáva

jeden zo zvyšných kandidátov **A** a odstraňuje jeden z pôvodných atribútov v kombinácii **B**. Pre takto vytvorené sa postupne aktualizuje **intersection**. Ak na konci instersection obsahuje A, tak sa našla funkčná závislosť.

V poslednom kroku stačí z prefixových blokov vytvoriť nové kombinácie, inicializovať C+ a pre nové kombinácie vynásobiť pôvodné Stripped Partitions.

Algoritmus končí, ak boli použité všetky atribúty, nový **level1** je prázdny alebo sa dosiahol parameter **MaxLHS**.

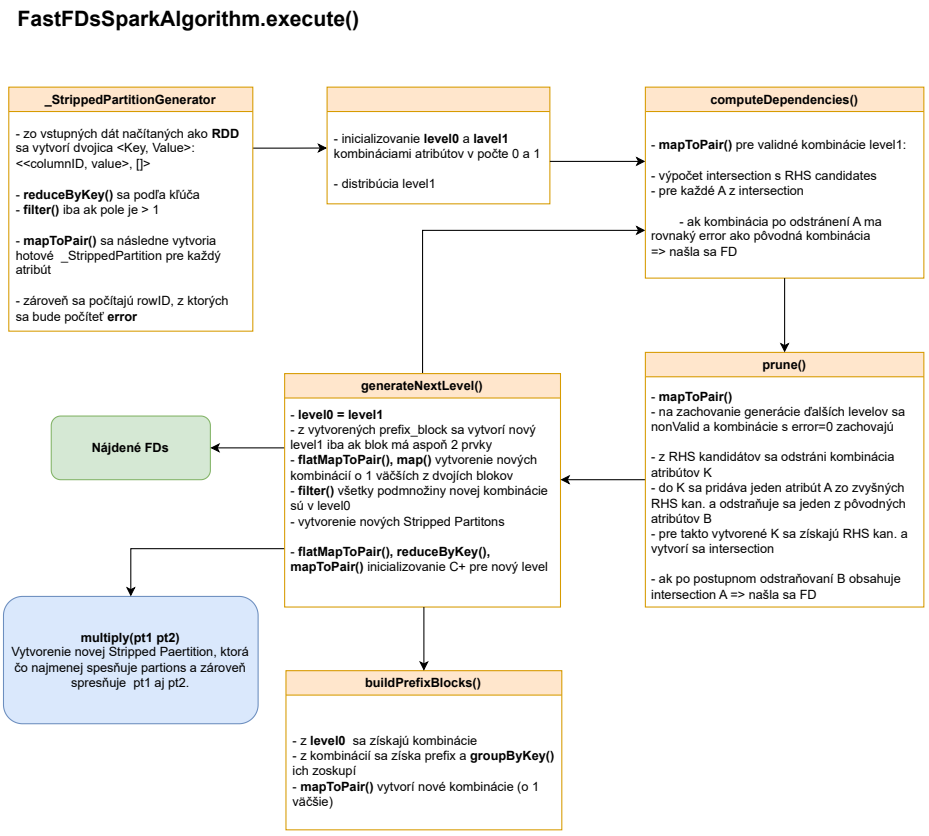

Figure 20: TaneSparkAlgorithm diagram

4.2 Experimenty

Hlavnou úlohou experimentov bolo namerať a porovnať časovú náročnosť pôvodnej a distribuovanej implementácie algoritmov. Na experimenty sa

použili vybrané datové sady, ktoré sú dostupné v GitHub repozitári[12].

Výsledky experimentov sú v tabuľke 1, kde je zelenou farbou naznačený lepší čas a v stĺpci **S/P** je zobrazený pomer Spark implementácie / Pôvodná implementácia.

Ako je možné vidieť, jediné podstatné zlepšenie je v algoritme **FDEP** na väčších datasetoch. Toto zlepšenie spôsobuje distribúcia výpočtu so zložitosťou n^2.

V ostatných prípadoch došlo k zhoršeniu časovej náročnosti o približne 2 až 35-krát, v extrémnom prípade **TANE** až 62-krát.

Takéto extrémne zhoršenie v prípade **TANE** môže byť spôsobené princípom výpočtu po vrstvách a taktiež distribúcia, či collecting dát na ich ďalší výpočet, čo je časovo náročné.

V prípade **HyFD** bol distribuovaný Validator, ktorý bol pôvodne paralelizovaný, z dôvodu náročného výpočtu, a tak bol vhodným kandidát. K výrazému zhoršeniu nedošlo, ale oäť sa ukazuje, že distribúcia a zber výsledkov je náročná na čas.

Algoritmy **Dep-Miner** a **FastFDs** majú podobnú štruktúru, ale rozdiely v čase výpočtu. Môže to byť spôsobené zjednodušením a spojením metód v **FastFDs** a napríklad nevytváranie Agree Sets. Taktiež má uloženú štruktúre relationships, ktorá sa použila pri vytváraní Difference Sets. Prechádzanie stromu v každom procese je jednoduchšie ako zber dát a výpočet LHS v prípade **Dep-Miner**.

[12] `https://github.com/DonRiccardo/DP-algoritmy`

Table 1: Porovnanie časovej zložitosti (v ms) algoritmov na vybraných datasetoch

Dataset	FDep			DepMiner		
	Pôvod. i.	Spark i.	S/P	Pôvod. im.	Spark i.	S/P
Abalone	1 591	2 456	1,54	1 584	54 617	34,48
IMDB-Movies	16 124	16 583	1,03	TL	TL	-
WBC-x1	138	538	3,89	537	5 327	9,92
WBC-x16	10 480	10 397	0,99	5 124	49 643	9,69
WBC-x64	222 230	139 371	0,63	18 616	267 203	14,35
WBC-NEWx79	670 830	463 876	0,69	3 382	TL	-

Dataset	FastFDs			HyFD		
	Pôvod. i.	Spark i.	S/P	Pôvod. i.	Spark i.	S/P
Abalone	1 671	4 308	2,58	190	949	5,01
IMDB-Movies	5 309	6 217	1,17	764	2 415	3,16
WBC-x1	490	1 603	3,27	107	602	5,62
WBC-x16	5 098	9 561	1,88	310	1 342	4,32
WBC-x64	18 614	30 388	1,63	633	2 750	4,34
WBC-NEWx79	3 634	14 128	3,89	1 479	10 714	7,24

Dataset	Tane		
	Pôvodná im.	Spark im.	S/P
Abalone	244	3 890	15,96
IMDB-Movies	893	TL	-
WBC-x1	266	5 327	20,02
WBC-x16	1 220	49 643	40,69
WBC-x64	4 289	267 203	62,30
WBC-NEWx79	1 929	TL	-

5 Záver

Výsledná aplikácia FD-finder s distribuovanými algoritmami je vhodným nástrojom na hľadanie funkčných závislostí. Podarilo sa splniť väčšinu požiadaviek kladených v špecifikácii s výnimkou niektorých. Napríklad sa výsledné FDs automaticky ukladajú do súboru, z dôvodu opakovaných spúšťaní algoritmov a taktiež výberu viacerých algoritmov naraz. Trochu je pozmenený princíp vytvárania datasetu a jobu z dôvodu lepšieho užívateľského komfortu.

Taktiež je aplikácia navrhnutá na jednoduché doplnenie algoritmu alebo formátu datasetu.

Miernym neúspechom, na ktorý bolo myslené v rizikách je, že niektoré z algoritmov nebolo možné efektívne distribuovať, a z toho dôvodu sú menej efektívne ako pôvodná implementácia. Na druhú stranu takto distribuované algoritmy sú stále vhodnejšie na rozsiahle datasety.

Samotné algoritmy, ktoré som distribuoval som prevzal z projektu Hasso-Plattner-Institut[13], ktorých úprava je popísaná v sekcii 4.1. V algotimoch zostala kostra výpočtu, myšlienky, postupy, zmenili sa iba niektoré metódy a proces výpočtu prostredníctvom Apache Spark.

V náväznosti na tento projekt budem ďalej pokračovať vo vývoji v diplomovej práci doplnením aplikácie o ďalšie funkcie a taktiež na základe získaných poznatkov sa pokúsiť doterajšie prístupy na hľadanie FDs prekopať alebo z nich vytvoriť nový prístup, ktorý bude dostatočne škálovať s potrebnou rýchlosťou výpočtu. Výsledky budú popísané v diplomovej práci a následne taktiež v článku.

Keďže aplikácia bola vyvíjaná iba ako jadro veľkého nástroja, môže byť do vyvíjaných nástrojov ľahko integrovateľná a môže poskytnúť pri profilovaní dát vhodné poznatky o ich kvalite, pretože pokrýva hlavné formáty z multimodelových dát (CSV, JSON).

[13] https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html

References

[1] Flach, P. A. ; Savnik, I. : Database dependency discovery: a machine learning approach. In: *AI Commun.* 12 (1999), Aug., Nr. 3, S. 139–160. – ISSN 0921–7126

[2] Huhtala, Y. ; Kärkkäinen, J. ; Porkka, P. ; Toivonen, H. : Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. In: *The Computer Journal* 42 (1999), Nr. 2, S. 100–111. http://dx.doi.org/10.1093/comjnl/42.2.100. – DOI 10.1093/comjnl/42.2.100

[3] Lopes, S. ; Petit, J.-M. ; Lakhal, L. : Efficient Discovery of Functional Dependencies and Armstrong Relations. In: Zaniolo, C. (Hrsg.) ; Lockemann, P. C. (Hrsg.) ; Scholl, M. H. (Hrsg.) ; Grust, T. (Hrsg.): *Lecture Notes in Computer Science* Bd. 1777. Springer (Advances in Database Technology - EDBT 2000 7th International Conference on Extending Database Technology Konstanz, Germany, March 27-31, 2000 Proceedings), 350-364

[4] Papenbrock, T. ; Ehrlich, J. ; Marten, J. ; Neubert, T. ; Rudolph, J.-P. ; Schönberg, M. ; Zwiener, J. ; Naumann, F. : Functional dependency discovery: an experimental evaluation of seven algorithms. In: *Proc. VLDB Endow.* 8 (2015), Jun., Nr. 10, 1082–1093. http://dx.doi.org/10.14778/2794367.2794377. – DOI 10.14778/2794367.2794377. – ISSN 2150–8097

[5] Papenbrock, T. ; Naumann, F. : A Hybrid Approach to Functional Dependency Discovery. Association for Computing Machinery (SIGMOD '16). – ISBN 9781450335317, 821–833

[6] Wyss, C. ; Giannella, C. ; Robertson, E. L.: FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances - Extended Abstract. Berlin, Heidelberg : Springer-Verlag, 2001 (DaWaK '01). – ISBN 3540425535, S. 101–110