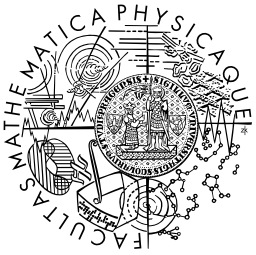


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Lukáš Kolek

Aproximativní datové profilování

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové a datové inženýrství

Praha 2021

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Děkuji RNDr. Michalovi Kopeckému, Ph.D. za vedení diplomové práce, věcné připomínky a čas, který mi věnoval. Cením si, že mi tímto bylo umožněno věnovat se tématu mého zájmu. Poděkování patří také mé rodině a blízkým za potřebnou podporu během studia.

Název práce: Aproximativní datové profilování

Autor: Lukáš Kolek

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D., Katedra softwarového inženýrství

Abstrakt: Na základě analýzy datových vstupů přináší datové profilování sumarizaci hodnot, která je vypočtena pomocí různých typů statistik. S růstem velikosti dat uložených v datových zdrojích se však nejedná o výpočetně snadnou operaci. Pro velká data není možné uložit všechny hodnoty v rychlé paměti RAM. Tudíž nelze použít jednoprůchodové a zároveň přesné algoritmy bez využití pomalejšího úložiště počítače. Diplomová práce má za cíl implementovat, porovnat a následně vybrat vhodné aproximativní algoritmy pro profilování objemných dat. Díky aplikaci aproximativního přístupu lze následně omezit paměť potřebnou pro výpočet natolik, že celou jednoprůchodovou analýzu dat je možné spočítat pouze v paměti RAM. Tím se proces profilování dat značně urychlí. Nástroj pak dokáže produkovat výsledky frekvenční analýzy, výpočtů kardinality, kvantilů nebo histogramů a dalších jednosloupcových statistik v krátkém čase s chybou řádově v desetinách procent.

Klíčová slova: data profilování aproximativní algoritmus

Title: Aproximative data profiling

Author: Lukáš Kolek

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D., Department of Software Engineering

Abstract: Data profiling is the process of analyzing data and producing an output with statistical summaries. The size of data rapidly increases and it is more difficult to process all data in a reasonable time. All data can not be stored in RAM memory, so it is not possible to run exact single-pass algorithms without using slower computer storage. The diploma thesis focuses on the implementation, comparison, and selection of suitable algorithms for data profiling of large input data. Usage of approximate algorithms brings a possibility to limit memory for computation, do the whole process in RAM memory and the duration of data profiling should be reduced. The tool can compute frequency analysis, cardinality, quantiles, histograms, and other single-column statistics in a short time with a relative error lower than one percent.

Keywords: data profiling aproximative algorithm

Obsah

Úvod

S růstem velikosti dat a počtu datových zdrojů je čím dál tím obtížnější udržovat informace o jejich obsahu. Díky analýze dat je možné vytvořit sumarizaci obsahu tabulek nebo souborů pomocí různých statistik. Výsledky poté mohou být nejen prezentovány uživateli, ale také zpracovány dalšími procesy. Analýza velkého objemu dat však může být náročná na velikost výpočetní paměti počítače. Jednou z variant, jak se s tímto vypořádat, je použití aproximativních algoritmů, které pracují s velmi malou pamětí oproti velikosti vstupu. Dále lze snížit čas výpočtu aplikací vzorkování vstupních dat. Daní za oba přístupy je však snížená přesnost výsledků.

Cíl práce

Aproximativní algoritmy v datovém profilování nejsou tolik využívány. Z tohoto důvodu má práce za cíl otestovat použitelnost existujících algoritmů za účelem redukce potřebné paměti pro běh celé analýzy. Tím dochází k propojení teoretické znalosti s praxí. Jakmile budou jednotlivé analýzy implementovány, je potřeba algoritmy mezi sebou i s přesnými variantami porovnat a zhodnotit jejich použitelnost v datovém profilování. Dalším cílem je vyzkoušet také aplikaci vzorkování vstupních dat a tuto variantu poté opět srovnat s ostatními přístupy. Pro zmíněné testování je nutné vytvořit aplikaci, která bude umět zpracovávat data z databází a z CSV souborů a umožňovat snadné rozšiřování o statistky a jejich parametrizování za účelem snadného testování. Díky aproximativním algoritmům budou sníženy požadavky na parametry profilujícího serveru a také bude odstraněna, nebo alespoň minimalizována, potřeba využití disku jako paměti při zpracování. Ve výsledku by aplikace měla poskytnout základní náhled na velká vstupní data v krátkém čase s velmi omezenou pamětí pro výpočet.

Struktura práce

V první kapitole práce je čtenář seznámen s datovým profilováním. Kapitola také představuje tento proces v širším kontextu datového managementu. Poté je popsán přehled základního rozdělení datového profilování společně s různými přístupy k řešení tohoto problému. V následující kapitole je sestaven seznam potřebných vlastností pro implementaci aplikace tak, aby bylo možné naplnit cíle práce. Tyto vlastnosti jsou také otestovány v několika již existujících nástrojích.

Následují dvě kapitoly, které se zabývají implementací aplikace. Třetí kapitola popisuje architekturu a jednotlivé komponenty aplikace společně s představením její konfigurace. Ve čtvrté kapitole jsou popsány vybrané implementační zajímavosti, které mají čtenáři přiblížit vnitřní strukturu aplikace. Po přečtení návodu v poslední části této kapitoly získá čtenář představu, jak rozšířit aplikaci o další analýzy nebo vstupní datové zdroje. Podrobnější dokumentace se nachází přímo v kódu na úrovni jednotlivých rozhraní.

Hlavní část práce je tvořena popisem použitých algoritmů s důrazem na jejich analýzu a porovnání. Slovní popis algoritmů je pro snazší pochopení doplněn také

o pseudokód. Na konci každé kapitoly jsou popsaná vybraná měření algoritmů z hlediska přesnosti a náročnosti na potřebnou paměť. Kromě frekvenční analýzy, výpočtu kardinality, histogramu a kvantilů práce obsahuje další analýzy, které již nemají aproximativní ekvivalent, jelikož nejsou tolik náročné na paměť. Avšak bez těchto algoritmů se nástroj na datové profilování v praxi neobejde.

Kapitola s výsledky práce vychází z dílčích pozorování a obsahuje měření chodu celé aplikace na reálných datech.

Závěrečná kapitola hodnotí naplnění stanovených cílů a prezentuje možnosti dalšího zpracování tématu.

1. Datový management, datové profilování

Mezi jednotlivé disciplíny datového managementu patří především efektivní správa a analýza dat, údržba datové kvality, ochrana informací, dostupnost a znovupoužitelnost uložených dat. Cílem datového managementu je zajistit, aby lidé nebo společnosti optimalizovali práci s daty, a tím zvýšit jejich užitek pro lepší rozhodování, snížit rizika týkající se nepřesných informací a usnadnit návazné operace s daty. S růstem objemu dat také nabývá důležitosti efektivní práce s informacemi.

Efektivní práce s daty začíná na úrovni výběru a správy datového úložiště, jako jsou SQL nebo NoSQL databáze, datové sklady[1] a datová jezera[2]. Na tuto úroveň navazuje analýza[3], integrace[4], čištění[5] nebo případně migrace dat[6]. Nejdále od samotných dat je správa metadat[7] nebo analýza toků dat[8]. Kromě zmíněných disciplín existují ještě další části datového managementu. Tím je podtržena jeho neodmyslitelná přítomnost v dnešním světě plném informací.

Budoucností zmíněných procesů a nástrojů je automatizace práce pomocí zapojení různých oblastí umělé inteligence. Rostoucí objem dat již často neumožňuje jejich lokální zpracování a rychle dostupné výsledky. Často se během celého procesu mění datové struktury, dat přibývá na objemu a stále přicházejí nové možnosti využití již dříve získaných informací. Spolu s těmito nároky roste důraz na bezpečnost a přibývají legislativní nařízení, jako jsou například GDPR[9] nebo CCPA[10].

1.1 Datové profilování

Hlavním cílem datového profilování je přinášet na základě analýzy sumarizaci hodnot v datovém zdroji pomocí různých typů statistik.

Proces produkuje statistiky, ze kterých jsou patrné různé vlastnosti dat, jako jsou například konzistence, úplnost, přesnost nebo duplikace dat. Statistiky jsou vypočítávány na úrovni sloupců, tabulek nebo celých datových zdrojů. Výsledky jsou předávány datovým stevardům[11] pomocí grafické nebo tabulkové vizualizace. Mezi jednotlivé analýzy patří především číselné statistiky, počty neúplných hodnot, frekvenční analýza, unikátnost, kardinalita apod.

Hranice mezi datovým profilováním a data miningem není přesně definována (Abedjan a kol., 2015), ale za hlavní rozdíl můžeme považovat, že datové profilování má za cíl získat metadata, pomoci pochopit data a zvýšit možnost použitelnosti dat. Naopak v případě data miningu je cílem pokusit se získat skryté informace z dat, a tím ještě zvýšit jejich přínos.

Datové profilování v praxi znamená, že uživatel nejprve vybere, jaký datový zdroj chce profilovat. Aplikace začne během čtení tento vstup analyzovat a na konci běhu vydá souhrn v podobě spočítaných výsledků. Většina aplikací poté

[1]Data warehouse [2]Data lake [3]Data analysis, Data mining, Data profiling [4]Data integration
[5]Data cleansing [6]Data migrations [7]Metadata management [8]Data lineage
[9]General Data Protection Regulation [10]California Consumer Privacy Act [11]Data steward

výstup dokáže vizualizovat. Uživatel je schopen se k výsledkům vracet a také ve statistikách vyhledávat.

V oblasti datového profilování stále existuje několik výzev. Mezi hlavní patří analýza velkých dat, počítání statistik z datových zdrojů jiných než relační databáze a také zapojení algoritmů umělé inteligence do celého procesu. Práce se zabývá první z výše jmenovaných výzev.

1.2 Využití datového profilování

Analýza dat a sbírání statistických výsledků nemusí sloužit jen pro lepší pochopení informací uložených v datovém zdroji. Lze nalézt další oblasti, které konzumují výsledky datového profilování, jako jsou datový katalog, klasifikace dat, datová kvalita nebo integrace dat. V neposlední řadě existuje několik využití výsledků pomocí umělé inteligence, která pomáhá k automatizaci a zkvalitnění procesů datového managementu (Ataccama, 2020). Mimo zmiňované použití v datovém managementu je dnes datové profilování zastoupeno také ve většině databází pro optimalizaci dotazů na základě odhadů náročnosti dané akce.

1.2.1 Datový katalog a klasifikace dat

V případě správy mnoha datových zdrojů je častou potřebou využití datového katalogu. Pomocí této platformy se lze připojit k mnoha datovým zdrojům za účelem sesbírání maximálního objemu užitečných informací na úrovni metadat.

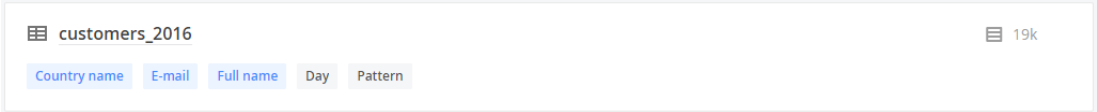
Nejprve jsou získány informace ohledně struktury datových zdrojů na úrovni tabulek, sloupců a dalších databázových elementů. Dalším krokem je analýza samotných dat. Proto se spustí datové profilování a výsledky se uloží společně s metadaty do zmíněného katalogu.

Aby se daly uložené informace snadno vyhledávat nebo třídit, provádí se klasifikace dat. Ke každé tabulce nebo sloupci v katalogu lze přiřadit značku[12], která vystihuje jeho daný účel nebo obsah. Seznam všech značek se ukládá do slovníku[13], ve kterém lze definovat pravidla pro jejich přiřazování. Pravidla, která jsou vhodná pro automatické přiřazení značek, mohou být použita pro detekci během profilování. Přiřazené značky jsou součástí výsledků zmíněné analýzy.

Bez katalogizace nebo klasifikace dat je velmi pravděpodobné, že uložené informace nejsou snadno dostupné nebo vyhledatelné, a jakékoliv získávání nových informací na základě dat je mnohem obtížnější. Kromě lepší organizace metadat lze na základě klasifikace dat detekovat tabulky nebo konkrétní sloupce obsahující osobní data a následně automaticky povolit přístup k těmto informacím pouze určité skupině analytiků.

Obrázek 1.1 znázorňuje, jak v praxi vypadá klasifikace dat. Na základě pravidel byly k tabulce a jejím sloupcům přiřazeny značky: Country name, E-mail, Full name, Day a Pattern. Uživatel má poté možnost například vyhledat všechny tabulky nebo sloupce se značkou E-mail. Ukázka pochází z nástroje Ataccama ONE, který je popsán v sekci 2.2.2.

[12] V praxi se často používá pojem „term“ nebo „tag“. [13] Glossary



Obrázek 1.1: Ukázka klasifikace dat z nástroje Ataccama ONE

1.2.2 Datová kvalita a transformace dat

Za pomoci výsledků datového profilování lze určit chyby v datech, a tím připravit vstupy pro zahájení procesu zvyšujícího datovou kvalitu. Ta je poté zvýšena aplikací transformací nebo manuální opravou dat. Na konci takového procesu je typicky monitoring kvality, který je možné opět vykonávat pomocí datového profilování v pravidelných intervalech.

Příkladem může být detekce několika různých zápisů pohlaví (M vs. Male vs. male) ve sloupci tabulky o uživatelích. Následné zpracování takových dat přináší problémy, které lze odstranit jednoduchou transformací výchozích dat. Případně lze detekovat chybějící data nebo překlepy v hodnotách.

1.2.3 Umělá inteligence

Velkou výzvou pro všechny platformy poskytující nástroje aplikovatelné v datovém managementu je zapojení umělé inteligence, a tím snížení pracnosti se správou dat. Na základě výsledků profilování dat jsou algoritmy založené na strojovém učení schopné vytvářet doporučení pro následné manuální akce, jako je detekce podobností, a tím redukce duplicit, nebo lze získat návrhy vhodných datových transformací. Také lze odhalovat problémy v datech dříve, než se objeví v následném zpracování, a tím zamezit špatným rozhodnutím na základě chyb ve vstupních datech. Jednou z cest, jak detekovat chyby, je pravidelné spouštění datového profilování a následné porovnání výsledků z předchozích běhů. Díky tomuto postupu lze v datech detekovat anomálie a zamezit tak následnému použití poškozených dat.

Společnost, která na konci každého týdne sbírá data o objednávkách ze svých regionálních poboček, může do procesu zahrnout datové profilování se zmíněnou detekcí anomálií. Jakmile pobočka předá svá data, automatický proces spustí datové profilování a na základě dat z předchozích týdnů porovná výsledky. Pokud je nalezena anomálie, pak proces sbírání dat může být zastaven dříve, než se pravděpodobná chyba v datech dostane i do dalších systémů.

1.3 Rozdělení datového profilování

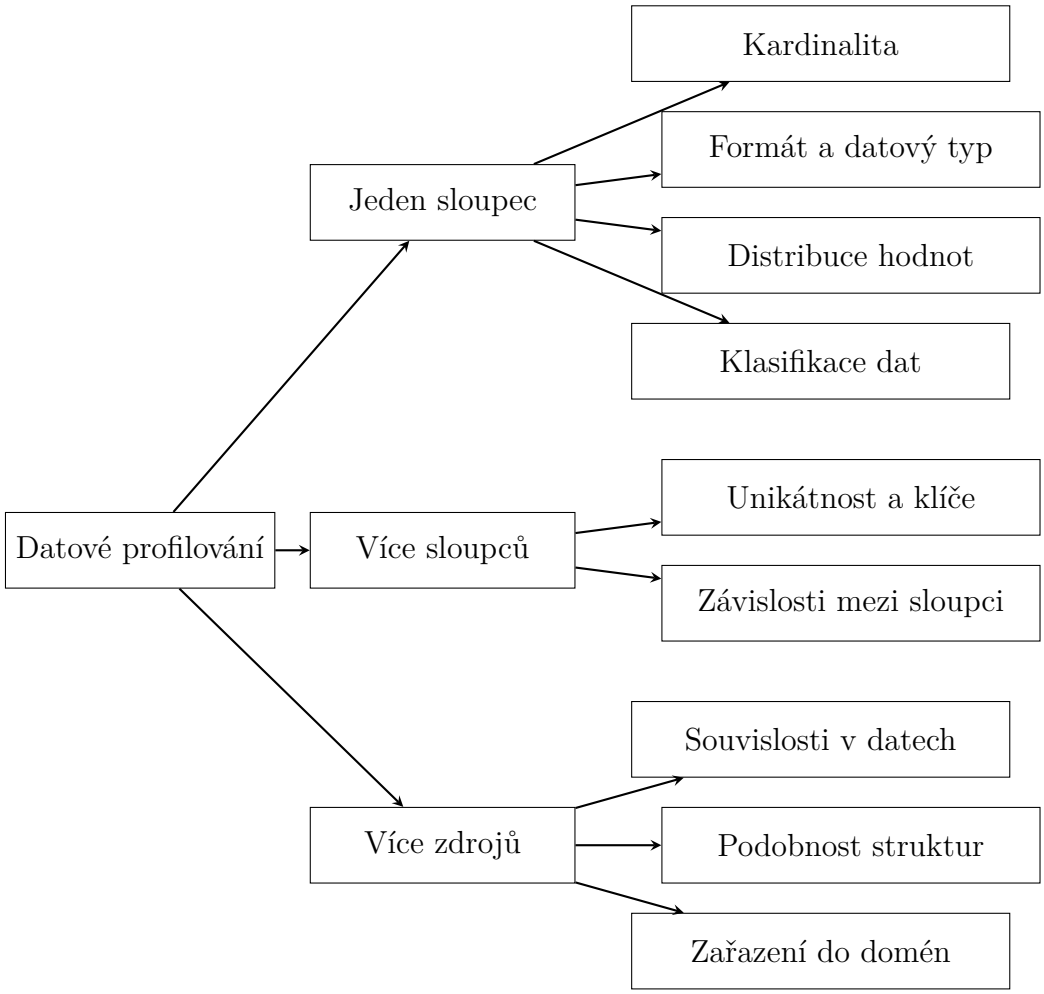
Datové profilování lze rozdělit na základní tři kategorie podle toho, s jakým vstupem analýzy pracují: jednosloupcové, vícesloupcové a analýzy s více vstupními soubory nebo tabulkami (Naumann, 2014). Rozdělení je znázorněno na obrázku 1.2.

První kategorií jsou analýzy jednoho sloupce. Mezi cíle těchto analýz patří výpočet základních statistik pro numerické hodnoty a řetězce, určení počtu unikátních hodnot, detekování, zda se hodnoty vyskytují v určitém formátu, zazna-

menání, jaké je rozložení hodnot pomocí frekvenční analýzy, histogramu nebo kvantilů, a provedení klasifikace dat.

Do další skupiny patří analýzy pracující stále nad jediným datovým souborem, ale libovolným počtem sloupců. Pomocí těchto algoritmů lze detekovat klíče na úrovni tabulek a funkční nebo jiné závislosti mezi sloupci.

Nakonec lze konstruovat algoritmy, které detekují duplicity, podobnosti nebo souvislosti mezi více datovými sadami na vstupu.



Obrázek 1.2: Rozdělení datového profilování dle typu analýz (Naumann, 2014)

Práce se zaměřuje na jednosloupcové analýzy, které jsou implementovány pomocí aproximativních algoritmů. Tento postup by měl dovolit omezit maximální paměť, která je potřebná k profilování velkých datových vstupů. Vícezdrojové algoritmy by samy o sobě přesahovaly rozsah diplomové práce. Z vícesloupcových analýz lze implementovat jednoduchou analýzu korelací mezi sloupci. Další analýzy však potřebují větší paměť. Nelze je proto zahrnout do nástroje pracujícího s velmi omezenou pamětí pro výpočet.

1.4 Přístupy k datovému profilování

Kromě základního rozdělení datového profilování podle použitých algoritmů uvedeného v sekci 1.3 existuje ještě několik přístupů, pomocí kterých lze datové profilování také rozdělit.

1.4.1 Profilování v databázi

Některé statistiky lze spočítat přímo v databázi. Tím by výsledná rychlost výpočtu měla být nejrychlejší, jelikož se ušetří čas potřebný pro přenos dat na jiný server. Například kód 1.1 demonstruje výpočet frekvenční analýzy pomocí jazyka SQL s využitím konstrukce `GROUP BY`. Kvůli odlišnostem v databázových jazycích je však obtížné naprogramovat analýzy obecně pro libovolný datový zdroj. Další nevýhodou je výpočetní náročnost pro databázový systém, který často běží v produkčním prostředí. Tímto způsobem by mohlo dojít k citelnému zpomalení daného serveru. Pokud lze předpokládat, že sekvenční čtení bude pro databázový systém mnohem snadnější operace než samotná analýza, pak lze výpočetní náročnost přenést na jiný server, a tím snížit náročnost výpočtu na databázovém serveru.

Dalším problémem při profilování dat v databázi může být náročnost zapsání některých pokročilých analýz. Například klasifikace dat ze sekce 5.5 často obsahuje desítky pravidel, která by musela být zapsána v jazyce SQL, a výsledná analýza by nebyla jednoprůchodová nebo by se jednalo o velmi rozsáhlý dotaz do databáze.

Z těchto důvodů se v praxi vyskytují zejména nástroje, které data z databázového serveru pouze čtou a statistiky počítají na jiném výpočetním serveru.

```
SELECT column, count(column) as freq
FROM table
GROUP BY column
ORDER BY freq DESC;
```

Kód 1.1: SQL dotaz pro získání frekvencí hodnot daného sloupce

1.4.2 Profilování na výpočetním serveru

Profilování na jiném serveru než přímo na databázovém přináší větší volnost v implementaci analýz. Hlavní nevýhodou ale je, že časová náročnost výpočtu je závislá na rychlosti čtení dat z databáze a následném přenosu dat mezi servery. Některé nástroje podporují analýzu na lokálním počítači, avšak v dnešní době kvůli růstu objemu dat většina výpočtů přechází na výpočetní servery nebo do cloudového prostředí, kde lze zajistit větší škálování. Obecně jsou tato prostředí pro výpočty vhodnější než uživatelský počítač.

Výhoda počítání analýz mimo databázový systém přináší také nezávislost algoritmů na typu datového zdroje. Stejný kód algoritmů lze použít pro analýzu dat z různých databázových systémů nebo CSV souborů pouhou změnou modulu pro čtení dat.

1.4.3 Profilování vzorku dat

S růstem velikosti dat přestalo být možné načíst všechny hodnoty do paměti RAM výpočetního serveru. Z tohoto důvodu je komplikovanější během jediného průchodu daty počítat statistiky, které jsou založené na frekvenční analýze, pro sloupce obsahující mnoho unikátních hodnot. Některé aplikace v případě nedostatku paměti řeší toto omezení pomocí využití disku. Tím se výpočet může výrazně zpomalit.

Pro ušetření paměti a zároveň zrychlení analýzy lze provádět výpočet pouze nad vzorkem dat. Většina databázových systémů podporuje vzorkování při čtení dat z databáze pomocí příkazu `SELECT`. Tímto přístupem je možné omezit většinou relativní nebo také absolutní počet čtených záznamů. V ideálním případě je při čtení pouze 10 % záznamů získáno více než 10x větší zrychlení. Díky výpočtu, který je proveden pouze v paměti RAM, lze očekávat mnohem větší hodnoty zrychlení. Aplikací rychlého výpočtu nad vzorkem se dá například zjistit přibližné informace o větším počtu neznámých tabulek a na základě takto získaných výsledků poté vybrat zajímavou část dat, která je znovu profilována již nad plným rozsahem.

1.4.4 Aproximativní profilování dat

Další variantou, jak počítat statistiky v paměti RAM nad velkým objemem dat, je využití aproximativních algoritmů, které produkují přibližné výsledky blížící se opravdovým hodnotám. Společnou vlastností těchto algoritmů je, že potřebují mnohem menší paměť pro výpočet. Často lze tuto paměť předem omezit. Společnou a zároveň logickou vlastností aproximativních algoritmů je, že čím je větší paměťové omezení, tím klesá přesnost algoritmu.

Aproximativní přístup lze použít pro statistiky, kdy nevadí, že vrácené výsledky budou s chybou v řádu procent. Nebo, obdobně jako v předchozím případě, lze profilovat všechny tabulky v databázi pomocí aproximativních analýz a poté vybrat zajímavou podmnožinu tabulek, která bude profilovaná přesnými algoritmy.

1.4.5 Inkrementální profilování

Aplikace profilující data jsou často využívané pro plánovaný opakovaný běh ve stanovených intervalech. Pokud data splňují, že jsou pouze inkrementální, není často potřeba profilovat celý vstup, ale pouze nový přírůstek dat. Pokud uživatele zajímá výsledek výpočtů pouze nad tímto inkrementem, je očekávatelná výrazná redukce časové a paměťové náročnosti. V případě, že je potřeba sloučit nové výsledky s již předešlými, mohou být přesné analýzy během výpočtu opět paměťově náročné. Příkladem je již zmíněná frekvenční analýza, u které je potřeba mezi běhy uložit všechny unikátní hodnoty s výskyty, aby při dalším běhu algoritmus mohl pokračovat. Z tohoto důvodu je vhodné použití aproximativního algoritmu s limitovanou paměťovou náročností i nad malými inkrementy při celkově velkých datech. Zmíněný aproximativní algoritmus může kromě vrácení výsledků také uložit neboli serializovat interní paměť pro následný výpočet. Později lze v tomto výpočtu opět pokračovat.

2. Analýza

Při definování požadovaných vlastností a návrhu aplikace byl kromě reálného použití kladen důraz také na práci s omezenou pamětí, na testovatelnost algoritmů, možnosti vzorkování a budoucí snadné rozšíření o další analýzy. Tyto požadavky byly tvořeny na základě schopností již existujících nástrojů. Často používané funkcionality byly doplněny především o aproximativní přístup.

2.1 Základní vlastnosti aplikace

Analýzy

- Algoritmy by měly být jednoprůchodové.
- Existující analýzy by měly podporovat různé algoritmy pro jejich výpočet.
- Je žádoucí, aby bylo možné měnit implementaci algoritmů analýzy mezi běhy aplikace.
- Aproximativní verze algoritmu by měla mít také přesný ekvivalent.
- Pro testovací účely by mělo být možné pouštět dvě stejné analýzy s různými algoritmy.
- Měl by existovat algoritmus dané analýzy, který pracuje pouze s předem omezenou pamětí.
- Před samotným spuštěním by mělo být možné konfigurovat, jaké analýzy a algoritmy jsou aktivní. Také je žádoucí, aby algoritmy měly konfigurovatelné parametry.

Čtení dat

- Je nutné, aby pomocí aplikace bylo možné číst data z SQL databáze a CSV souborů.
- Při čtení dat ze souborů by aplikace měla umět data vzorkovat[1].
- Měla by existovat možnost, jak rozšířit aplikaci o další datové zdroje. Například o čtení souborů z Amazon S3[2] nebo o podporu souborů ve formátu xlsx.

Výstup

- Základním formátem výstupu by měl být soubor ve formátu JSON umožňující následné automatické zpracování.
- Z důvodu chybějícího uživatelského rozhraní by aplikace měla obsahovat výstup ve formátu HTML. Tento formát je pro uživatele čitelnější než formát JSON.

[1] Pro SQL databáze lze vzorkování vynutit přímo v datovém zdroji.
[2] https://aws.amazon.com/s3/

Aplikace

- Aplikaci by mělo být možné snadno rozšiřovat o další analýzy.
- Aplikaci by mělo být možné spouštět v jednovláknovém i vícevláknovém režimu pro následné otestování obou přístupů.
- Mělo by být umožněno omezit počet souběžně načtených záznamů v paměti.

2.2 Srovnání již existujících nástrojů

Během analýzy již existujících nástrojů pro datové profilování bylo vybráno několik zástupců, kteří jsou zdarma dostupní a provádějí datové profilování databází nebo souborů. V plné verzi většina vybraných nástrojů navíc obsahuje další rozšíření. Častým příkladem placených modulů je datový katalog, případně rozšíření o moduly zaměřené na datovou kvalitu nebo transformace.

Porovnána byla tato řešení: Ataccama DQ Analyzer[3], Ataccama ONE[4], DataCleaner[5], IBM InfoSphere Information Analyzer[6], Talend Open Studio for Data Quality[7], Informatica Data Quality[8], Open Source Data Quality and Profiling[9]. Některé z těchto produktů jsou společnostmi poskytovány k vyzkoušení, nebo se dokonce jedná o aplikace zdarma. Tito volně dostupní zástupci jsou dále detailněji rozebráni a byl proveden test na několika souborech.

Prvním cílem bylo porovnat, jaké analýzy může uživatel aplikovat. Druhým cílem bylo zjistit, jak se daný nástroj vypořádá na průměrném stroji s objemným textovým souborem se 16 sloupci a s 15 milióny řádků, který má velikost přibližně 2 GB. Testování proběhlo na notebooku s procesorem Intel Core i5-8250U, 16GB RAM a SSD diskem. Nejedná se o výkonný stroj, ale stále by tyto parametry měly stačit k profilování souborů o velikosti 1-2 GB v řádech několika minut. Přesně k tomuto účelu by měly sloužit všechny zmíněné nástroje zdarma.

Komerční nástroje nejsou určeny pro domácí použití. Cílovou skupinu tvoří společnosti s mnoha datovými zdroji a s nutnou potřebou datového katalogu nebo monitoringu datové kvality, v některých případech také datové integrace. Jelikož nástroje nejsou volně dostupné, lze pouze z volně dostupných dokumentací vyčíst použité přístupy k datovému profilování. Častým předpokladem u těchto nástrojů je, že výpočet neběží již na lokálním počítači, ale na vzdáleném serveru s dostatečným výkonem.

2.2.1 Ataccama DQ Analyzer

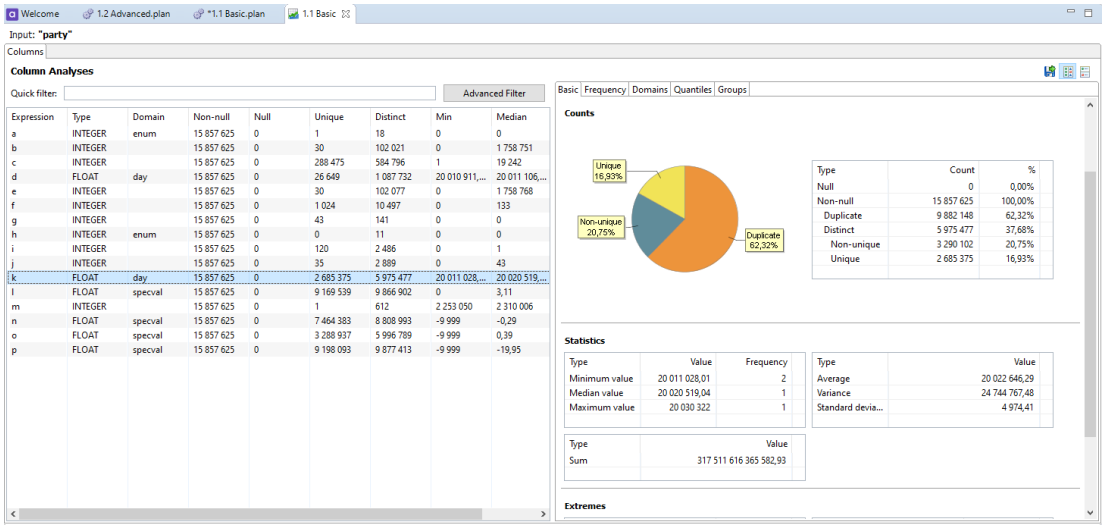
Aplikace Data Quality Analyzer od společnosti Ataccama[10] představuje zdarma dostupnou verzi nástroje pro účely práce s daty v oblasti datové kvality. Architektura celé aplikace je založena na platformě Eclipse[11]. Uživatel v této

[3]https://www.ataccama.com/download/dq-analyzer [4]https://one.ataccama.com
[5]https://datacleaner.github.io
[6]https://www.ibm.com/products/infosphere-information-analyzer
[7]https://www.talend.com/products/data-quality/data-quality-open-studio/
[8]https://www.informatica.com/products/data-quality/informatica-data-quality.html
[9]https://sourceforge.net/projects/dataquality/ [10]https://www.ataccama.com
[11]https://www.eclipse.org

aplikaci vytváří spustitelné výpočetní plány, které se skládají z kroků výpočtu a přechodů mezi danými kroky. Plán pro profilování testovacích dat začíná krokem `Text File Reader`, který čte data z CSV souborů, a následně je napojen na krok `Data Profiling`. Mezi zmíněné dva kroky je dále možné přidat procesy transformující data.

Výstupem kroku `Data Profiling`, který analyzuje data, je soubor `.profile`, který lze otevřít pouze v této aplikaci. Po jeho otevření je uživateli nabídnuta vizualizace výsledků. Pro účely datového profilování je možné číst z libovolné databáze pomocí JDBC driveru a CSV nebo Excel souborů. Plná verze aplikace nabízí podporu dalších datových zdrojů a velké množství různých kroků rozšiřujících plány. Výsledky datového profilování obsahují mnoho jednosloupcových statistik, klasifikaci dat na základě pravidel nebo analýzu závislostí mezi sloupci.

Pro profilování zmíněného rozsáhlého datového souboru nepotřebovala aplikace během analýzy více než 500 MB paměti, jelikož byly využity dočasné soubory na disku. Při profilování dat proběhly dvě fáze. Během první fáze se přečetly všechny řádky ze vstupního souboru a rostl objem dočasného úložiště. Ve druhé fázi již neprobíhalo čtení vstupních dat, ale probíhala pouze práce nazvaná Computing profile s dočasnými soubory. Výsledkem byl zmiňovaný `.profile` soubor. I přes využívání disku uživatel v tomto případě získal výsledky již po přibližně 7 minutách. Tento přístup nakonec ukázal, že znatelně překonává ostatní zdarma dostupné nástroje. Zajímavé by také bylo porovnání s komerčními nástroji, zda se i v tomto případě jedná o nejrychlejší řešení.



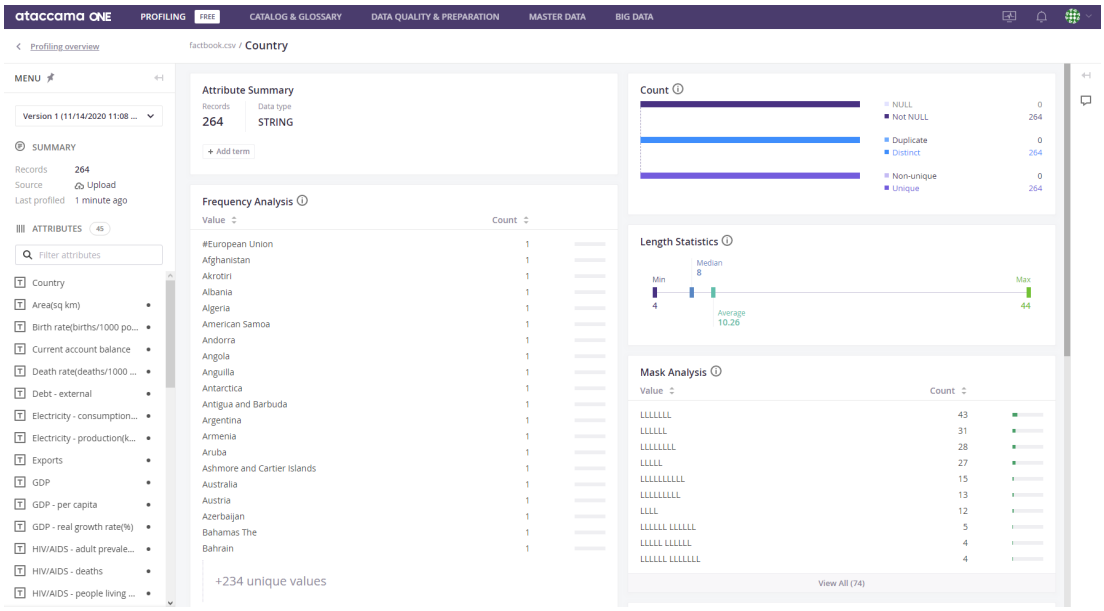
Obrázek 2.1: Ataccama Data Quality Analyzer - výsledky datového profilování

2.2.2 Ataccama ONE

Oproti první a zároveň starší aplikaci od společnosti Ataccama je Ataccama ONE webovou aplikací, která je opět v omezeném rozsahu zdarma dostupná pro registrované uživatele za účelem poskytnutí základní funkcionality celé platformy. Profilování funguje velmi podobně jako v případě DQ Analyzeru. Výhodou je, že uživatel již nemusí konfigurovat, jaké sloupce daný soubor obsahuje, ale tyto informace jsou extrahovány přímo z dat.

Kvůli potřebnému uploadu souboru k profilování proběhlo pouze testování možností daného nástroje s menšími datovými soubory. Profilování neběží na lokálním počítači, ale data jsou zpracovávaná na cizím vzdáleném serveru. Proto nástroj není vhodný pro analýzu citlivých dat. Aplikace kromě jednosloupcových statistik zobrazuje výsledky automatické i manuální klasifikace dat na základě značek s předem definovanými pravidly.

Dalším použitím aplikace v plné verzi je připojení se k technologiím Hadoop[12] nebo Spark[13] a také analýza datové kvality spolu s možnou transformací dat.



Obrázek 2.2: Ataccama ONE - výsledky datového profilování

2.2.3 DataCleaner community edition

Prvním zástupcem z řad Open Source nástrojů je DataCleaner ve verzi community edition. Na první pohled se jedná o zdařilou aplikaci, která nemá problém se připojit k velkému počtu datových zdrojů. Uživatel, stejně jako v případě Ataccama DQ Analyzer, tvoří plány a mezi vstupními a koncovými kroky lze dělat transformace a konverze dat.

Aplikace je ve srovnání s konkurencí lehce pozadu co do počtu různých statistik, které lze počítat během datového profilování. Naopak oproti ostatním testovaným nástrojům přináší aplikaci umělé inteligence, využití referenčních dat a komplexní transformace. Konfigurace profilování jednoho souboru nebo tabulky je v porovnání nejnáročnější, ale i tak je ovládání intuitivní.

Během testování s menšími soubory byly výsledky vypočítány až nečekaně rychle. Bohužel to se nedá říct o zpracování velkého testovacího souboru o velikosti 2 GB. Při běhu aplikace šlo vypozorovat, že práce probíhala pouze v paměti RAM. Jakmile výpočet dosáhl dvou třetin vstupního souboru, konzumace paměti se přehoupla přes 5 GB a přírůstek zpracovaných řádků byl pouze v řádech stovek za sekundu. Aplikace byla vypnuta po několika desítkách minut a výsledek nebyl získán.

[12]https://hadoop.apache.org [13]https://spark.apache.org

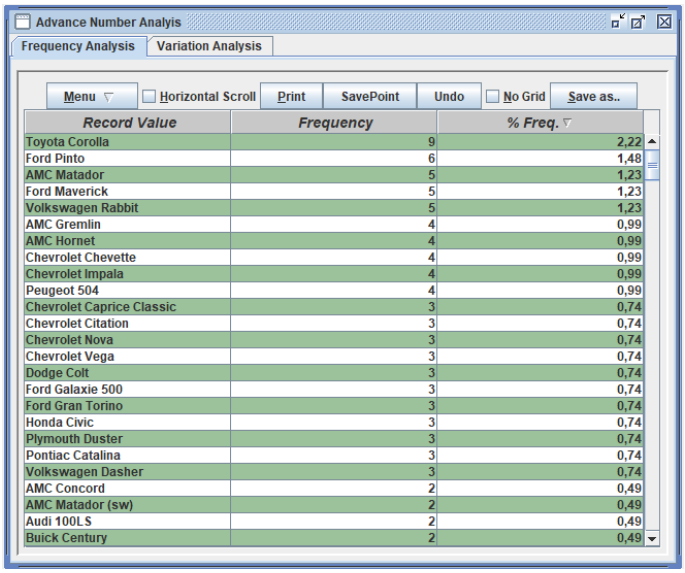
Aplikace je velmi vhodná pro práci v řádech jednotek miliónů řádků vstupních dat. Pro větší soubory není již aplikace použitelná, a i přes možnou záměnu hardwaru za výkonnější stroj je patrné, že implementace není vhodná pro objemná data.

Obrázek 2.3: DataCleaner - výsledky datového profilování

2.2.4 Open Source Data Quality and Profiling

Dalším Open Source zástupcem je aplikace Open Source Data Quality and Profiling známá také pod názvem Aggregate Profiler. I přes méně uživatelsky přívětivé ovládací rozhraní se jedná o užitečnou aplikaci zejména díky velkému počtu podporovaných datových zdrojů, možnostem transformace dat a také pokročilým analýzám, jako je vyhledávání podobností v datech nebo metody zpracování přirozeného jazyka.

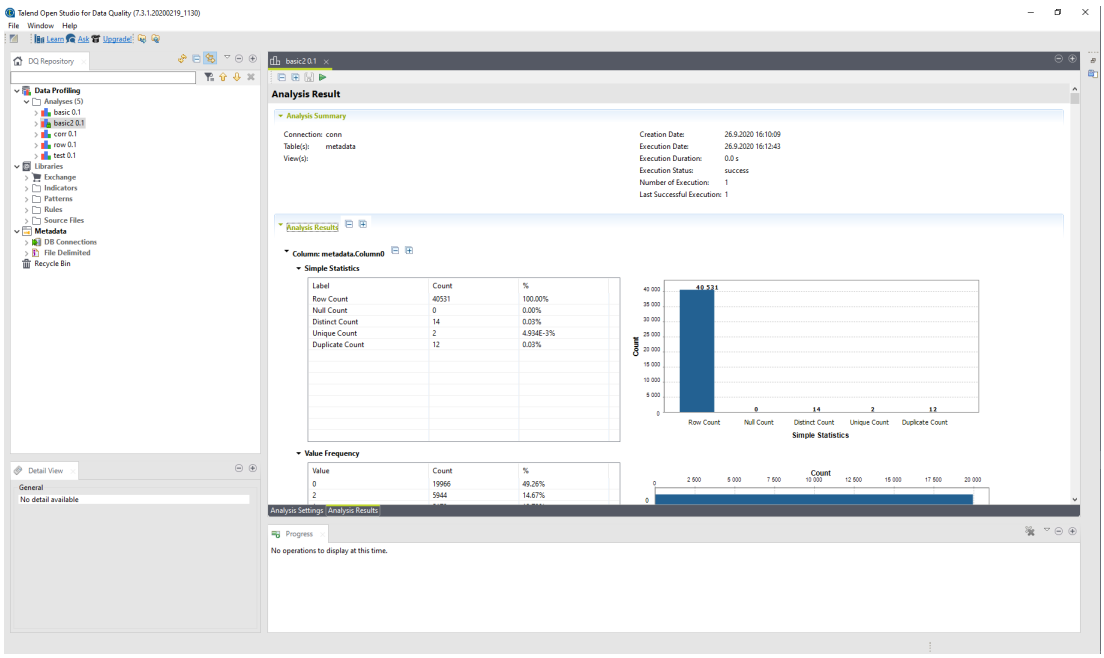
Testování nástroje pro menší soubory proběhlo i přes na první pohled horší uživatelské rozhraní snadno. Problém však nastal při pokusu o analýzu velkého souboru. Aplikace se pravděpodobně pokusila přečíst celý obsah, ale když paměť RAM překonala hranici 4 GB, program již přestal odpovídat. Nebyla zde možnost přečíst pouze prvních k řádků nebo jakýkoliv jiný vzorek dat tak, aby aplikace mohla běžet i pro objemná data.



Obrázek 2.4: Open Source Data Quality and Profiling - výsledky datového profilování

2.2.5 Talend Open Studio for Data Quality

Společnost Talend[14] poskytuje vedle svého Data Integration řešení pro velké společnosti také několik Open Source nástrojů. Jedním z nich je Open Studio for Data Quality se službou pro profilování dat. Tato aplikace je také založena na již zmiňované platformě Eclipse.



Obrázek 2.5: Talend Open Studio for Data Quality - výsledky datového profilování

Připojení lze učinit k mnoha databázím, clusterům nebo se dají číst soubory z lokálního souborového systému. Kromě sloupcových statistik lze analyzovat i

[14] https://www.talend.com

závislosti mezi sloupci a metadata. Aplikace byla použita také na profilování objemného souboru o 15 milionech řádků, ale během jedné hodiny aplikace nevydala výsledek. Pozitivní však je, že analýza neprobíhá pouze v paměti RAM. Při výměně za mnohem výkonnější stroj nebo volbě lepší konfigurace je pravděpodobné, že analýza proběhne v pořádku.

2.2.6 Srovnání testovaných nástrojů

Všechny testované nástroje mají velmi podobné chování. Nejprve uživatel zadá, jaký datový zdroj chce profilovat. Některé z nástrojů navíc obsahují mechanismus pro určení metadat na základě struktury vstupních dat. Pak není potřeba definovat vstupní sloupce a jejich datové typy. Po specifikaci vstupních dat může uživatel přidat transformace a jiné procesy před samotným profilováním. Nakonec je potřeba nakonfigurovat nastavení konkrétních statistik pro vstupní sloupce.

Až na jednu výjimku žádný z testovaných nástrojů neumožňoval použití aproximativního přístupu pro získání výsledků. Zmíněným použitím je aproximativní algoritmus pro nalezení funkcionálních závislostí mezi sloupci v nástroji Informatica Data Quality. Tento nástroj však není volně dostupný, a proto tato informace byla získána pouze z dokumentace.

Překvapivě žádný nástroj nepodporoval čtení vzorků dat jinak než prvních k záznamů. Při čtení dat z databáze často existovala možnost upravit SQL dotaz, ve kterém lze náhodné vzorkování přidat. To však neplatí pro soubory, kde tato varianta neexistuje.

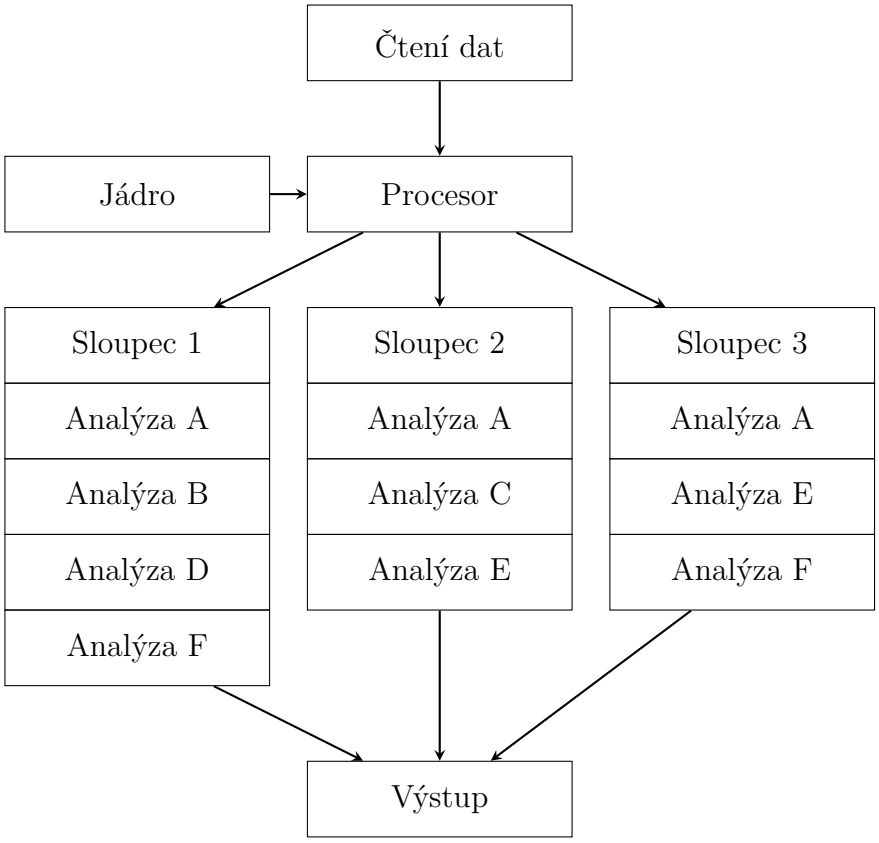
	Ataccama DQA	Ataccama ONE	DataCleaner	Aggregate Profiler	Talend Open Studio
Počet datových zdrojů	19	13	15	16	21
Podpora vzorkování SQL	+	+	+	+	+
Podpora vzorkování CSV	-	-	-	-	-
Snadná konfigurace metadat	-	+	+	-	+
Konfigurace statistik	++	-	+	-	+
Rozmanitost statistik	+	+	+	+	+
Vícesloupcové statistiky	+	-	+	+	-
Aproximativní algoritmy	-	-	-	-	-
Prezentace výsledků	+	++	++	+	+
Profilování velkých souborů	++	?	-	- -	-
Práce s omezenou pamětí	+	?	-	-	-

Tabulka 2.1: Srovnání volně dostupných nástrojů sloužících k profilování dat

3. Architektura

Architektura aplikace vychází z hlavního zaměření práce - porovnat více přístupů k datovému profilování. Všechny části aplikace jsou modulární, a právě díky tomu lze spouštět jednotlivé instance algoritmů pomocí konfigurace. Tímto postupem je umožněno snadné srovnávání různých přístupů k analýzám se stejnými vstupními daty. Návrh spočívá v rozdělení konfigurace, čtení dat, provádění analýz a tvorbě výstupu na jednotlivé komponenty, které mezi sebou komunikují pomocí rozhraní. O celou orchestraci chodu aplikace se pak stará jádro společně s procesorem.

Jádro aplikace načte vstupní konfiguraci, vytvoří potřebné instance komponent a spustí procesor. Procesor postupně získává vstupní data z komponenty určené pro čtení dat a předává hodnoty k analýze. Jakmile je celý vstup zpracován, práce procesoru skončí a následuje vytvoření výstupu. Po vypsání výsledků je celá aplikace ukončena.



Obrázek 3.1: Architektura analýzy dat

3.1 Konfigurace

Na vstupu přijímá aplikace konfigurační soubor ve formátu JSON. V této konfiguraci je definován zdroj vstupních dat a sloupce určené k analýze. Vybrané typy statistik jsou zde přiřazeny ke sloupcům. Dále konfigurace obsahuje parametrizaci jednotlivých algoritmů. Nakonec je možné definovat formát výstupu.

Jednotlivé ukázky konfigurací komponent jsou popsány v následujících sekcích. Dohromady tyto části konfigurace tvoří zmíněnou vstupní konfiguraci.

3.2 Čtení dat

Čtení dat probíhá jak z relační databáze, tak z CSV souborů. Komponenta je zodpovědná pouze za připojení se k datovému zdroji a čtení řádků. Přes rozhraní procesor následně získává jednotlivé řádky a předává je k analýze. Díky tomuto postupu je zachována jednoduchost čtecí komponenty. Ostatní komponenty pak využívají možností paralelního zpracování dat. Řádky nejsou pouze přečteny, ale jsou rozděleny dle sloupců. Hodnoty jsou převedeny na konkrétní datový typ programovacího jazyka.

V případě čtení z relační databáze probíhá komunikace mezi aplikací a zmíněnou databází pomocí jednotného rozhraní JDBC[1]. Použitím tohoto API je přítomna dostatečná abstrakce, aby byla aplikace použitelná pro více typů databázových systémů bez nutnosti zásahu v kódu. Kromě konfigurace (kód 3.1) nutných údajů k samotnému připojení do databáze je potřeba také uvést cestu k odpovídajícímu JDBC driveru a SQL dotaz `SELECT` pro získávání dat. Případné vzorkování dat lze přidat přímo do zmíněného SQL dotazu dle popisu v sekci 4.2.

```
"reader": {
    "type": "SQL",
    "jdbcString": "jdbc:postgresql://localhost:5432/test",
    "driverJar": "file:///C:/Data/postgresql-42.2.12.jar",
    "driverClass": "org.postgresql.Driver",
    "user": "user",
    "password": "password",
    "query": "select * from table;"
}
```

Kód 3.1: Konfigurace čtení z SQL databáze

Existuje zde také druhá varianta konfigurace čtení z databáze (kód 3.2). Stačí pouze uvést název tabulky a výsledný dotaz `SELECT` je vytvořen na základě definice sloupců (kód 3.1)[2].

V případě konfigurace profilování dat ze souborů (kód 3.3) je potřeba znát umístění konkrétního souboru. Dále je možné konfigurovat vzorkování dat a nakonec lze upřesnit další parametry pro správné načtení dat z CSV souboru.

Důležitou součástí konfigurace vstupu je definice sloupců (kód 3.4). Pro samotné čtení dat je potřeba znát název sloupce a jeho datový typ.

[1] Java Database Connectivity:
https://www.oracle.com/java/technologies/javase/javase-tech-database.html

[2] Generování dotazu se vzorkováním není podporováno kvůli odlišným databázovým syntaxím.

```
"reader": {
    "type": "SQL",
    "jdbcString": "jdbc:postgresql://localhost:5432/test",
    "driverJar": "file:///C:/Data/postgresql-42.2.12.jar",
    "driverClass": "org.postgresql.Driver",
    "user": "user",
    "password": "password",
    "table": "table"
}
```

Kód 3.2: Konfigurace čtení z SQL databáze

```
"reader": {
    "type": "CSV_SAMPLE",
    "path": "input_file.csv",
    "samplePercentage": 0.1, nebo "samplePercentage": 10000,
    "charset": "UTF-8",
    "containsHeader": true,
    "skipEmptyRows": false,
    "errorOnDifferentFieldCount": false,
    "fieldSeparator": ",",
    "textDelimiter": "\""
}
```

Kód 3.3: Konfigurace čtení vzorku z CSV souboru

```
"columns": [
    {
        "name": "column_1",
        "dataType": "STRING",
        ...
    }
]
```

Kód 3.4: Konfigurace vstupních sloupců

3.3 Analýza dat

Aplikace se primárně zaměřuje na sloupcovou analýzu dat. Analýzy jsou nezávislé implementace algoritmů, které na vstupu postupně dostávají jednotlivé hodnoty. Analýzy předem neznají celkový počet řádků, tudíž při každé nové hodnotě by mělo proběhnout její zpracování. Jednotlivé instance algoritmů jsou vytvořeny pro každý sloupec, kde je daná analýza definovaná. Neboli pokud bude analýza přiřazena ke třem sloupcům, pak budou existovat také tři instance dané analýzy, které se mohou lišit v konfiguraci parametrů.

V prvním kroku (kód 3.5) je nutné konfigurovat analýzy pomocí parametrů a přiřadit k nim unikátní identifikátory.

```
"analyzers": {
    "frequency_analyzer": {
        "type": "FREQUENCY_ANALYSIS",
        "params": {
            "algorithmType": "STICKY_SAMPLING"
            "outputCount": 20,
            "algorithmParams": {
                "minFrequency: 0.01,
                "error: 0.001,
                "failure: 0.0001
            }
        }
    }
}
```

Kód 3.5: Konfigurace analýzy

Jakmile je vytvořen seznam parametrizovatelných analýz, zbývá přiřadit tyto analýzy ke sloupcům (kód 3.6). Díky tomuto postupu lze pro více sloupců sdílet stejné parametry analýzy a zároveň lze aplikaci této analýzy omezit jen na vybrané sloupce.

```
"columns": [
    {
        "name": "column_1",
        "dataType": "STRING",
        "skip": false,
        "analyzers": [
            "string_analyzer",
            "frequency_analyzer",
            "histogram_analyzer"
        ]
    }
]
```

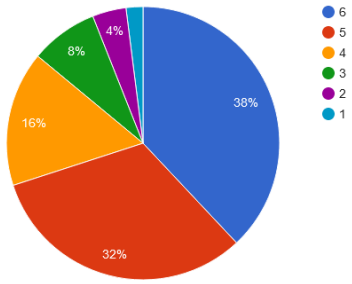
Kód 3.6: Konfigurace vstupních sloupců a přiřazených analýz

3.4 Výpis výsledků

Jakmile zpracování celého vstupu skončí, následuje vypsání výsledků. Základním výstupním formátem je JSON, druhým implementovaným formátem je jednoduchý HTML výstup. Předpokladem je, že JSON výstup slouží pro následné automatické zpracování a oproti tomu HTML výstup supluje chybějící uživatelské rozhraní a demonstruje zobrazení výsledků uživateli. Pro oba formáty výstupního souboru existují tři varianty zápisu výsledků analýz: výskyty hodnot (výstup 3.1), histogram[3] (výstup 3.2) a seznam klíč-hodnota (výstup 3.3). Zmíněné zápisy odpovídají potřebám jednotlivých analýz, které navíc nemusí díky dostatečné abstrakci implementovat rozdílné chování pro různé výstupní formáty.

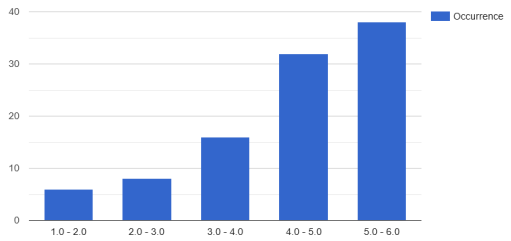
Ukázka kódu 3.7 znázorňuje, jak lze konfigurovat výstupní formát a umístění souboru s výsledky.

```
{
    "6" : 38,
    "5" : 32,
    "4" : 16,
    "3" : 8,
    "2" : 4,
    "1" : 2,
    "other" : 0
}
```



Výstup 3.1: Výskyty hodnot

```
[
{"from": 1, "to": 2, "count": 6},
{"from": 2, "to": 3, "count": 8},
{"from": 3, "to": 4, "count": 16},
{"from": 4, "to": 5, "count": 32},
{"from": 5, "to": 6, "count": 38}
]
```



Výstup 3.2: Histogram hodnot

```
{
    "max" : 6.0,
    "min" : 1.0,
    "range" : 5.0,
    "avg" : 4.86,
    "sum" : 486.0
}
```

Key	Value
max	6.0
min	1.0
range	5.0
avg	4.86
sum	486.0

Výstup 3.3: Klíč - hodnota

[3]V sekci 5.3 je definováno, že intervaly jsou zprava uzavřené.

```
"writer": {
    "type": "HTML",
    "file": "out.html"
}
```

Kód 3.7: Konfigurace výstupu s výsledky

3.5 Procesor a paralelismus

Architektonické řešení obsahuje jednoho producenta (čtení z datového zdroje) a mnoho konzumentů (sloupcové analýzy). Pro využití paralelního přístupu hovoří fakt, že čtení ze souborů je mnohem rychlejší než prováděné analýzy. Konzumenti navíc nepotřebují zpracovávat vždy stejný řádek, protože mezi nimi není žádná závislost. V případě čtení dat z databáze a přenosu dat po síti nelze v aplikaci zajistit zrychlení během provádění dotazu `SELECT` v databázovém serveru. Tudíž i v této variantě je největší prostor pro zrychlení běhu aplikace v samotné analýze dat. Důležitým požadavkem na implementaci aplikace je správné vyřešení situace v případě, že producent je rychlejší než konzumenti. Nesmí totiž nastat situace, kdy producent zahltí paměť přečtenými daty a program již nebude mít dostatek volného prostoru pro analýzu.

Bylo zkoumáno několik přístupů k navržení architektury a jako nejvhodnější se ukázala dvě řešení: omezené blokující fronty[4] a knihovna Disruptor[5].

Blokující fronta První zamýšlenou variantou, jak implementovat předávání dat mezi jedním producentem a několika konzumenty, je použití více vláken a fronty ArrayBlockingQueue.

Producent v tomto případě přidává nové prvky na jeden konec omezené blokující fronty a konzumenti je z opačného konce odebírají. Pokud se tato fronta naplní, pak je producent blokován, a naopak, když se fronta vyprázdní, čeká se na straně daného konzumenta.

Při této implementaci lze pro každé vlákno obsahující část konzumentů vytvořit jednu frontu a pomocí producenta vkládat reference na nové prvky do všech front zároveň. Pokud by aplikace měla využívat pouze jedinou frontu, pak by se jednalo o synchronizačně náročnější postup, který je však elegantně vyřešen v následující variantě.

Disruptor Podobný přístup jako při použití blokující fronty využívá knihovna Disruptor. Interně se jedná o cyklickou frontu[6], kde jsou producenti i konzumenti blokováni stejně jako v předchozím případě. Při této variantě není potřeba více front pro každé vlákno, protože knihovna interně drží informaci, které prvky byly již zpracovány všemi konzumenty.

Kromě potřeby pouze jediné fronty má knihovna Disruptor i jiné výhody oproti variantě s blokující frontou ArrayBlockingQueue. Díky znovupoužití ob-

[4] V programovacím jazyce Java se jedná o rozhraní java.util.concurrent.BlockingQueue

[5] https://www.baeldung.com/lmax-disruptor-concurrency

[6] http://mechanitis.blogspot.com/2011/06/dissecting-disruptor-whats-so-special.html

jektů ve frontě je lépe využitá keš procesoru a je snížený počet potřebných volání garbage collectoru pro dealokaci nepotřebných objektů v paměti. Testování ukázalo, že propustnost této implementace fronty v porovnání s ArrayBlockingQueue je až 7-8x vyšší (LMAX-Exchange, 2012).

RxJava a Akka Streams Existují také další vhodné frameworky pro implementaci interního procesoru, jako jsou například RXJava[7] a Akka Streams[8]. Jejich použití však není triviální a bez předchozí znalosti je obtížné z těchto zástupců vybrat správný framework. Změna by měla velký dopad na celou architekturu programu a výsledný výkon by ještě více závisel na správném použití vybraného frameworku. Použití knihovny Disruptor se ukázalo jako dostatečně obecné a rychlé.

Dávkové zpracování Následně je ještě implementováno rozšíření o dávkový algoritmus. Producent vytvoří dávku s N řádky a poté každý konzument analyzuje záznamy z jedné dávky hned za sebou. Řešení by mělo dosahovat vyššího výkonu kvůli keši procesoru, kdy data potřebná pro danou analýzu redukují výpadky keše.

3.6 Načtení konfigurace sloupců

Datové profilování často není osamocený proces. Před samotnou analýzou dat probíhá také analýza dostupných metadat, která jsou následně uložena v datovém katalogu. Na základě znalosti názvů sloupců a jejich datových typů lze již snadno nakonfigurovat proces datového profilování. V praxi je však tato konfigurace usnadněná pomocí uživatelského rozhraní.

Implementovaná aplikace neobsahuje ani uživatelské rozhraní, ani část, která by získávala informace o sloupcích. Ruční konfigurace tak může být velmi obtížná pro vstupní data o desítkách až stovkách sloupců. Z tohoto důvodu je načtení metadat datovým katalogem simulováno menším nástrojem, který je přiložený k práci. Nástroj přečte základní konfiguraci vstupního zdroje a vygeneruje zbytek potřebné konfigurace na základě obsahu tabulek nebo souborů a konfigurací analýz. S využitím jeho výstupu má uživatel možnost upravit parametry algoritmů a výsledná konfigurace je již použitelná k samotnému spuštění profilování dat.

Konfigurace aplikace pro načtení metadat (kód 3.9) vypadá velmi podobně jako konfigurace samotného profilování. Pro oba typy vstupních konfigurací jsou společné definice modulu pro čtení, analýz a výstupu. Při načítání metadat je navíc potřeba definice aplikací analýz, která popisuje, jaké instance analýz se mají přiřazovat ke konkrétním datovým typům vstupních sloupců.

Výsledná konfigurace poté obsahuje zkopírované společné definice, a navíc definice `applications` je nahrazena definicí konkrétních sloupců `columns`.

[7] https://github.com/ReactiveX/RxJava

[8] https://doc.akka.io/docs/akka/current/stream/index.html

```
{
  "reader": { ... },
  "applications": {
    "STRING": [
      "bloom_filter"
    ],
    "LONG": [
      "bloom_filter"
    ]
  },
  "analyzers": {
    "bloom_filter": { ... }
  },
  "writer": { ... }
}
```

Kód 3.8: Konfigurace pro načtení metadat

```
{
  "reader": { ... },
  "columns" : [ {
    "name" : "column_1",
    "dataType" : "STRING",
    "skip" : false,
    "analyzers" : [ "bloom_filter" ]
  } ],
  "analyzers": {
    "bloom_filter": { ... }
  },
  "writer": { ... }
}
```

Kód 3.9: Výsledná konfigurace po načtení metadat

4. Implementace

Pro přímou přenositelnost mezi různými platformami, snadnou implementaci a rozšířenost jazyka v praxi je aplikace napsána v Javě[1]. Důsledkem tohoto rozhodnutí je výhoda snadné komunikace s databázemi pomocí jednotného rozhraní JDBC a existence užitečných knihoven pro budoucí rozšíření o více datových zdrojů.

Zajímavou alternativou by mohlo být použití jazyka Go[2]. Hlavní výhoda by nastala v menší iniciální velikosti paměti RAM a nejspíše také ve vyšším výkonu. Analýza objemných dat je však především náročná na paměť, nehledě na jazyk. Tudíž by i v tomto případě bylo potřeba zvolit takové algoritmy, které nepotřebují udržovat celý vstup v paměti. I z tohoto důvodu je tedy varianta s implementací v Javě přijatelná.

Testování je zajištěno na úrovní Unit testů. Všechny analýzy mají velké pokrytí testy. Ostatní třídy jsou pokryty dle uvážení. Pro snadný zápis testů byl zvolen v dnešní době oblíbený jazyk Kotlin[3]. Jelikož bylo při tvorbě tohoto jazyka myšleno na souběžné používání s Javou, je tato kombinace velmi snadná. Kromě některých jednodušších konstrukcí nejsou ve zvoleném přístupu zásadní výhody, a tudíž by stejného výsledku šlo snadno docílit i v samotné Javě. Sestavení aplikace probíhá pomocí nástroje Maven[4].

4.1 Datové typy

Aplikace pracuje s pěti základními datovými typy:

- `STRING` (`java.lang.String`) - textové řetězce
- `LONG` (`java.lang.Long`) - 64bitové celočíselné hodnoty
- `DOUBLE` (`java.lang.Double`) - 64bitové reálné hodnoty podle standardu IEEE 754
- `DATE` (`java.util.Date`) - datumy s časem
- `BOOLEAN` (`java.lang.Boolean`) - logické hodnoty

Všechny datové typy podporují také prázdné hodnoty (`null`) a jednotlivé hodnoty stejného datového typu jsou vzájemně porovnatelné.

4.2 Vzorkování dat

Důležitou součástí aplikace je také podpora vzorkování vstupních dat. Maximální počet přečtených řádků při vzorkování lze vyjádřit jak absolutním počtem, tak procentuální částí řádků. Důležitou vlastností pro vstupní vzorek dat je rovnoměrné rozdělení přečtených řádků souboru. Pokud aplikace přečte například prvních 10 % dat, vzorek je dostatečně malý, a dojde tím ke zrychlení celého

[1]Java: https://www.java.com/en/ [2]https://golang.org [3]Kotlin: https://kotlinlang.org
[4]Maven: https://maven.apache.org

procesu. Data však nejsou dostatečně náhodná a je menší pravděpodobnost, že jsou pokryty všechny vlastnosti daného vstupu. Z tohoto důvodu je důležité, aby byl vzorek rovnoměrně tvořen daty z celého rozsahu souboru.

4.2.1 Vzorkování při čtení z databáze

V případě profilování dat z databáze je vhodné provést vzorkování dat přímo v SQL dotazu. Neproběhne pak zbytečný přenos dat. První z možností, jak vybrat náhodný vzorek dat z tabulky v databázi, je vygenerovat pro každý řádek náhodné číslo, setřídit data pomocí této hodnoty a vybrat pouze prvních k hodnot. Řešení je znázorněno v kódu 4.1.

```
SELECT * FROM table ORDER BY random() LIMIT 100;
```

Kód 4.1: Získání 100 náhodných řádků z tabulky *table* v databázi PostgreSQL

Pokud je potřeba získat vzorek o velikosti vyjádřené procenty, lze opět využít náhodné hodnoty, jako je tomu v ukázce kódu 4.2. Tento postup je možné aplikovat také pro variantu s absolutním počtem náhodných řádků. V kódu 4.3 nejprve proběhne dotaz na celkový počet řádků. Ten je následně převeden na variantu s relativním počtem řádků. Zároveň tato varianta nevyžaduje třídění, a tedy méně zatěžuje databázový server.

```
SELECT * FROM table WHERE random() < 0.1;
```

Kód 4.2: Získání přibližně 10 % náhodných řádků z tabulky *table* v databázi PostgreSQL

```
SELECT * FROM test WHERE random() < (SELECT 100.0/count(*) FROM test);
```

Kód 4.3: Druhá varianta získání přibližně 100 náhodných řádků z tabulky *table* v databázi PostgreSQL

Zmíněné varianty nemusejí být však optimální, proto databáze již často nabízejí přímou podporu vzorkování. Kód 4.4 znázorňuje, jak je možné získat 100 náhodných řádků z tabulky s podporou vzorkování v databázi PostgreSQL[5]. Případně lze také vytvořit dotaz na vzorek deseti procent dat pomocí kódu 4.5.

```
SELECT * FROM table TABLESAMPLE SYSTEM_ROWS (100);
```

Kód 4.4: Získání 100 náhodných řádků z tabulky *table* s podporou vzorkování v databázi PostgreSQL

[5]Dotaz SELECT v PostgreSQL: https://www.postgresql.org/docs/current/sql-select.html

```
SELECT * FROM table TABLESAMPLE SYSTEM (10);
```

Kód 4.5: Získání přibližně 10 % náhodných řádků z tabulky *table* s podporou vzorkování v databázi PostgreSQL

4.2.2 Vzorkování při čtení ze souboru

Při čtení vzorků z CSV souborů je situace komplikovanější. Bez použití komplexních frameworků nebyla nalezena snadná cesta, jak zajistit vzorkování pro CSV soubory. Právě proto je v aplikaci funkcionalita doimplementovaná a zde podrobněji rozebraná. Zmíněná implementace splňuje požadované vlastnosti - lze určit počet přečtených řádků jak v absolutním čísle, tak v poměru vůči celému souboru, a vzorek přečtených dat je rovnoměrně náhodný z celého souboru.

Na vstupu je soubor o B bytech a N řádcích. Velikost vzorku V je udána v procentech (V_{rel}) nebo v absolutní hodnotě (V_{abs}). Počet řádků však není znám bez přečtení celého souboru a pro rychlý běh analýzy není akceptovatelné, aby se při prvním průchodu určil počet řádků a při druhém se vytvářel výsledný vzorek dat. V pokročilé variantě použití aplikace může probíhat čtení z jiného serveru. Soubor by se přenášel mezi servery vícekrát nebo by se nejprve musel uložit a až poté by probíhala dvouprůchodová analýza. Použití zmíněné varianty s více průchody se tudíž projeví na rychlosti celé analýzy. Z tohoto důvodu, pokud je počet řádků potřebný, algoritmus toto číslo pouze odhadne, a tím je zajištěno, že není potřeba více než jedno čtení celého souboru.

Pokud se jedná o variantu s procentuální velikostí vzorku, kde $V_{\text{rel}}\epsilon(0,1)$, stačí pro každý řádek spustit podmínku $random() < V_{\text{rel}}$ pro funkci $random()$, která vrací náhodné číslo v intervalu $< 0,1)$. V případě pozitivního výsledku podmínky je daný řádek zpracován, jinak se tento řádek ignoruje. Díky tomuto postupu je na výstupu přibližně $V_{\text{rel}} * N$ záznamů.

V případě varianty s absolutní hodnotou pro $V_{\text{abs}} > 0$ nelze jednoduše použít předchozí varianta, protože není znám počet řádků N v souboru. Počet řádků v souboru je však nahrazen za jeho odhad. Většina datových souborů nemá příliš rozdílné délky řádků. Tudíž lze přečíst prvních k řádků a poté spočítat, kolik bytů má průměrně jeden řádek. Nakonec stačí celkový počet bytů B v souboru vydělit průměrným počtem bytů b_{avg_k} jednoho řádku, neboli

$$N' = \frac{B}{b_{\text{avg}_k}} \approx N. \tag{4.1}$$

Takto lze odhadnout celkový počet řádků N' v souboru a na základě tohoto výsledku lze již převést implementaci na předchozí variantu pomocí rovnice

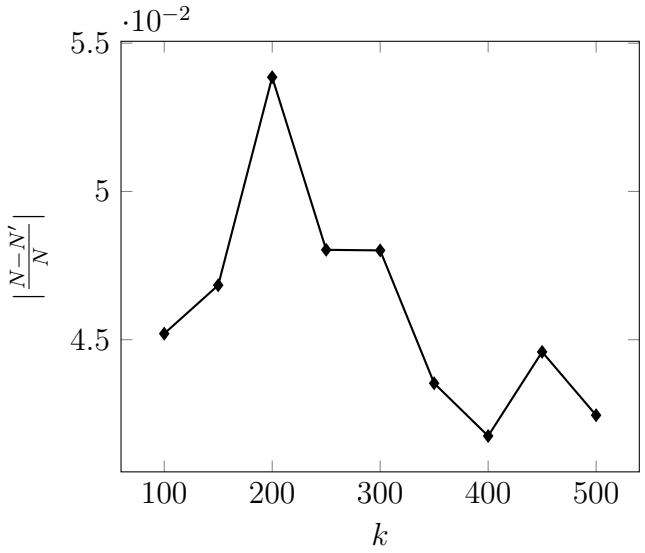
$$V_{\text{rel}} = \frac{V_{\text{abs}}}{N'} \tag{4.2}$$

a spustit algoritmus s velikostí vzorku V_{rel}.

Algoritmus sloužící k odhadu počtu řádků v CSV souboru byl aplikován na deset náhodných vstupních souborů stažených z webu Kaggle[6] o velikosti 5 až 40 miliónů řádků. Průměrná chyba odhadu se pohybovala mezi 1 až 7 procenty. Na

[6] https://www.kaggle.com

základě měření 4.1 se ukázalo, že přesnost odhadu byla průměrně 4.6 % nezávisle na velikosti k. Díky tomuto pozorování lze usoudit, že i čtením prvních 100 řádků lze vyprodukovat použitelný odhad.



Graf 4.1: Procentuální chyba odhadu počtu řádků při různém k

4.3 Vytváření komponent - návrhový vzor Factory

Velký důraz byl při implementaci kladen na rozšiřitelnost jednotlivých komponent aplikace, spouštění vybraných algoritmů na základě konfigurace a možnost jejich parametrizace uživatelem. Pro splnění těchto požadavků je použit jednotný postup pro rozšiřování dílčích částí aplikace a na základě konfigurace se vytvářejí nové instance tříd. Aplikace je založena na principu Inversion of Control s implementací pomocí návrhového vzoru Factory.

Použitá implementace návrhového vzoru Factory je znázorněná v kódu 4.6. Nejprve je potřeba implementovat rozhraní `Component`, kde se ve funkci `doAction` nachází hlavní daná logika komponenty. Třída může mít libovolný konstruktor, jeho parametry však odpovídají obsahu příslušné implementace rozhraní `ComponentParams`. Dalším krokem je vytvoření implementace rozhraní `ComponentFactory`, která na základě parametrů vytváří nové instance komponent. Jednotlivé implementace `ComponentFactory` jsou následně registrovány ve výčtovém typu `ComponentType`, pomocí kterého již lze snadno vytvářet libovolné instance na základě konfigurace přímo od uživatele. Dle typu komponenty a za použití načtené konfigurace je poté vykonán kód 4.7, který se již postará o zmíněné vytvoření instance.

Pro některé implementace analýz je zvolena dvoustupňová logika tvorby nových instancí. Nejprve je potřeba návrhový vzor Factory aplikovat na vytvoření analýz a poté znovu interně pro vytvoření jednotlivých algoritmů. Díky tomuto návrhu může uživatel v konfiguraci zvolit libovolný algoritmus a předat mu parametry. Implementace rozhraní `ComponentParams` také obsahují výchozí hodnoty pro případ, kdy uživatel některé parametry aplikaci nepředá.

```
interface Component {
    void doAction();
}

interface ComponentParams {
}

class DemoComponent implements Component {
    public DemoComponent(/* params */) {
        /* init */
    }

    @Override
    public void doAction() {
        // action
    }
}

class DemoParams implements ComponentParams {
    // values
}

interface ComponentFactory<T extends ComponentParams> {
    Component createInstance(T params);
}

class DemoComponentFactory implements ComponentFactory<DemoParams> {
    @Override
    DemoComponent createInstance(DemoParams params) {
        return new DemoComponent(/* params */);
    }
}

enum ComponentType {
    // more component implementations
    DEMO_COMPONENT(new DemoComponentFactory());

    private final ComponentFactory factory;

    ComponentType(ComponentFactory factory) {
        this.factory = factory;
    }

    Component newInstance(ComponentParams params) {
        return factory.createInstance(params);
    }
}
```

Kód 4.6: Vytváření komponent pomocí návrhového vzoru Factory

```
Component component = config.getComponentType().newInstance(params)
```

Kód 4.7: Vytváření nových instancí komponent

4.4 Proxy (zástupce) pro analýzy

Jednotlivé implementace analýz často pracují s určitými omezeními. Některé analýzy nemusí podporovat prázdné hodnoty nebo mohou pracovat pouze nad určitým datovým typem. Aby se zredukoval duplicitní kód, kdy různé analýzy analogickým způsobem přeskakují prázdné hodnoty nebo validují datový typ hodnoty, vznikl mechanizmus implementující návrhový vzor Proxy neboli Zástupce. Pokud je implementace analýzy označena některou z anotací `@NotNullAnalyzer`, `@NumberAnalyzer` a `@StringAnalyzer`, pak je instance ještě obalena pomocí delegování příslušnou dodatečnou kontrolou nebo konverzí. Tato obálka dodává společnou funkcionalitu jednotlivým implementacím. Kód 4.8 znázorňuje implementaci jedné z možných Proxy.

```
@Slf4j
@RequiredArgsConstructor
public class NotNullAnalyzerWrapper implements ValueAnalyzer {

    private final ValueAnalyzer analyzer;

    @Override
    public void analyze(Object value) {
        if (value != null) {
            analyzer.analyze(value);
        } else {
            log.debug("skippedValue={}", value);
        }
    }

    @Override
    public void writeResults(ColumnResultWriter columnResultWriter) {
        analyzer.writeResults(columnResultWriter);
    }
}
```

Kód 4.8: Proxy v případě použití anotace `@NotNullAnalyzer`

4.5 Významná rozhraní

- **`ValueAnalyzer`** - Implementace rozhraní `ValueAnalyzer` slouží k vytváření jednotlivých analýz. Vzniklé třídy obsahují metodu `next`, která provede zpracování vstupní hodnoty, a dále metodu `writeResults`, ve které proběhne pomocí `ColumnResultWriter` zápis výsledků určených k výstupu. Za běhu aplikace je přiřazena nová instance třídy k právě jednomu sloupci.

- **`AnalyzerParams`** - Rozhraní `AnalyzerParams` slouží pro parametrizaci analýz. Předpokladem je, že se konkrétní obsah této třídy nastaví na základě uživatelovy konfigurace. Proměnné v jednotlivých implementacích odpovídají konstruktorům příslušných algoritmů. Navíc je zde možnost definice výchozích hodnot pro případ, kdy uživatel některý z potřebných parametrů neuvede.

- **`ValueAnalyzerType`** - Výčtový typ `ValueAnalyzerType` za pomocí implementací `ValueAnalyzerFactory` slouží k vytváření instancí analýz pomocí postupu popsaného v sekci 4.3. Implementace rozhraní `ValueAnalyzerFactory` odpovídají vždy jedné implementaci `ValueAnalyzer` a jsou registrovány ve výčtovém typu `ValueAnalyzerType`.

- **`ProfilerProcessor`** - Pro implementace různých variant procesorů slouží rozhraní `ProfilerProcessor`. Procesor řídí chod aplikace během zpracování dle postupu, který je definován v sekci 3. Při startu pomocí metody **`run`** dostává jednu instanci `RowReader` a seznam instancí analýz roztříděných podle sloupců pomocí třídy `ColumnAnalyzers`.

- **`RowReader`** - Čtení vstupních tabulek a souborů probíhá odděleně od zpracování dat. Komunikace probíhá pomocí rozhraní `RowReader`, na kterém je možné volat metodu `nextRow` pro získáni následujícího řádku. Hodnoty jednoho řádku jsou předávány pomocí datového objektu `DataRow`. Data jsou dynamicky čtena ze zdroje po dávkách.

  Stejně jako jsou analýzy vytvářené pomocí návrhového vzoru Factory ze sekce 4.3, tak i tato komponenta je při běhu instanciována stejným postupem. Pro tyto účely slouží výčtový typ `ReaderType` společně s rozhraními `RowReaderFactory` a `RowReaderConfig`.

- **`ResultWriter`** - Výpis výsledků probíhá pomocí rozhraní `ResultWriter`. Interně je však rozhraní zodpovědné pouze za kompletování výsledků analyzovaných sloupců. Pro vytváření výsledků jednotlivých analýz slouží rozhraní `ColumnResultWriter`, které již podporuje všechny potřebné formy výstupů ze sekce 3.4.

  Nakonec i pro tuto komponentu existují třídy `ResultWriterFactory`, `WriterConfig` a `WriterType` sloužící k vytváření nových instancí dle postupu zmíněného v sekci 4.3.

4.6 Hešování

Podstatná část algoritmů závisí na kvalitním a rychlém hešování. Z toho důvodu bylo potřeba vybrat vhodnou hešovací funkci. Pro hešování byla nakonec zvolena knihovna Guava[7], která také doporučuje hešovací funkci MurmurHash3[8].

Jelikož aplikace pracuje s datovým typem `Object` a hešovací funkce potřebuje znát konkrétní datový typ, je tento kód skryt ve třídě `ObjectHashFunction`. Také

[7] https://github.com/google/guava/wiki/HashingExplained
[8] https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

volba hešovací funkce je díky tomu nastavena na jediném globálním místě, a tím je v budoucnu snadné přepnout interní implementaci hešování.

4.7 Možnosti rozšíření aplikace

Sekce obsahuje postup, jak lze aplikaci rozšířit o další datové zdroje a analýzy. Obě rozšíření vycházejí z postupu popsaného v sekci 4.3. Obdobným způsobem lze také doimplementovat nový výstupní formát nebo další variantu procesoru.

4.7.1 Přidání nového datové zdroje

Pro implementaci nového čtecího modulu jsou potřeba následující tři kroky:

1. Nejprve je potřeba implementovat rozhraní `RowReader`[9]. Rozhraní obsahuje metodu `DataRow nextRow(DataRow dataRow)`, která má za cíl produkovat nové řádky při čtení z datového zdroje. V případě konce vstupu je použitá konvence s návratovou hodnotou `null`. Díky rozšíření zmíněného rozhraní o další funkcionalitu pomocí rozhraní `java.lang.AutoCloseable` je také nutné implementovat metodu `void close()`. Metoda se zavolá na konci čtení a jejím cílem je ukončit připojení k datovému zdroji.

2. Jakmile je implementace modulu připravena, následuje vytvoření implementace rozhraní `RowReaderFactory`. Tento postup je nutný pro vytváření nových instancí pomocí návrhového vzoru Factory. Rozhraní vynucuje implementaci metody `R createInstance(C config, List<ColumnConfig> columns)`, která dostává na vstupu konfiguraci modulu a seznam vstupních sloupců. Konfigurace se vytváří pomocí třídy `RowReaderConfig`. Její obsah by měl odpovídat konstruktoru pro vytváření instancí nového modulu. Stejně tak musí být deserializovatelná z formátu JSON.

3. Nakonec, když jsou všechny tři třídy připraveny, je potřeba zaregistrovat novou implementaci do třídy `ReaderType`.

4.7.2 Implementace nové analýzy

Při rozšiřování aplikace o další analýzy je postup velmi obdobný jako při implementaci nového modulu pro čtení:

1. Algoritmus nové analýzy je potřeba umístit do implementace metody `void analyze(Object value)` z rozhraní `ValueAnalyzer`[10], která je volána pro každou vstupní hodnotu. Druhá metoda k implementaci ze zmíněného rozhraní je `void writeResults(ColumnResultWriter colResultWriter)`. Jakmile aplikace skončí výpočet, proběhne volání právě této metody na všech instancích analýz. Jejím cílem je vytvořit výsledky pomocí třídy `ColumnResultWriter`.

[9] Moduly pro čtení jsou v balíčku `com.kolek.lukas.approx.profiler.reader`.
[10] Analýzy jsou v balíčku `com.kolek.lukas.approx.profiler.analysis`.

2. Stejně jako v předchozím případě jsou pro správné použití návrhového vzoru Factory potřeba implementace rozhraní `RowAnalyzerFactory` a `AnalyzerParams`.

3. V posledním kroku je nutné zaregistrovat také novou analýzu do typu `ValueAnalyzerType`.

5. Analýzy

Práce se soustředí především na porovnání aproximativních algoritmů jak mezi sebou, tak i s přesnou variantou. Hlavní zaměření je na frekvenční analýzu, počítání kardinality, histogramů a kvantilů. Zmíněné analýzy jsou implementovány v několika variantách, vždy doplněných o měření jejich přesnosti na základě velikosti vstupu a potřebné paměti. Kromě těchto analýz kapitola popisuje implementaci procesu pro klasifikaci dat, která je nedílnou součástí profilovacího nástroje. V závěru jsou popsány ještě další, doplňující analýzy, bez kterých by se datové profilování neobešlo. Jejich implementace jsou však natolik přímočaré, že již měření neobsahují.

5.1 Frekvenční analýza

Cílem frekvenční analýzy je najít frekvence k hodnot s největším výskytem ve vstupních datech. Analýza na vstupu dostává hodnoty libovolného datového typu. Výstupem jsou dvojice hodnota a její frekvence v datech. Analýza může odhalit, zda některé hodnoty převyšují výskyty ostatních hodnot, a pomocí této informace může uživatel snadněji dál pracovat s analyzovanými daty. Na základě výstupu frekvenční analýzy lze také identifikovat referenční data nebo číselníky[1].

Při výpočtu frekvencí nad velkým objemem dat existuje zásadní problém: přesný algoritmus potřebuje současně uchovávat všechny unikátní hodnoty s jejich frekvencemi. Práce proto obsahuje kromě přesného algoritmu i několik aproximativních algoritmů, které pracují s omezeným počtem hodnot v paměti. Interní paměť těchto algoritmů lze před spuštěním konfigurovat na základě očekávané velikosti vstupu. Tím se omezí paměťová náročnost celého algoritmu analýzy.

5.1.1 Přesný algoritmus

Pro dosažení přesného výsledku frekvenční analýzy je potřeba průběžně ukládat všechny unikátní hodnoty a ke každé hodnotě udržovat počítadlo s její frekvencí ve vstupních datech (kód 5.2).

Implementace používá k uchovávání hodnot s počítadly třídu *java.util.HashMap*. Pro aktualizaci frekvencí je ideální funkce *merge*. Pokud při volání této funkce není prvek ve zmíněné kolekci ještě obsažen, vloží se s definovanou prvotní hodnotou - v tomto případě 1. Jestliže prvek již existuje, zavolá se funkce, která přemapuje uložený počet. Použitá lambda funkce zvýší hodnotu o jedna. Ve výsledku je tento přístup rychlejší než volání kombinace metod `cointains/get/put` díky menšímu počtu vyhledávání v kolekci.

```
map.merge(pvek, 1, (ulozenaH, novaH) -> ulozenaH + novaH);
```

Kód 5.1: Funkce java.util.HashMap.merge

[1] http://www.dataquality.cz/index.php?ID=5&ArtID=37&clanek=201502_MDM

```
Vstup:
N: prvky na vstupu

Algoritmus:
C <- datová struktura s dvojicemi prvek-počítadlo

pro každé n z N
    if C obsahuje n then
        zvýšit frekvenci n v C
    else
        vložit prvek n do C s frekvencí 1
    end
end

Výstup:
TOPk hodnot s frekvencemi z C
```

Kód 5.2: Přesný výpočet frekvenční analýzy

5.1.2 Lossy Counting

Prvním aproximačním algoritmem, který počítá frekvenční analýzu, je Lossy Counting (Manku a Motwani, 2002)(Vogiatzis, 2015). Algoritmus (kód 5.3) pracuje s daty rozloženými na úseky neboli okna. Pro každé okno se spočítá frekvenční analýza. Jakmile výpočet dorazí na konec daného okna, proběhne spojení nových hodnot s již známými frekvencemi. Poté se všechna počítadla sníží o jedna a prvky s nulovými počítadly se odstraní.

Jelikož algoritmus snižuje hodnoty počítadla a uživatele zajímá kromě množiny nejčastějších prvků také jejich frekvence, počítadlo je zdvojené. První počítadlo funguje přesně podle výše popsaného algoritmu. Druhé počítadlo jenom kopíruje všechny inkrementy prvního počítadla, ale nikdy se nezmenšuje. Výsledný odhad frekvence je tak uložen ve druhém z počítadel.

Algoritmus je konfigurovatelný pomocí parametrů s - minimální frekvence hodnot na výstupu - a ϵ - maximální chyba frekvence pro hodnotu. Rozumné parametry algoritmu jsou $s = 0{,}01$ a $\epsilon = 0{,}001$. Tímto se algoritmus dostane na maximálně $\epsilon^{-1} \log(\epsilon N)$ souběžně uložených dvojic v paměti. Ve výsledku budou frekvence se zastoupením alespoň 1 % s chybou menší než 0,1 %.

```
Vstup:
s: minimální frekvence na výstupu
e: maximální chyba
N: prvky na vstupu

Algoritmus:
w <- šířka jednoho okna 1/e
C <- datová struktura s dvojicemi prvek-počítadlo celkem
C_w <- datová struktura s dvojicemi prvek-počítadlo daného okna

pro každé n_i z N
    if C_w obsahuje n then
        zvýšit frekvenci n_i v C
    else
        vložit prvek n_i do C_w s frekvencí 1
    end

    if n_i mod w == 0 nebo N nemá již další hodnoty then
        spojit C s C_w
        snížit všechna počítadla o 1
        odebrat prvky s počítadlem rovným 0
    else
end

Výstup:
TOPk hodnost z C s frekvencemi >= s*|N|
```

Kód 5.3: Algoritmus Lossy Counting

5.1.3 Sticky Sampling

Následující pravděpodobnostní algoritmus, který počítá frekvenční analýzu, je Sticky Sampling (Manku a Motwani, 2002)(Vogiatzis, 2015). Algoritmus (kód 5.4) pracuje na bázi zpracování dat po oknech. Potřebné parametry jsou: minimální vrácená frekvence s, maximální chyba ϵ a pravděpodobnost selhání algoritmu δ. Vhodné nastavení parametrů bez znalosti vlastností datového vstupu je: $s = 0{,}01$, $\epsilon = 0{,}001$ a $\delta = 0{,}0001$. Iniciální velikost okna $2t$ je definována rovnicí:

$$t = \frac{1}{\epsilon} \log(s^{-1}\delta^{-1}). \tag{5.1}$$

Pokud prvek již existuje v datové struktuře, je počítadlo jeho hodnoty zvýšeno o 1. Naopak neznámý prvek je vložen s pravděpodobností $1/r$. Proměnná r je nastavena pro prvních $2t$ prvků na hodnotu 1. Po každém dokončení okna je proměnná r zdvojnásobena a velikost okna je rovna rt.

Po dokončení zpracování dat z jednotlivých oken navíc probíhá procedura `redukceHodnot` (kód 5.5). Ta pro každou uloženou hodnotu „hází mincí“, dokud nepadne 1. Za všechny výsledky 0 se sníží počítadlo dané hodnoty o jedna. Pokud hodnota počítadla klesne pod 1, je tento prvek z datové struktury odstraněn. Díky této proceduře je s jistou pravděpodobností udržována omezená velikost paměti.

Díky snižování počítadel se však přichází o přesnost frekvencí prvků. Proto je u každého prvku také uloženo, kolikrát byla frekvence tohoto prvku snížena. Při tvorbě výsledků je pak tato hodnota připočtena. Očekávaný maximální počet uložených prvků v paměti je $2t$.

```
Vstup:
s: minimální frekvence na výstupu
e: maximální chyba
d: pravděpodobnost selhání algoritmu
N: prvky na vstupu

Algoritmus:
t <- 1/e * Math.log(1/(s * d))
r <- 1
w <- velikost okna nastavená na počáteční hodnotu 2 * t
C <- datová struktura s globálními dvojicemi prvek-počítadlo

pro každé n_i z N
    if C obsahuje n_i then
        zvýšit frekvenci n_i v C
    else if random() < 1/r then
        vložit prvek n_i do C s frekvencí 1
    end

    w <- w - 1

    if w == 0 then
        redukceHodnot()
        r <- r * 2
        w <- t * r
    else
end

Výstup:
TOPk hodnot z C s frekvencemi > s*|N|
```

Kód 5.4: Algoritmus Sticky Sampling

```
Vstup:
C <- datová struktura s dvojicemi prvek-počítadlo

Algoritmus
pro každé c z C
    while "hodit mincí" == 0 then
        snížit počítadlo c o 1
    end

    if frekvence c < 1 then
        odebrat c z C
    end
end

Výstup:
Všechny zbylé hodnoty s frekvencemi z C
```

Kód 5.5: Funkce *redukceHodnot* v algoritmus Sticky Sampling

5.1.4 Space-Saving

Další algoritmus (kód 5.6) Space-Saving (Metwally a kol., 2004) spočívá v monitorování určitého počtu prvků. Prvky s nejmenšími výskyty se nahrazují novými. Pokud je prvek e již uložen v datové struktuře, stačí pouze zvýšit jeho frekvenci f_{e}. Pokud se prvek ve struktuře nenachází, je nalezen minimální prvek e_{min} s frekvencí f_{min}. Poté je nahrazen e_{min} prvkem e a frekvence f_{e} se nastaví na $f_{\mathrm{min}} + 1$.

Počet počítadel m je určen jako $m = 1/\epsilon$, kde parametr ϵ představuje maximální relativní chybu frekvence libovolného prvku. Díky této konfiguraci algoritmu jsou na výstupu pouze prvky s větší frekvencí než ϵN neboli s frekvencí větší než N/m. Prvky, které nejsou na výstupu, mají frekvenci mezi 0 a minimem z frekvencí mezi vrácenými prvky.

Jelikož algoritmus pro každý prvek pokládá dotaz, zda se prvek již nachází v datové struktuře, je využita hešmapa. Klíč představuje prvek, hodnotu pak počítadlo frekvence. Velikost mapy je limitována parametrem m. Dotaz na existenci a zároveň zvýšení počítadla má výslednou očekávanou časovou složitost $O(1)$.

Pro častý dotaz na minimum z uložených prvků je využitá halda. Díky tomu odpovídá časová složitost zmíněného dotazu $O(1)$. Změna frekvence i náhrada minimálního prvku v haldě pracuje v nejhorším případě v $O(log(m))$, kde $log(m)$ je konstanta, tudíž lze říct, že pracuje opět v $O(1)$.

Celkem tedy algoritmus ukládá v paměti maximálně m prvků a pomocí spojení obou datových struktur pracuje v časové složitosti $O(N)$.

```
Vstup:
e: maximální relativní chyba
N: prvky na vstupu

Algoritmus:
m <- 1/e
C <- datová struktura s m počítadly

pro každé n z N
    if C obsahuje n then
        zvýšit frekvenci n v C
    else
        min <- prvek s minimální frekvencí v C
        odebrat min z C
        vložit n s frekvencí min + 1
    end
end

Výstup:
TOPk hodnot s frekvencemi z C
```

Kód 5.6: Algoritmus Space-Saving

5.1.5 Count-Min Sketch

Oproti předchozím příkladům pracuje algoritmus Count-Min Sketch (kód 5.7) (Cormode a Muthukrishnan, 2004) s hešováním. Algoritmus používá l hešovacích funkcí a ke každé funkci h_i patří pole počítadel p_i délky m. Výsledný heš x funkce h_i představuje index v poli p_i. Z tohoto důvodu jsou potřeba hešovací funkce s délkou heše alespoň $\lceil \log_2(m) \rceil$.

Každý prvek je zahešován postupně všemi funkcemi. Příslušné počítadlo na indexu daném hešem je zvýšeno o 1. Frekvence prvku se rovná minimu z indexovaných počítadel v předchozím kroku. Díky hešování více funkcemi a následnému vybrání minima z hodnot je pomocí redukce možných kolizí snížena šance na nepřesný výsledek. Skutečná frekvence poté není s jistotou větší než výsledná frekvence, protože pro každý výskyt prvku je vždy minimum zvýšeno o jedna. Jedině při kolizích ve všech polích je výsledná frekvence větší než skutečná.

V ukázce (tabulka 5.1) je znázorněn postup algoritmu. Hodnota x je postupně zahešována funkcemi h_i. Jejich výsledek určuje index v polích počítadel. U jednotlivých počítadel je pak znázorněna zvýšená hodnota o 1. U funkce h_3 lze vidět, že v jednom z předchozích kroků nastala kolize. Z tohoto důvodu se pro výslednou frekvenci bere minimální hodnota, která je v tomto případě 3.

Popsaná datová struktura dokáže pro prvek odpovědět, jaká je jeho frekvence. Jelikož jsou však uložené pouze heše, nelze podat dotaz na k prvků s největšími frekvencemi. Z tohoto důvodu implementace využívá haldu, která je navíc adresovaná pomocí mapy jako v předchozím případě. Kromě toho je limitovaná maximálním počtem hodnot k. Mimo představený algoritmus se při každém vložení prvku také aktualizuje frekvence ve zmíněné haldě. Pokud prvek v haldě

	i_1	i_2	i_3	i_4	i_5	...
$h_1(x) = 2$	0	2+1	1	0	0	...
$h_2(x) = 1$	2+1	0	0	0	5	...
$h_3(x) = 5$	0	0	3	0	3+1	...
$h_4(x) = 4$	4	0	0	2+1	1	...

Tabulka 5.1: Ukázka algoritmu Count-Min Sketch s kolizí

již existuje, pouze se zvýší jeho frekvence a halda se případně upraví. Jestliže však prvek neexistuje, může dojít k záměně prvku za minimální prvek. Musí však platit, že zmíněný prvek má větší frekvenci, než je aktuální minimum v haldě.

Algoritmus potřebuje uložit v paměti $m * l$ počítadel. Průběžně také spravuje k hodnot s nejvyšší frekvencí. Za předpokladu využití omezené haldy a mapy jako u algoritmu Space-Saving pracuje v časové složitosti *O(N)*.

```
Vstup:
m: délka heše
l: počet hešovacích funkcí
k: počet uložených hodnot s frekvencemi
N: prvky na vstupu

Algoritmus:
H <- l hešovacích funkcí s délkou heše ceil(log2(m))
P <- l * m počítadel
C <- struktura s k dvojicemi hodnota-frekvence

pro každé n z N
    pro každé hi z H
        z <- heš hodnoty n pomocí hi
        zvýšit hodnotu počítadla Hiz
        f <- min(f, Hiz)
    end

    if |C| < l or C obsahuje n then
        zvýšit hodnotu n
    else if f > fmin v C then
        nahradit minimum v C prvkem n
        nastavit frekvenci n v C na fmin
    end
end

Výstup:
TOPk hodnot s frekvencemi z C
```

Kód 5.7: Algoritmus Count-Min Sketch

Vhodné nastavení konstant l a m (Cormode a Muthukrishnan, 2004) je:

$$l = \frac{e}{\epsilon} \quad \text{a} \quad m = -\log(\frac{1}{\delta}), \tag{5.2}$$

kde ϵ představuje maximální relativní chybu odpovědi s pravděpodobností $1 - \delta$. Výsledná paměť algoritmu (bez TOPk uložených hodnot) je:

$$l * m = -\frac{e}{\epsilon} \log(\frac{1}{\delta}) \text{ bitů}, \tag{5.3}$$

5.1.6 Analýza

Testování zmíněných čtyř aproximativních algoritmů probíhalo nejprve na dvou vygenerovaných sadách, které měly velikosti několika miliónů řádků s maximálně statisíci unikátními hodnotami. Rozdělení prvků v první datové sadě je znázorněné v tabulce 5.2. Druhá sada byla podobná, jednalo se totiž o Zipf rozdělení[2]. Toto rozdělení se nejčastěji používá při analýze algoritmů pracujících s frekvencemi hodnot. Poté proběhla analýza reálných dat.

prvek	*p*
0	0.32
1	0.16
2	0.08
3	0.04
4	0.02
5	0.01
...	...
17	0.00002
18	0.00001
...	...
360 020	0.00001

Tabulka 5.2: Dataset A

Výsledky měření ukázaly, že při nastavení chyby na $\epsilon = 0{,}1$ % u algoritmů Lossy counting, Sticky Sampling a Space saving jsou odhady v případě dotazu TOP20 velmi přesné. Pro žádný ze vstupních souborů nebyla absolutní chyba frekvencí jednotlivých hodnot větší než v řádu jednotek. V žádném z měření nebylo zaměněno pořadí prvních k hodnot dle jejich frekvencí. Algoritmus Count-Min Sketch produkuje díky odlišnému přístupu větší chyby. Jeho chování je následně podrobněji popsáno.

Implementaci frekvenční analýzy pomocí algoritmu Lossy Counting nelze použít pro profilování dat s omezenou pamětí. Jeho paměťová náročnost závisí na velikosti vstupu. Například pro vstup o délce 10 miliónů hodnot při $\epsilon = 0{,}001$ je odhad paměti přibližně 10 tisíc uložených hodnot s počítadly v paměti. V případě algoritmu Sticky Sampling je situace lepší. Paměť není závislá na velikosti

[2]https://mathworld.wolfram.com/ZipfDistribution.html

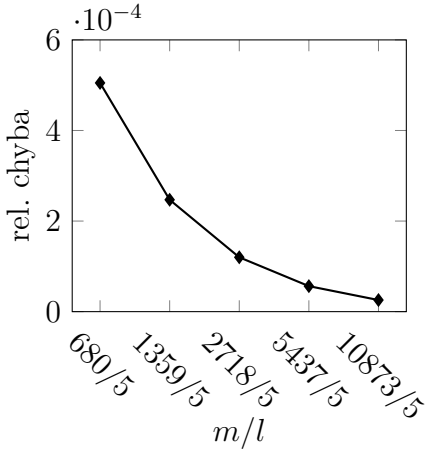
vstupu, ale pro zmíněné parametry $s = 0{,}01$, $\epsilon = 0{,}001$ a $\delta = 0{,}0001$ se maximum může dostat až k 12 tisícům prvků.

Pro profilování dat s omezenou pamětí jsou obě předchozí hodnoty maximální interní paměti stále velmi vysoké. Během analýzy dat lze předpokládat, že se instance frekvenční analýzy budou vyskytovat skoro u všech vstupních sloupců. V obou případech pak může počet hodnot v paměti snadno růst ke statisícům. Algoritmus Space-Saving překonává oba předchozí algoritmy. Pro stejnou hodnotu $\epsilon = 0{,}001$ je potřeba pouze 1000 prvků v paměti bez závislosti na velikosti vstupu.

Čtvrtý implementovaný aproximativní přístup využívá hešování. Implementovaná verze algoritmu Count-Min Sketch ukládá pouze TOPk prvků v paměti. K tomu navíc potřebuje ještě pole počítadel. Velikost paměti je pak závislá na velikostech parametrů ϵ a δ.

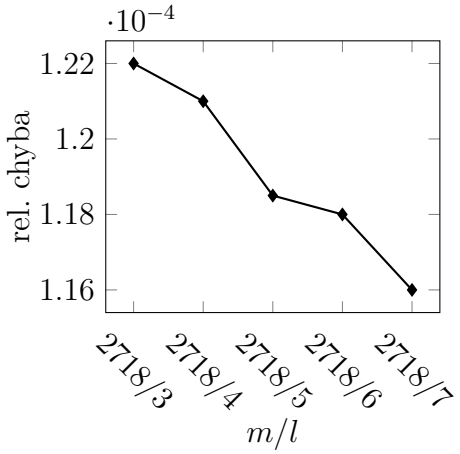
Uvedené měření algoritmu Count-Min Sketch se zaměřuje na závislost konstant ϵ a δ na výsledné přesnosti výsledků. V prvním případě (obrázek 5.1) je snižovaná konstanta ϵ a naopak ve druhém (obrázek 5.2) je snižovaná hodnota δ. Z výsledků lze vyčíst, že mnohem větší závislost přesnosti odhadů je na konstantě ϵ neboli na velikosti m, a tím potřebné paměti. Při snižování hodnoty δ výsledná paměť sice také roste, ale velikost chyby klesá řádově 100x pomaleji oproti předchozímu případu. Z toho vyplývá, že není ideální primárně zvyšovat počet hešovacích funkcí l, a přesnost výpočtu je nejvíce závislá na velikosti pole m.

ϵ	δ	m	l	paměť
0.004	0.01	680	5	13KB
0.002	0.01	1359	5	26KB
0.001	0.01	2718	5	53KB
0.0005	0.01	5437	5	106KB
0.00025	0.01	10873	5	212KB



Obrázek 5.1: Count-Min Sketch: závislost ϵ na přesnosti frekvencí TOP20 prvků (m = délka pole, l = počet hešovacích funkcí)

ϵ	δ	m	l	paměť
0.001	0.03	2718	3	31KB
0.001	0.02	2718	4	42KB
0.001	0.01	2718	5	53KB
0.001	0.0033	2718	6	63KB
0.001	0.0011	2718	7	74KB



Obrázek 5.2: Count-Min Sketch: závislost δ na přesnosti frekvencí TOP20 prvků (m = délka pole, l = počet hešovacích funkcí)

Nevýhodou algoritmu Count-Min Sketch je, že výpočet frekvencí TOP20 hodnot je skoro vždy s chybou, i když pouze řádově v tisícinách procent. Pro vstupní data, která obsahují dlouhé znakové řetězce, je ale implementace pomocí algoritmu Count-Min Sketch paměťově nejméně náročná. V ostatních případech je paměťová náročnost srovnatelná s algoritmem Space-Saving. Algoritmus Space-Saving má navíc výbornou přesnost u náhodných dat. Na základě toho lze ze zmíněných implementací označit variantu Space-Saving jako nejlepší pro frekvenční analýzu. Count-Min Sketch je pak vhodný pro sloupce s hodnotami zabírajícími větší paměť, jako jsou například dlouhé znakové řetězce.

V případě frekvenční analýzy náhodného vzorku dat se nejedná o žádné překvapení. Například pro vzorek dat o velikosti 40 % řádků lze očekávat 60% pokles všech výsledných frekvencí. V tomto případě může frekvenční analýza pomoci detekovat hodnoty, které svou frekvencí výrazně převyšují ostatní. Pokud jsou ale všechny frekvence v malém rozmezí a žádné se příliš nevychylují, může být pořadí TOPk prvků častěji nepřesné kvůli aplikovanému vzorkování.

5.2 Kardinalita

Jednou ze základních statistik je kardinalita daného sloupce. Tato statistika je nejpřínosnější, pokud sloupec obsahuje malý počet hodnot, nebo naopak - je-li sloupec unikátní. Může se stát, že neplatí ani jedna z možností, a přesto může statistika přinášet užitečný pohled na data.

Aby analýza pracovala přesně, potřebuje uložit všechny unikátní prvky. Proto pracuje v prostoru $O(N)$. Z tohoto důvodu není snadné tuto statistiku spočítat pro velké datové soubory v paměti RAM počítače. Jednou z možností, jak se lze s tímto problémem vypořádat, je opět použití aproximačního přístupu. Práce obsahuje kromě přesné varianty také implementaci tří aproximačních algoritmů s omezenou pamětí: Bloom filtry, Linear counting a HyperLogLog.

Všechny implementované aproximační algoritmy jsou založeny na hešování a předem stanovené velikosti paměti. Při zachování konstantní chyby algoritmu je velikost paměti stále vázaná na velikost vstupu, ale oproti přesné analýze je

paměťová náročnost mnohem menší. Hlavní nevýhodou aproximativního přístupu je, že kvůli použitému hešování a možným kolizím nelze s jistotou určit, zda je sloupec unikátní.

5.2.1 Přesný algoritmus

Přesné řešení je implementováno pomocí datové struktury *java.util.HashSet*. Prvky jsou do této datové struktury postupně vkládány a kdykoliv je možné podat dotaz na počet unikátních hodnot. Nevýhodou přesného řešení je zmiňovaná paměť, protože ta roste lineárně s objemem unikátních hodnot na vstupu.

5.2.2 Bloom filtr

Prvním implementovaným aproximativním algoritmem je Bloom filtr (kód 5.8) (Bloom, 1970). Jedná se o strukturu, která dokáže odpovědět, zda se prvek x pravděpodobně nachází, nebo se určitě nenachází v množině M.

Algoritmus hešuje vstupní data o délce N k hešovacími funkcemi. Poté do bitového pole B ukládá informaci, zda se takový heš již vyskytl. Velikost pole B je závislá na délce výsledného heše. Délka heše použitých funkcí h je $\lceil \log_2(m) \rceil$.

Pokud alespoň jedna z hešovacích funkcí vrátí heš, který algoritmus ještě nemá uložen, jedná se jistě o nový, neznámý prvek. Jestliže však všechny heše algoritmus již viděl, nelze poznat, zda prvek je nový, nebo duplicitní. Mohla totiž nastat kolize všech hešů. Z toho plyne, že přesnost algoritmu roste s počtem hešovacích funkcí společně s jejich délkou. Naopak přesnost klesá se zaplněním bitového pole. Proto je potřeba předem odhadnout ideální velikost pole a počet použitých funkcí. Vhodné nastavení parametrů k, m je závislé na pravděpodobnosti ϵ chybné odpovědi při N prvcích (Broder a Mitzenmacher, 2004):

$$k = \frac{m}{N} \log 2 \text{ a } m = -\frac{N \log_2 \epsilon}{\log 2}. \tag{5.4}$$

V tabulce 5.3 je znázorněn růst paměti v závislosti na vstupních parametrech N a ϵ.

N	ϵ	Potřebná paměť	Počet hešovacích funkcí
10^6	10^{-5}	3MB	17
10^6	10^{-7}	4MB	23
10^8	10^{-5}	286MB	17
10^8	10^{-7}	400MB	23

Tabulka 5.3: Potřebná paměť pro Bloom filtry

Výsledný počet unikátních hodnot lze uchovávat pomocí počítadla, které se zvýší v případě nové unikátní vstupní hodnoty.

```
Vstup:
k: počet hešovacích funkcí
m: délka pole
N: prvky na vstupu

Algoritmus:
c <- počítadlo nastavené na 0
H <- množina k hešovacích funkcí h s velikostí heše log2(m)
B <- bitové pole o velikosti m se všemi nulovými bity

pro každé n z N
    if existuje h z H: Bh(n) == 0 then
        c <- c + 1
    end
    pro každé h z H
        Bh(n) <- 1
    end
end

Výstup:
V proměnné c je uložený odhad kardinality.
```

Kód 5.8: Algoritmus Bloom filtr s odhadem kardinality vstupních dat

5.2.3 Linear Counting

Čím více unikátních hodnot na vstupu hešovací funkce dostane, tím pravděpodobněji vzniknou kolize. Hlavní myšlenkou algoritmu Linear Counting (kód 5.9) (Whang a kol., 1990) je vzít v úvahu také počet kolizí hešů při průběhu výpočtu. Algoritmus pracuje s jedním bitovým polem B velikosti m a s jedinou hešovací funkcí h s délkou heše $\lceil \log_2 m \rceil$. Na indexu $h(x)$ v bitovém poli B je hodnota 1 právě tehdy, když vstup obsahuje prvek x. Výsledný odhad kardinality množiny je

$$-m \log(\frac{X}{m}), \tag{5.5}$$

kde X je počet bitů nastavených na 0 v poli B.

Předpoklad chování algoritmu na základě $|B|$ a $|N|$:

- $|N| \ll |B|$ - nejsou očekávané kolize, tudíž by algoritmus měl vrátit velmi dobrý odhad kardinality.
- $|B| \approx |N|$ - jsou očekávané kolize, tudíž prosté spočtení jedničkových bitů je nepřesné. Na základě obsazenosti pole B a pomocí rovnice 5.5 lze ale dostat lepší odhad.
- $|N| \gg |B|$ - pravděpodobně všechny bity jsou nastavené na 1 a počet kolizí je tak velký, že algoritmus nemůže vrátit přesný výsledek.

```
Vstup:
m: délka bitového pole
N: prvky na vstupu

Algoritmus:
h <- hešovací funkce s velikostí heše log2(m)
B <- bitové pole o velikosti m se všemi nulovými bity

pro každé n z N
    Bh(n) <- 1
end

Výstup:
X <- počet nulových bitů v poli B
Odhadovaný počet unikátních prvků: -m*log(X/m)
```

Kód 5.9: Algoritmus Linear Couting

5.2.4 LogLog

Popis posledního aproximativního přístupu k výpočtu kardinality začíná u algoritmu LogLog (Durand a Flajolet, 2003)(Havrlant, 2014b), který je nakonec vylepšen na variantu HyperLogLog (Flajolet a kol., 2007)(Havrlant, 2014a).

Hodnoty na vstupu si lze představit jako množinu H, ve které jsou veškeré prvky zahešované do binárních čísel o stejné délce. Počet binárních prvků končících na 0 v množině H je na základě pravděpodobnosti $0.5|H|$, počet hodnot končících na 00 je $0.25|H|$ atd. Jinými slovy platí, že počet prvků množiny H o velikosti N končících na m nul je

$$v = \frac{1}{2^{\mathrm{m}}} * N. \tag{5.6}$$

Po osamostatnění N lze rovnici přepsat do tvaru

$$2^m * v = N. \tag{5.7}$$

Za předpokladu, že pro právě jeden prvek končící na m nul je podle pravděpodobnosti potřeba množina o velikosti N, je dosazeno $v = 1$. Výsledná rovnice je zjednodušena do tvaru

$$2^m = N. \tag{5.8}$$

Na základě této myšlenky lze sestavit algoritmus (kód 5.2.5), který nalezne na vstupu maximální délku sufixu heše tvořeného pouze nulami. Pomocí této hodnoty a rovnice 5.8 může být odhadntý počet unikátních hodnot vstupní množiny.

Pokud by algoritmus ukládal pro celý vstup pouze jedinou maximální délku nulového sufixu, neprodukoval by uspokojivé výsledky. Proto je vhodné vstupní množinu rozdělit na 2^{k} disjunktních podmnožin a ukládat tuto délku daného suffixu pro každou podmnožinu zvlášť. V implementaci algoritmu se jedná konkrétně o pole počítadel M. Pro rozdělení vstupní množiny opět poslouží heš jednotlivých hodnot, který udává index v poli M.

Aby algoritmus nemusel v každém kroku pro určení příslušné podmnožiny a nulového sufixu vytvářet dva různé heše, je pro obě operace zároveň využit stejný heš. Prvních k bitů heše se použije pro určení indexu. Zbylých d bitů je ono binární číslo, ze kterého se určí velikost nulového sufixu. Na konci je již možné odhadnout výslednou kardinalitu jako

$$|N| \approx \alpha * 2^{\mathrm{m_{avg}}} * |M|, \tag{5.9}$$

kde m_{avg} je průměr maximálních délek nulových sufixů ve všech podmnožinách a $\alpha = 0.794$ je definovaná konstanta zpřesňující výpočet.

```
Vstup:
d: počet bitů heše pro určení velikosti sufixu nul
k: počet bitů indexu
N: prvky na vstupu

Algoritmus:
h <- hešovacích funkce s velikostí heše d+k
M <- pole 2^k počítadel
p(y) <- funkce, která určí počet nul na konci binárního čísla y

pro každé n z N
    <b_1, b_2, ... b_{d+k}>_2 <- h(n)
    M[<b_1, ... b_d>_2] <- max(M[<b_1, ... b_d>_2], p(<b_{d+1}, ... b_{d+k}>_2))
end

Výstup:
0.794 * 2^avg(M) * |M|
```

Kód 5.10: Algoritmus LogLog

5.2.5 HyperLogLog

Předešlý aproximativní algoritmus LogLog lze pro získání vyšší přesnosti odhadu vylepšit na dvou místech. Výsledný algoritmus (kód 5.11) se pak nazývá HyperLogLog.

První nedostatek algoritmu je následující: při výpočtu kardinality a použití aritmetického průměru může být výsledná hodnota zkreslena jedním extrémním prvkem v poli počítadel M. Proto je aritmetický průměr hodnot nahrazen harmonickým průměrem. Navíc se nejedná o průměr hodnot m z pole M, ale o průměr 2^m. Díky tomuto postupu je zvýšena přesnost odhadu. Výsledná rovnice je upravena do podoby

$$|N| \approx \alpha * H * |M|, \tag{5.10}$$

kde H je geometrický průměr počítadel v poli M. Hodnota konstanty α je oproti předešlému algoritmu závislá na velikosti pole M. Výpočet této konstanty α je znázorněn v tabulce 5.4.

$\|M\|$	α
16	0,673
32	0,697
64	0,709
>64	$\frac{0,7213}{(1+\frac{1,079}{\|M\|})}$

Tabulka 5.4: Výpočet konstanty α v algoritmu HyperLogLog

Dále se ukázalo, že algoritmus má špatné výsledky pro malé kardinality. Aby bylo možné dostávat i v tomto případě uspokojivé výsledky, nelze použít pouze rovnici 5.10. Ale pokud je odhad pomocí této rovnice menší než 2,5$|M|$ a existují nulová počítadla v M, provede se výpočet pomocí algoritmu Linear Counting na poli M. Nulová počítadla jsou převedena na nulové bity. Zbylá počítadla tvoří bity s hodnotou jedna. Algoritmus Linear Counting nakonec vrátí odhad, který by měl být pro malé kardinality přesnější než pouze upravený algoritmus LogLog s harmonickým průměrem.

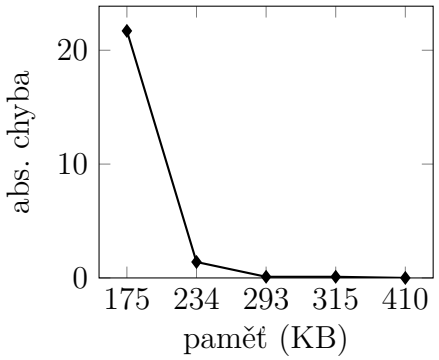
```
Vstup:
d: počet bitů heše pro určení počtu nul
k: počet bitů indexu
N: prvky na vstupu

Algoritmus:
h <- hašovací funkce s velikostí heše d+k
M <- množina 2^k počítadel
p(y) <- funkce, která určí počet nul na konci binárního čísla y

pro každé n z N
    <b_1, b_2, ... b_d+k>_2 <- h(n)
    M[<b_1, ... b_d>_2] <- max(M[<b_1, ... b_d>_2], p(<b_d+1, ... b_d+k>_2))
end

Výstup:
Pro kardinality větší než 2,5|M|: alpha * avg_harm(2^m z M) * |M|
Pro malé kardinality: linearCounting(M)
```

Kód 5.11: Algoritmus HyperLogLog

5.2.6 Analýza

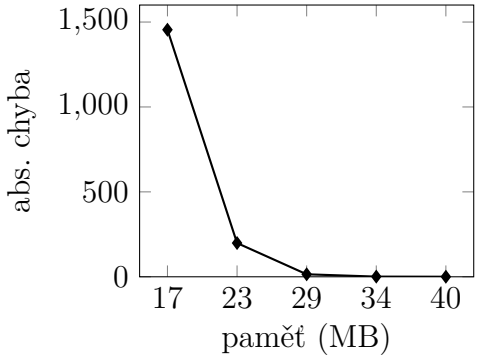
Hlavním cílem měření, popsaných v této sekci, je nalézt ideální aproximativní algoritmus pro výpočet kardinality v datovém profilování s omezenou pamětí. Jsou zde srovnány tři varianty: Bloom filtr (obrázky 5.3 a 5.4), Linear counting (obrázky 5.5 a 5.6) a HyperLogLog (obrázky 5.7 a 5.8). Pro zmíněné varianty byly použity vstupy o délce 100 tisíců a 10 miliónů unikátních vstupních hodnot. Měření se na základě omezení velikosti interní paměti zabývá přesností algoritmů, která je vyjádřená jako průměrná absolutní chyba výpočtu.

ϵ	m	k	paměť (KB)
10^{-3}	1 437 759	10	175
10^{-4}	1 917 012	13	234
10^{-5}	2 396 265	17	293
10^{-6}	2 875 518	20	315
10^{-7}	3 354 771	23	410



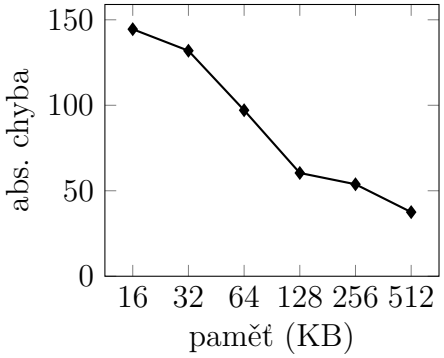
Obrázek 5.3: Bloom filtr: 100 tisíc unikátních hodnot (m = velikost bitového pole, k = počet hešovacích funkcí, ϵ = pravděpodobnost chybné odpovědi)

ϵ	m	k	paměť (MB)
10^{-3}	143 775 876	10	17
10^{-4}	191 701 168	13	23
10^{-5}	239 626 460	17	29
10^{-6}	287 551 752	20	34
10^{-7}	335 477 044	23	40



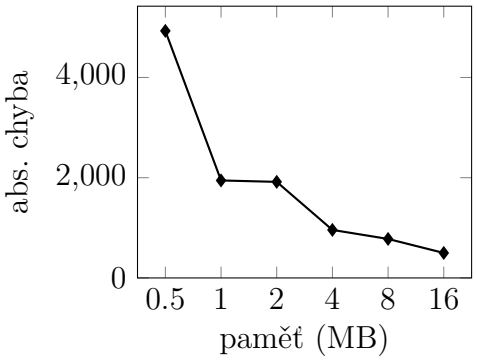
Obrázek 5.4: Bloom filtr: 10 miliónů unikátních hodnot (m = velikost bitového pole, k = počet hešovacích funkcí, ϵ = pravděpodobnost chybné odpovědi)

m	paměť (KB)
2^{17}	16
2^{18}	32
2^{19}	64
2^{20}	128
2^{21}	256
2^{22}	512



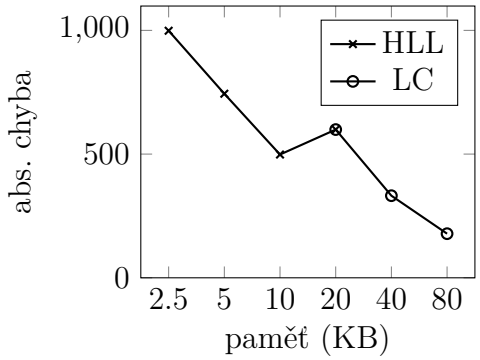
Obrázek 5.5: Linear counting: 100 tisíc unikátních hodnot (m = velikost bitového pole)

m	paměť (MB)
2^{22}	0,5
2^{23}	1
2^{24}	2
2^{25}	4
2^{26}	8
2^{27}	16



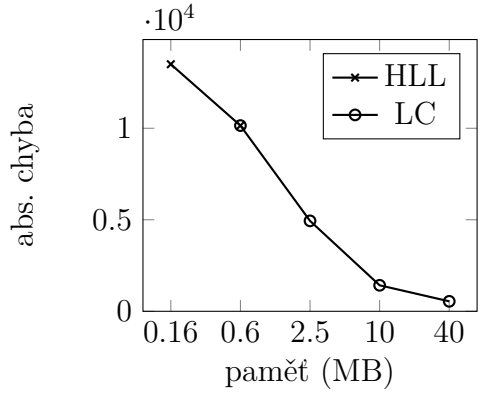
Obrázek 5.6: Linear counting: 10 miliónů unikátních hodnot (m = velikost bitového pole)

k	d	$2^k * \lceil \log_2 d \rceil$	paměť (KB)
12	17	20 480	2,5
13	17	40 960	5
14	17	81 920	10
15	17	163 840	20
16	16	327 680	40
17	15	655 360	80



Obrázek 5.7: HyperLogLog: 100 tisíc unikátních hodnot (k = délka indexu, d = délka heše)

k	d	$2^k * \lceil \log_2 d \rceil$	paměť (MB)
18	31	1 310 720	0,16
20	31	5 242 880	0,6
22	31	20 971 520	2,5
24	31	83 886 080	10
26	31	335 544 320	40



Obrázek 5.8: HyperLogLog: 10 miliónů unikátních hodnot (k = délka indexu, d = délka heše)

Na základě měření potřebuje algoritmus HyperLogLog velmi malou paměť pro výslednou chybu okolo jednoho procenta. Pokud je však potřeba zvýšit přesnost a je možné k tomu využít větší paměť pro stejný počet záznamů, tento algoritmus již není vhodnou variantou. Zmíněná vlastnost plyne přímo z důvodů pro

rozšíření algoritmu LogLog o Linear counting. Kvůli podmínce zmíněné v sekci 5.2.5 se přepne algoritmus na variantu upraveného algoritmu Linear counting. Ta produkuje lepší odhady pro případy s menším poměrem počtu unikátních hodnot a velikosti paměti. Algoritmu HyperLogLog není potřeba přidávat více paměti. Rovnou lze použít přímo algoritmus Linear counting. Dle měření a předchozího pozorování se tak jeví algoritmus Linear counting jako vhodná varianta pro určení kardinality v datovém profilování s omezenou pamětí.

Algoritmus Bloom filtr je vhodný pro určení unikátnosti sloupce menší velikosti - do statisíců unikátních záznamů. Pro získání srovnatelné přesnosti je potřeba mnohem větší paměť než u zbylých variant. Výhodou však je, že z důvodu interní implementace není vydávána větší kardinalita, než je skutečná hodnota. Proto v případě odhadu kardinality rovné počtu vstupních hodnot lze odpovědět, že jsou všechny prvky unikátní.

Algoritmus HyperLogLog lze použít pro extrémně velké tabulky - od stovek miliónů nebo jednotek miliard prvků. V tomto případě algoritmus Linear counting bude již potřebovat pro podobnou přesnost tak velké pole počítadel, že se uživatel musí spokojit s větší nepřesností z důvodu omezení paměti.

Odhady kardinality na základě vzorku lze učinit, ale jedná se většinou o analýzu více vzorků najednou (Wu, 2012). Vzorkování a jednoprůchodovou analýzu nelze pro odhad kardinality použít bez znalosti dalších vlastností vstupních dat (například bez znalosti pravděpodobnostního rozdělení dat). Na druhou stranu i jednoprůchodová varianta se vzorkováním může být někdy užitečná. Lze detekovat podezření na unikátní sloupce (ve vzorku dat jsou pouze unikátní hodnoty) nebo sloupce obsahující pouze výčet hodnot (ve vzorku dat je velmi malý počet unikátních hodnot).

5.3 Histogram hodnot

Histogram znázorňuje rozložení dat pomocí sloupcového grafu. Implementovaná varianta je omezena na číselné hodnoty, tedy každý sloupec vyjadřuje četnost hodnot v daném intervalu. Výsledkem analýzy je pak k intervalů a jejich frekvence v analyzovaných datech. Všechny intervaly jsou zprava uzavřené kromě prvního, který je uzavřený z obou stran.

Histogramy lze rozdělit do několika kategorií. *Histogramy s rovnoměrnou šířkou* obsahují sloupce představující stejně široké intervaly. *Histogramy s rovnoměrnou hloubkou* mají sloupce s libovolně širokým intervalem, ale všechny představují stejný počet hodnot. V případě některých aproximačních algoritmů je možné se setkat s histogramy, které nemají ani jednu ze zmíněných vlastností. *Histogram s rovnoměrnou hloubkou* lze vytvořit pomocí algoritmu pro výpočet kvantilů, protože právě kvantily rozdělují setříděná vstupní data na k stejně početných celků.

Implementace se však zabývá pouze *histogramy s rovnoměrnou šířkou*. Analýza na vstupu dostává proud číselných hodnot a parametr k, který určuje výsledný počet stejně velkých intervalů.

5.3.1 Přesný algoritmus

Přesný výpočet histogramu nad proudem dat spočívá ve spočtení úplné frekvenční analýzy (nestačí však pouze k nejčastějších prvků jako v sekci 5.1) a ná-

sledném vytvoření histogramu. Jakmile skončí frekvenční analýza, je třeba zjistit pouze velikost intervalu mezi minimálním a maximálním prvkem a tento interval rozdělit na k rovnoměrných intervalů. Frekvence intervalu je poté součtem frekvencí klíčů z frekvenční analýzy patřících do daného dílčího rozsahu.

5.3.2 Výpočet s omezenou pamětí

Pro výpočet histogramů s omezenou pamětí není možné nejprve uložit všechny hodnoty do paměti a na základě hodnot vytvořit intervaly a spočítat jejich frekvence. Proto je hlavním cílem aproximativního přístupu buď odhadnout pevné výsledné intervaly na začátku výpočtu, nebo upravovat intervaly průběžně. Několik variant algoritmů (Shiu, 2014), které vypočítávají histogramy nad proudem dat:

- **Pevné intervaly** - Nejprve se spočítá frekvenční analýza pro prvních m různých hodnot. Následně se vytvoří rovnoměrné intervaly pro výsledný histogram. Poté probíhá pouze zvyšování frekvence těchto intervalů.
- **Proměnná šířka intervalů** - Na začátku se zvolí m různých intervalů. V průběhu analýzy se při zvyšování frekvencí spojují nebo rozdělují vedlejší intervaly podle stanoveného pravidla. Tím se udržuje jejich celkový počet v určité mezi. Před vrácením výsledných intervalů dochází k transformaci interních intervalů na výsledný počet k intervalů.
- **Logaritmické intervaly** - Na začátku se vytvoří intervaly s logaritmicky se zvětšující šířkou. V uvedeném příkladě 5.5 má prvních 100 intervalů šířku 0,001 a dohromady pokrývají interval od 0 do 0,1. Následujících 90 intervalů má velikost 0,01, dohromady pokrývají rozsah 0,1 až 1 atd. V průběhu výpočtu probíhá zvyšování frekvencí intervalů. Před vrácením výstupu dochází ke sloučení intervalů tak, aby jejich výsledný počet byl k a aby byla pokrytá celá vstupní množina dat.

Začátek intervalu	Konec intervalu	Velikost přírůstku
0	0,1	0,001
0,1	1	0,01
1	10	0,1
10	100	1
100	1000	10
atd.		

Tabulka 5.5: Varianta s intervaly logaritmické velikosti

Vytvořená implementace (tabulka 5.6) vychází z varianty *logaritmické intervaly* s tím rozdílem, že intervaly nejsou po sobě jdoucí, ale jsou rozděleny do úrovní. Prostřední interval každé úrovně začíná na stejném místě. Pro každý interval i na dané úrovni u platí, že jeho velikost je $|i| = 10^u$. Počet intervalů jedné úrovně je konfigurovatelný stejně jako počet všech úrovní.

Prostřední interval nezačíná vždy na hodnotě 0. Algoritmus nejprve počítá frekvenční analýzu prvních m hodnot. Při $m + 1$-ní hodnotě se vypočte medián klíčů, který určí střed pro konstrukci všech úrovní intervalů. Tento střed je ještě zarovnán na příslušný řád úrovně (z hodnoty 143 není vytvořen interval 143-153, ale interval 140-150).

Úroveň	Začátek	Konec	Velikost vnitřního intervalu
-3	-0,1	0,1	0,001
-2	-1	1	0,01
-1	-10	10	0,1
0	-100	100	1
1	-1000	1000	10
atd.			

Tabulka 5.6: Implementovaná varianta se středem v 0

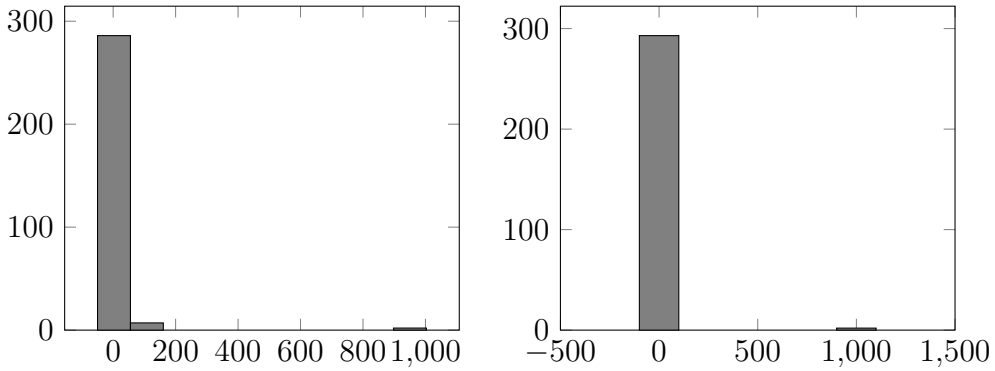
Jakmile jsou intervaly připraveny, je frekvence každé vstupní hodnoty zaznamenána do histogramů na všech úrovních. Pokud se hodnota nachází mimo rozsah intervalů na některé z úrovní, pouze se zaznamená, zda je hodnota větší, nebo menší než daný interval. Počet nezaznamenaných hodnot je vrácen společně s výsledným histogramem.

Výpis výsledků algoritmu je konfigurovatelný parametry k - počet výsledných intervalů a p - minimální procento hodnot ve výsledném histogramu oproti velikosti vstupu. Při vypsání výsledků se vybere ve vzestupném pořadí první úroveň s intervaly taková, aby součet frekvencí uložených uvnitř intervalů byl alespoň pN. Pro vybranou úroveň intervalů následně proběhne transformace na k výstupních intervalů. Krajní intervaly jsou poté zmenšeny tak, aby celý histogram nepokrýval v součtu větší rozsah, než je mezi minimem a maximem.

5.3.3 Analýza

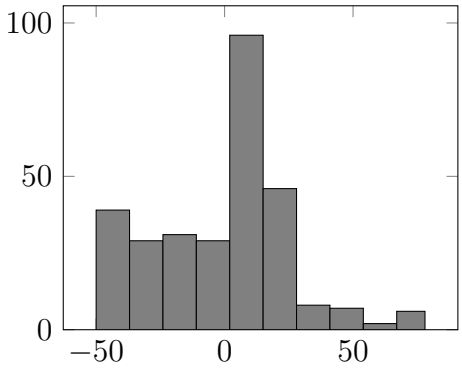
Důležitou součástí aproximativního přístupu ze sekce 5.3.2 je správné nastavení konstanty p. Pro lepší demonstraci vlivu této konstanty na výsledný histogram byla provedena analýza dat s výskytem několika anomálních hodnot (obrázek 5.9).

U aproximativního přístupu s $p = 1$ (obrázek 5.9b) lze vidět, že bylo potřeba použít větší velikost intervalů než při přesném přístupu (obrázek 5.9a). Na výstupu jsou tudíž pouze dva intervaly s nenulovou frekvencí. V případě přesného výpočtu jsou výsledné velikosti intervalů vytvořeny až na základě rozdílu maxima a minima. Z tohoto důvodu je časté, že výstup přesné analýzy obsahuje více intervalů, a tím je histogram detailnější. Třetí varianta (obrázek 5.9c) znázorňuje vliv konstanty p na výsledný histogram. Pro měření bylo použito $p = 0{,}9$. V tomto histogramu nelze vyčíst, že jsou některé hodnoty anomální. Na druhou stranu jsou však data rozprostřená do deseti dílčích intervalů a výsledný histogram lépe popisuje více než 90 % hodnot.



(a) Přesný výpočet

(b) Aproximativní výpočet $p = 1$



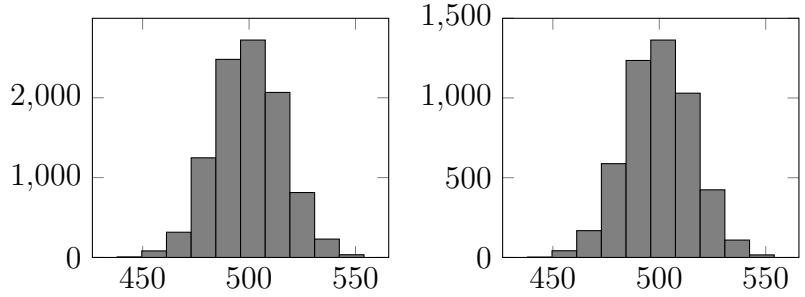
(c) Aproximativní výpočet $p = 0{,}9$

Obrázek 5.9: Histogram: vstup s anomáliemi

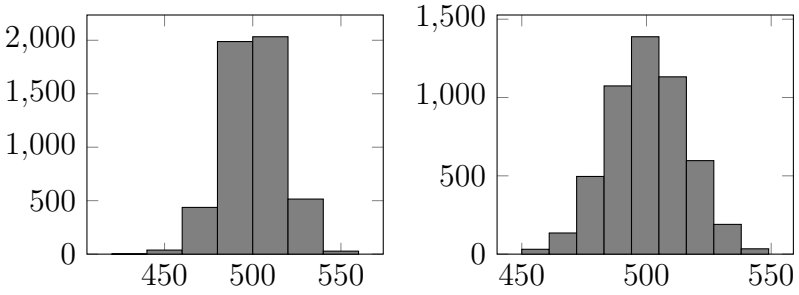
Díky tomuto měření byla implementace rozšířená o vydávání více histogramů pro různé hodnoty konstanty p. Výpočet stačí spustit jen jednou a je na uživateli, jak detailní informaci chce na výstupu. V uvedeném příkladu (obrázek 5.3) by pro uživatele mohla být nejlepší kombinace parametrů $p = 1$ a $p = 0{,}9$. Zmíněné vylepšení histogramu lze samozřejmě aplikovat také na přesný algoritmus. Zde se však obecně očekává, že algoritmus bude produkovat dobré výsledky i pro samotnou variantu $p = 1$.

Další vlastnosti analýzy lze demonstrovat na příkladu binomického rozdělení. Výsledky obsahují kromě několika variant hodnot konstanty p také aplikaci vzorkování. První sada měření proběhla na vzorku o velikosti 50 % (obrázky 5.10) a druhá pouze na 10 % (obrázky 5.11) řádků. V obou případech lze vidět, že přesná i aproximativní analýza produkují v případě vzorkování velmi použitelné výsledky.

Na základě měření lze usoudit, že zvolený aproximativní přístup s parametrem p je vhodný pro profilování dat s omezenou pamětí. Testování proběhlo s parametry 15 úrovní a 100 intervalů na každé úrovni. Pokryjí se tak intervaly o celkové velikosti od 10^{-4} do 10^{10}. Pro jednu zmíněnou úroveň je potřeba přibližně 0,5 KB. Celková paměť pak odpovídá přibližně pouhým 8 KB pro jeden sloupec, na kterém je tato analýza aplikovaná.

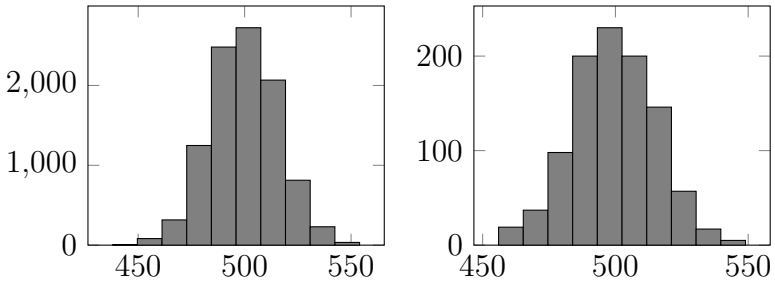


(a) Přesná analýza (b) Přesná analýza, vzorek 50%

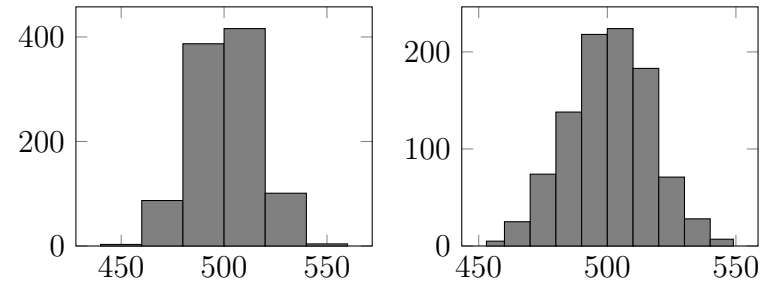


(c) Aprox., vzorek 50%, $p = 1{,}0$ (d) Aprox., vzorek 50%, $p = 0{,}9$

Obrázek 5.10: Histogram: vzorkování 50%



(a) Přesná analýza (b) Přesná analýza, vzorek 10%



(c) Aprox., vzorek 10%, $p = 1{,}0$ (d) Aprox., vzorek 10%, $p = 0{,}9$

Obrázek 5.11: Histogram: vzorkování 10%

5.4 Kvantily

Kvantily jsou hodnoty, které rozdělují setříděný datový soubor na stejně velké části. Nejznámějším kvantilem je medián, který rozděluje soubor na poloviny.

Jedná se tedy o hodnotu s prostředním indexem ze setříděného vstupu. Přesněji ϕ-kvantil ($\phi \in [0,1]$) je na indexu $\lceil \phi N \rceil$. Mezi další hojně používané kvantily patří kvartily (rozdělení 1/4), decily (1/10) a percentily (1/100). Mimo použití kvantilů pro statistickou analýzu dat je časté užití této analýzy pro optimalizaci databázových dotazů. Databázové systémy často implementují také aproximativní variantu kvantilů[3].

Implementace analýzy mají konfigurovatelný počet kvantilů na výstupu. Použitá varianta pro porovnání je s parametrem 10. Výsledné hodnoty tak rozdělují datový soubor na decily. Analýza je aplikovatelná pro libovolný datový typ, který lze setřídit. V aplikaci tento předpoklad splňují všechny datové typy. Jen pro datový typ `BOOLEAN` není tato informace užitečná, protože zcela popisující je v tomto případě frekvenční analýza.

Z popisu kvantilů lze vyčíst, že se jedná o paměťově náročnou statistiku kvůli potřebě setřídit všechna data. Implementovaný aproximativní přístup pracuje s předem definovaným omezením paměti na úkor přesnosti výpočtu.

5.4.1 Přesný algoritmus

Jedním ze způsobů, jak spočítat přesným výpočtem kvantily, je prosté setřídění vstupních dat a následné vrácení hodnot na odpovídajících $(i * N/k)$-tých indexech. Algoritmus tak funguje v časové složitosti $O(Nlog(N))$, ale potřebuje celý vstup v paměti.

Existují také algoritmy pro vyhledávání k-tého nejmenšího prvku s lepší očekávanou časovou složitostí než při naivní implementaci. Pro k-tý prvek se často používá algoritmus Quickselect, který má očekávanou časovou složitost lepší než prosté setřídění, ale v nejhorším případě je kvadratická. Pro snížení časové náročnosti lze použít aproximativní algoritmy, které však pracují nad celým vstupem v paměti. Tato práce je ale zaměřená především na omezení potřebné paměti během výpočtu, a z tohoto důvodu práce neobsahuje tento typ algoritmů.

5.4.2 MRL98

Pro výpočet aproximativních kvantilů (Chen a Zhang, 2020) byl zvolen algoritmus MRL98 (Manku a kol., 1998). Hlavní výhodou tohoto algoritmu je možnost omezit paměť, a tím zamezit potřebě uložit všechny vstupní hodnoty do paměti. ϵ-aproximativní ϕ-kvantil je libovolný element z intervalu $\lceil (\phi - \epsilon)N \rceil$ až $\lceil (\phi + \epsilon)N \rceil$.

Algoritmus pracuje s b buffery (jednorozměrné pole prvků) o velikosti d prvků a ke každému bufferu B je přiřazena jeho váha $w(B)$. Algoritmus se poté skládá ze tří základních operací:

- **`NEW`** - Na vstupu dostává prázdný buffer a proud nejvýše d následujících hodnot ze vstupu. Na výstupu produkuje plný buffer s váhou 1. Pokud po přečtení celého vstupu není poslední buffer zcela zaplněn, lze přidat na volná místa rovnoměrně $+nekonečno$ a $-nekonečno$.

[3]Oracle: https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/ query-optimizer-concepts.html#GUID-DB2B2D98-B47E-4D44-8F94-E44DCFAEF305

- **COLLAPSE** - Na vstupu dostává m bufferů a na výstupu vrátí jeden buffer vytvořený ze vstupu. Fungování této operace je znázorněno v kódu 5.12.

  Výsledná váha W je součtem m vstupních vah bufferů. Prvky výsledného bufferu jsou získány tak, že funkce vytvoří jednu setříděnou posloupnost hodnot ze vstupních bufferů. Zde jsou prvky duplikovány tolikrát, kolik je váha příslušného bufferu. Prvky výsledného bufferu jsou vybrány pro liché W z indexů $j * W + (W + 1)/2$ a pro W sudé z indexů $j * W + W/2$ nebo $j * W + (W + 2)/2$.

- **OUTPUT** - Funkce dostane na vstupu všechny buffery a číslo ϕ definující, jaký kvantil má být na výsledku. Funkce poté funguje obdobně jako COLLAPSE se stejným principem vážení. Využívá však všechny buffery najednou a na výstupu je pouze ϕN-tý prvek.

```
Vstup:
b1: 12, 52, 72, 102, 132
w(b1): 2
b2: 23, 33, 83, 143, 153
w(b2): 3
b3: 44, 64, 94, 114, 124
w(b3): 4

Spojení do jedné posloupnosti:
 12  12  23  23  [23]  33  33  33  44
 44  44  44  52  [52]  64  64  64  64
 72  72  83  83  [83]  94  94  94  94
102 102 114 114 [114] 114 124 124 124
124 132 132 143 [143] 143 153 153 153

Výpočet váhy:
w(b1) + w(b2) + (b3) = 9

Výstup:
Buffer s prvky na indexech j*9+5: 23, 52, 83, 114, 143
Váha výstupu: 9
```

Kód 5.12: Funkce COLLAPSE (Manku a kol., 1998)

Nakonec je ještě potřeba implementovat strategii, kdy budou probíhat funkce COLLAPSE a NEW. Zvolená strategie přiděluje každému bufferu B navíc jeho úroveň $l(B)$. Pokud zbývá pouze poslední volný buffer, proběhne na tomto bufferu funkce NEW a bude mu přidělena minimální hodnota z existujících úrovní. Pokud zbývá více než jeden, pak pro každý takový buffer proběhne funkce NEW a přiřadí se mu úroveň 0. Pokud již žádný volný buffer nezbývá, proběhne funkce COLLAPSE na všech množinách bufferů se stejnou úrovní. Výsledná úroveň se poté zvýší o jedna.

Výběr vhodných hodnot b (počet bufferů), d (délka bufferu), a tím definování velikosti paměti, není přímočarý. Přesnější popis lze najít v (Manku a kol., 1998).

Pro příklad jsou znázorněné hodnoty parametrů v 5.7. Výsledný potřebný prostor závisející na velikosti chyby ϵ a velikosti vstupu N je $O(\frac{1}{\epsilon}\log^2 \epsilon N)$ uložených prvků v paměti.

	Hodnota b			**Hodnota** d			**Paměť**		
ϵ, N	10^5	10^7	10^9	10^5	10^7	10^9	10^5	10^7	10^9
0,1	5	10	12	55	60	77	0,3 K	0,6 K	0,9 K
0,01	7	9	10	217	412	765	1,5 K	3,7 K	7,7 K
0,001	3	5	10	2778	5495	5954	8,3 K	27,5 K	59,5 K

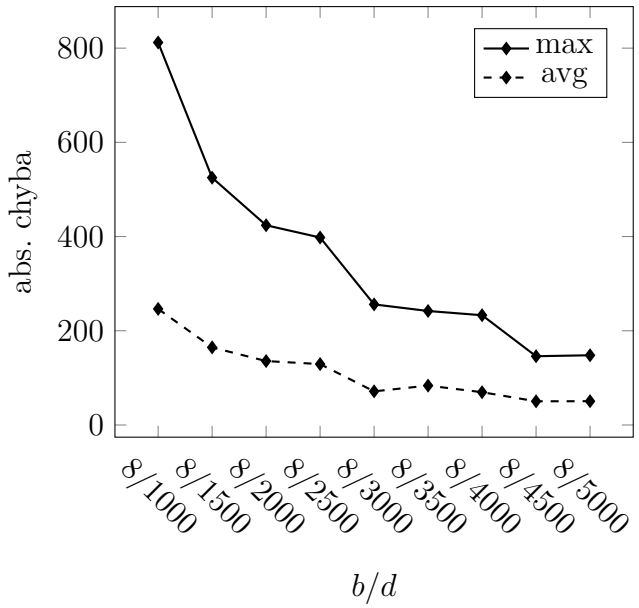
Tabulka 5.7: Volba parametrů b a d na základě velikosti vstupních dat (Manku a kol., 1998)

5.4.3 Analýza

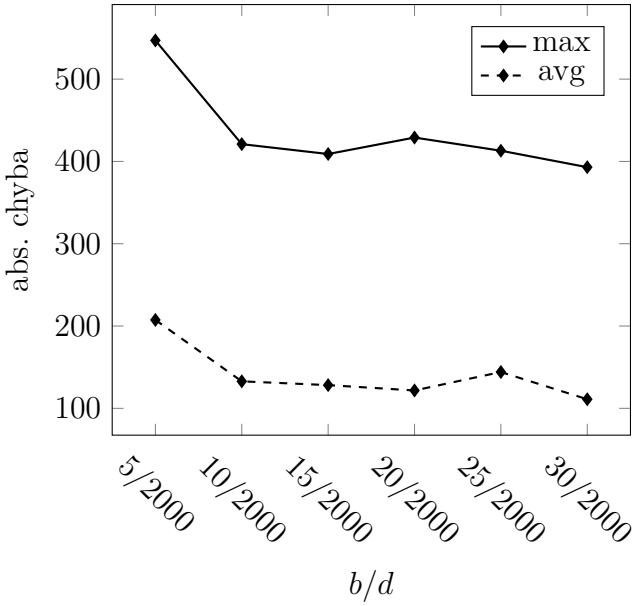
Měření se zabývá algoritmem MR98 a jeho různým nastavením parametrů v porovnání s přesnými výsledky. Měřené hodnoty představují maxima a průměry absolutních chyb výsledného aproximativního kvantilu. Některé výsledky však uvádějí relativní chybu, která je definována jako poměr absolutní chyby aproximativního kvantilu a velikosti intervalu mezi dvěma přesnými kvantily. Všechna testování probíhala na decilech ($k = 10$).

Během prvního měření (obrázek 5.12) byl zafixován počet bufferů na $b = 8$ a byla pouze inkrementována velikost daného bufferu d. Vygenerovaná testovací data obsahovala 1 milión řádků. Chyba výpočtu klesala nejvíce mezi hodnotami $d = 1000$ až $d = 3000$. Poté se již chyba zmenšovala velmi málo v porovnání s růstem paměti. Během měření byla velikost potřebné paměti od 8000 do 40000 prvků. Při zmiňovaném nastavení parametru $d = 3000$ odpovídala maximální chyba hodnotě 256 (0.26 %) a průměrná 71 (0.07 %). Maximální počet prvků v paměti byl v tomto případě omezen na 24000.

V případě zafixování délky bufferu (obrázek 5.13) na $d = 2000$ s různými hodnotami b se ukázalo, že počet bufferů tolik nezvyšuje přesnost výpočtu při stejném objemu dat. Od hodnoty $b = 10$ se již maximální chyba výpočtu pohybovala okolo 400 a průměrná chyba byla v intervalu 110-130.



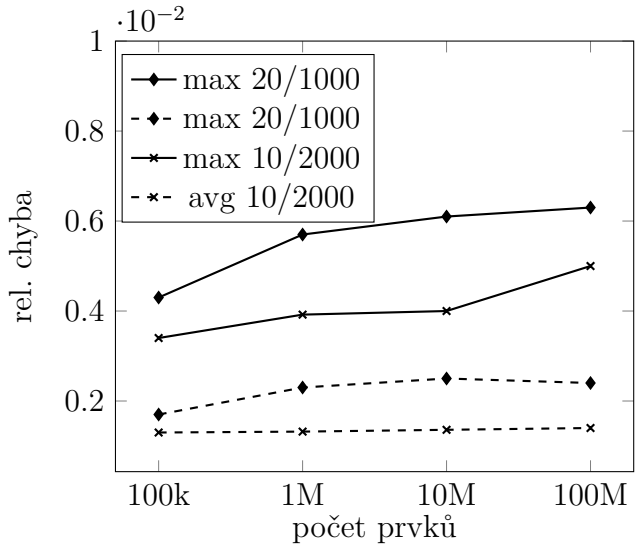
Obrázek 5.12: Výpočet kvantilů na vstupu o délce 1 milión unikátních řádků s parametrem $b = 8$ a různou velikostí parametru d



Obrázek 5.13: Výpočet kvantilů na vstupu o délce 1 milión unikátních řádků s parametrem $d = 2000$ a různou velikostí parametru b

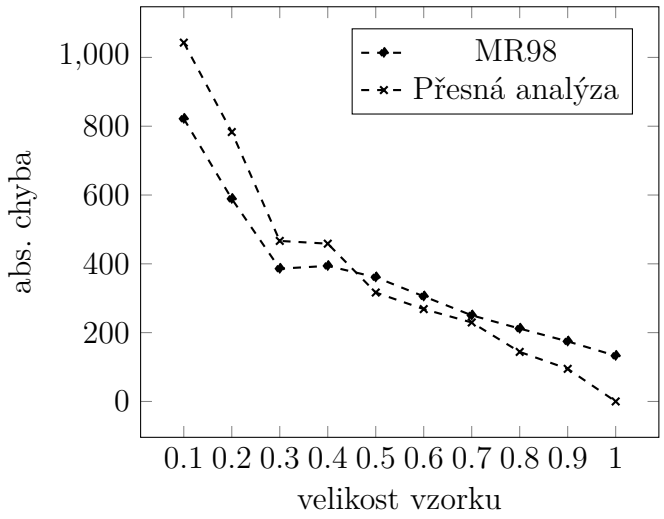
Ne vždy je však předem jasný počet vstupních hodnot. Proto proběhlo také testování (5.14), které se zaměřovalo na pokles přesnosti aproximativního algoritmu pro různě velké vstupy. Parametry byly ale vždy stejné. Na základě předchozích měření bylo použito omezení na 20000 prvků v paměti. Konkrétně se jednalo o kombinace parametrů $b = 10$, $d = 2000$ a $b = 20$, $d = 1000$. Zmíněné nastavení se následně použilo na měření se vstupy o délce sto tisíc až sto miliónů unikátních řádků. Výsledky ukázaly, že omezením maximálního počtu prvků v paměti na 20.000 lze dosáhnout až překvapivě velmi slušných výsledků i pro

velká data v řádech miliónů řádků. Při 100 miliónech unikátních hodnot se jedná o 5000násobnou úsporu paměti při ztrátě přesnosti v řádu procent.



Obrázek 5.14: Aproximativní výpočet kvantilů s omezenou pamětí na 20000 prvků

Poslední měření (obrázek 5.15) obsahuje variantu se vzorkováním. Zde je možné vidět zajímavý fakt: průměrná chyba způsobená vzorkováním není již příliš ovlivněná aplikováním aproximativního algoritmu. Jedná se tak o vhodnou kombinaci pro ušetření paměti i času zároveň.



Obrázek 5.15: Průměrná chyba výpočtu kvantilů se vzorkováním 10 milionů záznamů

5.5 Klasifikace dat

Cílem analýzy je na základě dat přiřadit značky ke sloupcům dle definovaných pravidel. Výchozí chování je následující: pravidlo musí vyhovovat pro všechny hodnoty daného sloupce, aby proběhlo přiřazení značky. Uživatel má ale možnost nastavit, kolik procent vyhovujících hodnot stačí, aby přiřazení proběhlo.

Výsledkem je snaha o klasifikaci sloupců, které nemají ještě dostatečnou kvalitu dat, popř. jsou pravidla definována ve zjednodušené formě.

Značka	Email
Pravidlo	Regulární výraz: `[^@\s]+@([^@\s\.]+\.)+[^@\.\s]+`
Datový typ	STRING
Mez pro přiřazení	95%

Tabulka 5.8: Ukázka značky *Email* s pravidlem pro automatické přiřazení

Analýza nejprve načte pravidla definovaná uživatelem (tabulka 5.8). Poté se pokusí aplikovat pravidlo na všechny vstupní hodnoty. Pokud je počet kladných výsledků dostatečný, proběhne přiřazení značky na daný sloupec.

Implementaci lze rozdělit na dvě části: jádro, které se stará o vyhodnocování výrazů, a logiku přiřazující značky na základě vyhodnocení výrazů. Je však potřeba zvolit gramatiku, pomocí které se budou pravidla zapisovat, a také způsob, jakým bude probíhat jejich vyhodnocování. Napsání vlastního robustního jazyka a jádra pro vyhodnocování, které pracuje v rozumné rychlosti, je složitý úkol. Existuje však několik hotových řešení, která stačí použít, a nad zmíněné jádro dodělat vrstvu pro samotnou klasifikaci dat.

Do užšího výběru byly zahrnuty dvě možné implementace jádra: JShell[4] a SpEL[5]. Existuje i mnoho dalších variant: JSP expression language[6], MVEL[7] nebo JEXL[8].

JShell V JDK[9] 9 byla přidána možnost interaktivní práce v Javě pomocí nástroje JShell. Uživatel má možnost psát výrazy, které jsou rovnou vyhodnocované v konzoli. Pro účely zápisu a vyhodnocování výrazů je tento nástroj vhodný, jelikož lze zapsat libovolný Java kód. Uživatel tudíž není omezen rozsahem použitého jazyka. Na druhou stranu není snadné zapisovat tyto výrazy v Javě bez znalostí programování. Obdobně lze například použít také syntaxi jazyka JavaScript.

Zásadním problémem zmíněné varianty je nízká rychlost a velké nároky na paměť. Interně jsou ukládány všechny vyhodnocené výrazy, a tudíž se nejedná o ideální nástroj pracující s omezenou pamětí. Také zmíněná rychlost je podstatně horší než u vybraného řešení.

Spring expression language Nakonec vybraným řešením je Spring Expression Language (SpEL). Jedná se o součást frameworku Spring[10], který přináší mnoho usnadnění při programování rozsáhlých aplikací. Kromě zásadního využití Dependency Injection dodává při práci s databází abstrakci, zjednodušuje tvorbu API rozhraní nebo řeší zabezpečení aplikací. Pro většinu částí frameworku je společné, že se výrazy často zapisují pomocí jazyka SpEL. Programátor má

[4]https://docs.oracle.com/javase/10/jshell/introduction-jshell.htm

[5]https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/expressions.html

[6]https://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html

[7]http://mvel.documentnode.com [8]http://commons.apache.org/proper/commons-jexl/

[9]Java Development Kit [10]https://spring.io

například možnost psát výrazy v anotacích *@Value*[11]. Framework poté dosadí výsledek vyhodnoceného výrazu do proměnné.

Celý SpEL je dostupný jako knihovna bez nutnosti závislostí na jiných komponentách z frameworku Spring. Proto jej lze samostatně použít pro implementaci analýzy. Ukázalo se, že tato varianta je mnohem rychlejší než JShell. Při výpočtu navíc neroste interní paměť. Jazyk má vlastní syntaxi, souběžně lze ale využívat také konstrukce z jazyka Java. V neposlední řadě je zde několik funkcionalit, které poskytují uživateli snadnější zápis výrazů než ve zmiňovaném jazyce Java. Jedná se například o automatickou typovou konverzi nebo elvis operátor[12]. Zapsané pravidlo (tabulka 5.9) je libovolný výraz podle specifikace jazyka SpEL se dvěma omezeními. Vstupní hodnota je předávaná pomocí proměnné `value`, která má typ na základě definice pravidla. Druhým omezením pro výrazy je, že návratový typ musí být `Bolean`.

Značka	Výraz
Email	`#value matches '[^@\s]+@([^@\s\.]+\.)+[^@\.\s]+'`
UUID	`#value matches '[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}'`
Rodné číslo	`(#value matches '[0-9]{6}/[0-9]{3,4}') && (T(java.lang.Integer).parseInt(#value.replace('/',''')) % 11 == 0)`
CZ tel. číslo	`#value.replace(' ', '') matches '\+420[0-9]{9}'`

Tabulka 5.9: Ukázka definice tagů pomocí jazyka SpEL

5.6 Masky a vzory dat

Jednou z cest, jak porozumět rozsáhlým datům, je rozpoznat vnitřní strukturu hodnot. Textová data často tvoří popsatelnou strukturu. Díky tomu může uživatel snadněji porozumět datovému souboru. Pomocí této analýzy je také možné kontrolovat kvalitu dat. Například pokud víme, že hodnoty musí obsahovat dvě slova oddělená pomlčkou, a najdeme i jiné formáty, je potřeba vytvořit opravu. Zmíněný postup však patří do kategorie datové kvality, ne přímo do profilování. Uvedená analýza je často pouze vstupem pro další zpracování.

Jednou z možností, jak popsat data, je vytvořit regulární výraz na základě hodnot daného sloupce. Kvůli složitosti některých regulárních výrazů je tato analýza náročná na paměť, a zejména na čas. V praxi se také může ukázat, že výstup není snadno čitelný pro koncového uživatele.

Implementovaná analýza produkuje zjednodušený formát, který popisuje vstupní data. Výstupem není pouze jeden formát popisující všechny hodnoty, ale četnost různých formátů ve vstupních datech. Výstup je rozdělen do dvou kategorií:

- **Masky (masks)** - jednotlivé znaky jsou pouze převedeny do tříd: hodnoty [a-z] jsou převedeny na hodnotu `a`, [A-Z] na `A`, [0-9] na `0` a ostatní znaky jsou ponechány.

[11] https://www.baeldung.com/spring-value-annotation

[12] https://docs.spring.io/spring-framework/docs/4.3.10.RELEASE/spring-framework-reference/html/expressions.html#expressions-operator-elvis

- **Vzory (patterns)** - jednotlivé znaky jsou převedeny jako u masek, ale poté proběhne spojení sousedních stejných tříd - [Aa]+ na W, 0+ na N. Ostatní znaky jsou také zachovány.

Vstup	Maska	Vzor
Jan Novák	Aaa Aaaaa	W W
730101/0255	DDDDDD/DDDD	N/N
+420 123456789	+DDD DDDDDDDDD	+N N
+420123456789	+DDDDDDDDDDDD	+N
1000.57 Kč	DDDD.DD Aa	N.N W

Tabulka 5.10: Ukázka masek a vzorů pro různé hodnoty

Maska	Četnost
DDDDDDDDD	4093
+DDDDDDDDDDDD	1153
DDD DDD DDD	271
+DDD DDDDDDDDD	89
+DDD DDD DDD DDD	7

Tabulka 5.11: Výstup analýzy masek - telefonní čísla

Vzor	Četnost
W W	512
W W. W	64
W W-W	32

Tabulka 5.12: Výstup analýzy vzorů - jména

5.6.1 Implementace

Jelikož se jedná o frekvenční analýzu nad vstupem, který je pouze upraven, paměťová náročnost je v nejhorším případě $O(N)$. Aby analýza mohla běžet s omezenou pamětí, má konfigurovatelné maximum různých uložených masek a vzorů. Pokud je toto maximum překonáno, analýza již neprobíhá. Dochází pouze k předání informace o překonání maximální velikosti množiny různých masek nebo vzorů.

Vzory není potřeba počítat paralelně s maskami, stačí je spočítat až při vypsání výsledků. Pokud však počet masek přesáhne konfigurovaný limit, všechny masky se převedou na vzory. Algoritmus následně počítá pouze ty.

5.7 Ostatní analýzy

Práce obsahuje několik doplňujících analýz. Ty již nejsou tolik zajímavé z hlediska implementace, avšak pro uživatele mohou být stejně potřebné jako všechny předchozí.

5.7.1 Číselné statistiky

Pro číselné sloupce jsou spočítány statistiky typu minimum, maximum, průměr a rozsah.

5.7.2 Řetězcové statistiky

Analýza nad řetězci popisuje délku hodnot - minimum, maximum, průměr a rozsah, dále průměrný počet slov, znaků, malých písmen, velkých písmen a číslic pro jednu hodnotu.

5.7.3 Maximální a minimální hodnoty

Aplikace pracuje pouze s datovými typy, kde jsou jednotlivé hodnoty porovnatelné. Tudíž lze určit k největších nebo nejmenších hodnot. Přesně tento princip analýza implementuje. Uživatel zadá počet prvků a analýza vrátí k největších a nejmenších hodnot.

5.7.4 Pořadí dat

V některých případech bývají data setříděna podle některého ze sloupců. Analýza testuje, zda je zachováno pořadí hodnot ve sloupci.

Vlastnost	Kód
Vzestupně	ASC
Vzestupně s rovností	ASC OR EQUAL
Sestupně	DESC
Sestupně s rovností	DESC OR EQUAL
Všechny hodnoty rovny	ALL EQUALS
Jinak	NO ORDERING

Tabulka 5.13: Popis možných výsledků analýzy určující pořadí dat

5.7.5 Benfordův zákon

Benfordův zákon (Benford, 1938) říká, že v některých přirozeně se vyskytujících souborech číselných dat se první cifry vyskytují podle rozdělení v tabulce 5.14.

Výsledkem analýzy je histogram hodnoty první cifry čísla. Na základě výsledků analýzy je možné například detekovat podvody nebo chyby v datech (Saville, 2006).

Číslice	%
1	30,1%
2	17,6%
3	12,5%
4	9,7%
5	7,9%
6	6,7%
7	5,8%
8	5,1%
9	4,6%

Tabulka 5.14: Benfordovo rozložení jednotlivých číslic

6. Výsledky

Chování aproximativního profilování je demonstrováno na vstupních datech, která pocházejí z webové stránky Kaggle[1]. Konkrétně se jedná o datovou sadu NYC Parking Tickets[2]. Sada obsahuje čtyři CSV soubory o 10 miliónech řádků a 43 až 51 sloupcích. Každý soubor má velikost okolo 2 GB. Konkrétní parametry vstupních dat a jejich následné úpravy jsou popsány v průběhu této kapitoly a také v tabulce 6.1.

Hlavním cílem kapitoly je ukázat rozdíl mezi potřebnou pamětí RAM pro aproximativní profilování dat oproti jejímu přesnému ekvivalentu. Během měření byla omezována velikost haldy Java aplikace na co nejmenší velikost. Výsledná minimální hodnota ale nesměla příliš omezovat rychlost aplikace z důvodu častého volání Garbage Collectoru. Velikost potřebné paměti RAM byla pak asi o 20 MB větší než velikost haldy kvůli interní implementaci JVM[3]. Zmíněný rozdíl je zanedbatelný, proto tyto hodnoty nejsou zaznamenány v grafech. Všechny naměřené hodnoty reprezentují pouze minimální velikost potřebné haldy. Chování bylo zkoumáno pomocí nástroje VisualVM[4]. Ten poskytuje nejen přehled o alokované paměti, ale také počty a velikosti konkrétních tříd v paměti nebo časovou náročnost implementovaných metod.

	Soubor A	Soubor B	Soubor C
velikost souboru	2 GB	6.3 GB	5.7 GB
počet sloupců	43	43	43
sloupců STRING	26	27	27
sloupců LONG	15	15	16
sloupců DATE	1	1	0
sloupců BOOLEAN	1	0	0
počet řádků	10,8 M	10 M	10 M
unikátních hodnot	4 %	4 %	40 %
neprázdných hodnot	77 %	100 %	100 %

Tabulka 6.1: Přehled parametrů vstupních souborů

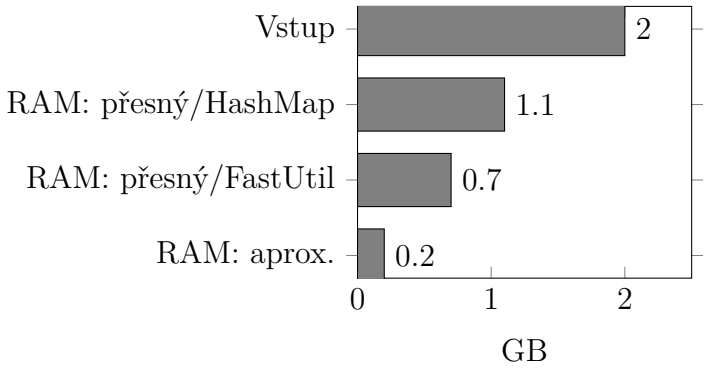
Kromě aproximativních analýz (frekvenční analýza, výpočet kardinality, kvantilů a histogramu hodnot) je pro profilování dat s omezenou pamětí využito také několik přesných analýz, které nemají implementaci závislou na velikosti vstupu (analýza masek a vzorů dat, výpočet číselných a textových statistik, nalezení maximálních a minimálních hodnot, analýza pořadí dat a frekvencí prvních cifer nebo provedení klasifikace dat). Vybrané aproximativní algoritmy společně s jejich parametry lze nalézt v tabulce 6.2. Výběr byl určen na základě výsledků z kapitoly 5. Referenční přesné profilování je pro získání správných hodnot omezené. Výpočet neobsahuje určení kardinality, histogramu hodnot a kvantilů. Tyto statistiky lze vypočítat na základě výsledků přesné frekvenční analýzy, tudíž se informace nemusí v paměti zbytečně duplikovat. Aplikace byla navržena pro aproximativní přístup a nebyla zde potřeba, aby analýzy měly takto sdílenou paměť.

[1]https://www.kaggle.com [2]https://www.kaggle.com/new-york-city/nyc-parking-tickets
[3]https://www.javatpoint.com/jvm-java-virtual-machine [4]https://visualvm.github.io

Analýza	Algoritmus	Parametry
Frekvenční analýza	Space-Saving	$\epsilon = 0{,}001$
Kardinalita	Linear Counting	$m = 2^{23}$
Histogram hodnot	S omezenou pamětí	$max_{\mathrm{i}} = 100, max_{\mathrm{u}} = 15$
Kvantily	MRL98	$b = 20, d = 1000$

Tabulka 6.2: Přehled použitých aproximativních algoritmů s parametry pro analýzu 10 miliónů záznamů

První popsané měření proběhlo na souboru A. Jedná se konkrétně o soubor `Parking_Violations_Issued_-_Fiscal_Year_2017.csv`[5]. Profilování tohoto vstupu slouží jako ukázka chování algoritmů na reálných datech. Nejprve proběhlo přesné profilování. Během měření však bylo zjištěno, že implementace mapy pomocí třídy `HashMap`[6] není ideální. Až třetina celkové paměti byla spotřebována pouze pro interní uchovávání této datové struktury. Proto byla doimplementovaná také varianta s použitím mapy `Object2IntOpenHashMap`[7] z knihovny FastUtil[8], které by měla stačit menší paměť pro ukládání hodnot. Potřebná paměť se pak opravdu snížila o 400 MB. Následně bylo spuštěno aproximativní profilování, které potřebovalo pouze 200 MB pro profilování celého 2GB souboru. Oproti přesné analýze se tak jedná o čtvrtinovou velikost potřebné paměti s nepřesností výsledků řádově v tisících procent.



Obrázek 6.1: Velikost potřebné paměti během profilování souboru A

Parametry aproximativních algoritmů jsou závislé na velikosti vstupu a počtu unikátních hodnot, ale tato závislost je velmi malá. Primární závislost je na přesnosti odhadu. Následující dvě měření demonstrují, že aproximativní algoritmy již nemění chování na základě různých parametrů vstupních souborů. Jediná výjimka je v případě algoritmu Space-Saving, který uchovává maximálně $1/\epsilon$ unikátních hodnot. Pokud však sloupec obsahuje méně unikátních hodnot, než je tento počet, je potřebná paměť přirozeně menší.

Vstupní soubor B vychází strukturou z reálného souboru A. S podobnými parametry byl vygenerován vstup, který však neobsahuje prázdné hodnoty. Velikost souboru vzrostla na trojnásobek hlavně z důvodu generovaných hodnot.

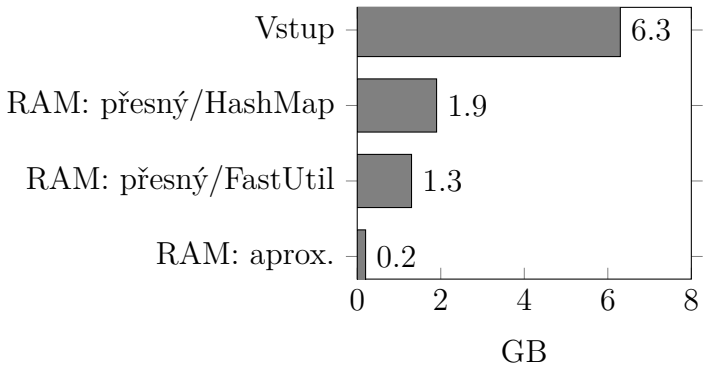
[5] https://www.kaggle.com/new-york-city/nyc-parking-tickets?select=Parking_Violations_Issued_-_Fiscal_Year_2017.csv

[6] java.util.HashMap [7] it.unimi.dsi.fastutil.objects.Object2IntOpenHashMap

[8] http://fastutil.di.unimi.it

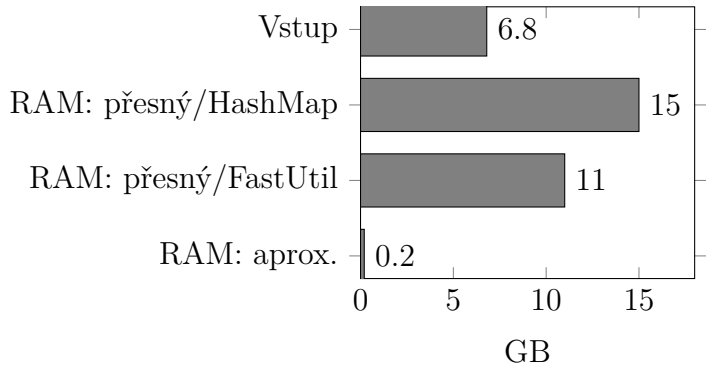
Kromě větší průměrné délky textových řetězců došlo k nárůstu velikosti souboru také z důvodu použití datového typu `Long`[9] při generování náhodných čísel, který po zápisu hodnot do souboru produkuje řetězce až o délce dvaceti znaků. V paměti jsou pak všechna čísla uložena do 8 bytů nehledě na velikosti jejich textové reprezentace.

Na základě změn parametrů souboru vzrostla potřebná paměť pro přesný výpočet pomocí `HashMap` z 1.1 na 1.9 GB a v případě `FastUtil` z 0.7 na 1.3 GB (obrázek 6.2). Důležitým pozorováním je, že zde nedošlo k nárůstu potřebné paměti u aproximativního přístupu s chybou stále maximálně v tisících procent. Hlavním důvodem je nastavení parametrů algoritmů na základě výsledné přesnosti a celkového počtu záznamů. Žádná statistika pak není přímo závislá na zvýšeném počtu neprázdných a unikátních hodnot. V paměti je také uloženo mnohem méně hodnot, tudíž zvýšení průměrné délky řetězců je ve finále zanedbatelné.



Obrázek 6.2: Velikost potřebné paměti během profilování souboru B

Vstupní sloupce souboru B stále obsahují pouze 4 % unikátních hodnot. Proto je v souboru C tento poměr zvětšen na 40 %. Aproximativní přístup opět dle očekávání nezaznamenal nárůst potřebné paměti ani maximální chyby výsledných hodnot. Problém však nastal u přesného výpočtu. Zde již bylo potřeba použít pro výpočet více než 10 GB paměti, což představuje výrazný nárůst požadavků na výpočetní stroj, na kterém běží datové profilování. Je tak velmi náročné profilovat velké soubory v paměti RAM pomocí přesných algoritmů.



Obrázek 6.3: Velikost potřebné paměti během profilování souboru C

[9]java.lang.Long

Aproximativní algoritmus byl nakonec porovnán s nástrojem Ataccama DQA, který je popsán v sekci 2.2.1. Při profilování souboru C potřebovala tato aplikace také pouhých 200 MB paměti RAM, ale navíc k tomu využívala disk. Během výpočtu bylo na disk zapsáno okolo 8 GB dat. Z toho vyplývá, že je zde ještě přidaná závislost na rychlosti zápisu na disk. Problém zde nastal v časové náročnosti analýzy. Celkem se jednalo o 57 minut. (Ataccama DQA byl výkonnostně nejlepší z testovaných nástrojů). V případě implementované aproximativní analýzy se jedná pouze o 3,5 minuty a zároveň není potřeba při výpočtu využívat disk.

Závěr

V diplomové práci se podařilo splnit stanovený cíl: nalézt, implementovat a otestovat aproximativní algoritmy vhodné pro datové profilování s omezenou pamětí. Měření ukázalo, že soubory o velikosti až desítek GB lze profilovat v paměti, která dosahuje pouze několika stovek MB. Paměťová náročnost je pak primárně závislá na počtu sloupců a na nastavení parametrů algoritmů dle požadované přesnosti.

Kromě implementace již známých aproximativních algoritmů Lossy Counting, Sticky Sampling, Space-Saving, Count-Min Sketch, Bloom filtr, Linear Counting, HyperLogLog a MRL98 práce předložila také vlastní variantu výpočtů histogramů, vzorkování souborů, hledání vzorů v datech a klasifikaci dat s využitím jazyka SpEL. Navíc bylo využito paralelního zpracování pomocí knihovny Disruptor.

Nadále lze práci rozvíjet implementací dalších typů jednosloupcových analýz. Také je možné rozšíření o vícesloupcové aproximativní analýzy. Druhou variantou, jak v práci pokračovat, je doladění implementace a přidání uživatelského rozhraní. Pak lze považovat práci za hotový nástroj k profilování dat s omezenou pamětí.

Seznam použité literatury

Abedjan, Z., Golab, L. a Naumann, F. (2015). Profiling relational data: a survey. *The VLDB Journal*, **24**(4), 557–581. doi: 10.1007/s00778-015-0389-y.

Ataccama (2020). How ai is transforming data management. URL `https://www.youtube.com/watch?v=bTHVpijICFI`.

Benford, F. (1938). The law of anomalous numbers. *Proceedings of the American Philosophical Society*, **78**(4), 551–572.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7), 422–42. doi: 10.1145/362686.362692.

Broder, A. a Mitzenmacher, M. (2004). Network applications of bloom filters: A survey. *Internet Mathematics*, **1**(4), 485–509. doi: 10.1080/15427951.2004.10129096.

Chen, Z. a Zhang, A. (2020). A survey of approximate quantile computation on large-scale data. *IEEE Access*, **8**, 34585–34597. doi: 10.1109/access.2020.2974919.

Cormode, G. a Muthukrishnan, S. (2004). An improved data stream summary: The count-min sketch and its applications. *LATIN 2004: Theoretical Informatics Lecture Notes in Computer Science*, page 29–38. doi: 10.1007/978-3-540-24698-5_7.

Durand, M. a Flajolet, P. (2003). Loglog counting of large cardinalities. *Algorithms - ESA 2003 Lecture Notes in Computer Science*, page 605–617. doi: 10.1007/978-3-540-39658-1_55.

Flajolet, P., Fusy, , Gandouet, O. a Durand, M. (2007). Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *2007 Conference on Analysis of Algorithms*, page 127–146.

Havrlant, L. (2014a). Hyperloglog: Jak odhadnout počet unikátních hodnot. URL `https://programio.havrlant.cz/hyperloglog/`.

Havrlant, L. (2014b). Loglog: Jak odhadnout počet unikátních hodnot. URL `https://programio.havrlant.cz/loglog/`.

LMAX-Exchange (2012). Performance results. URL `https://github.com/LMAX-Exchange/disruptor/wiki/Performance-Results`.

Manku, G. S. a Motwani, R. (2002). Approximate frequency counts over data streams. *VLDB 02: Proceedings of the 28th International Conference on Very Large Databases*, page 346–357. doi: 10.1016/b978-155860869-6/50038-x.

Manku, G. S., Rajagopalan, S. a Lindsay, B. G. (1998). Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record*, **27**(2), 426–435. doi: 10.1145/276305.276342.

Metwally, A., Agrawal, D. a Abbadi, A. E. (2004). Efficient computation of frequent and top-k elements in data streams. *Database Theory - ICDT 2005 Lecture Notes in Computer Science*, page 398–412. doi: 10.1007/978-3-540-30570-5_27.

Naumann, F. (2014). Data profiling revisited. *ACM SIGMOD Record*, **42**(4), 40–49. doi: 10.1145/2590989.2590995.

Saville, A. (2006). Using benford's law to detect data error and fraud: An examination of companies listed on the johannesburg stock exchange. *South African Journal of Economic and Management Sciences (SAJEMS)*, **9**. doi: 10.4102/sajems.v9i3.1092.

Shiu, A. (2014). Optimal streaming histograms. URL `https://amplitude.com/blog/2014/08/06/optimal-streaming-histograms`.

Vogiatzis, M. (2015). Frequency counting algorithms over data streams. URL `https://micvog.com/2015/07/18/frequency-counting-algorithms-over-data-streams/`.

Whang, K.-Y., Vander-Zanden, B. T. a Taylor, H. M. (1990). A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, **15**(2), 208–229. doi: 10.1145/78922.78925.

Wu, W. (2012). Sampling-based cardinality estimation algorithms: A survey and an empirical evaluation.

A. Přílohy

A.1 Obsah přílohy práce

- `data/` - ukázkové vstupy a výstupy nástroje
- `source/` - zdrojové kódy
- `approx-profiler.jar` - nástroj pro aproximativní profilování dat
- `metadata-get.jar` - pomocná aplikace pro načtení metadat
- `README_ANALYSIS.md` - přehled všech parametrů analýz
- `README.md` - ukázka několika scénářů demonstrujících postupy z textu

A.2 Spuštění aplikace

Aplikace `approx-profiler.jar`:

- `-c,--config-file` - cesta ke konfiguračnímu souboru
- `-p,--processor-type` - typ procesoru `SIMPLE/PARALLEL`
- `--help` - nápověda

Aplikace `approx-profiler.jar`:

- `-i,--input-file` - cesta ke vstupní konfiguraci
- `-o,--output-file` - cesta k výstupu
- `--help` - nápověda