ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Kategorizace aplikací na základě uživatelského chování v komponentě Gaming Mode
Student: Tomáš Vybíral
Vedoucí: Ing. Petr Kolář
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce zimního semestru 2020/21

Pokyny pro vypracování

Komponenta Gaming Mode antiviru Avast slouží ke snížení výkonového dopadu ostatních aplikací na běžící hru a potlačení nesouvisejících vyskakovacích oken. Základní kategorizace spuštěných aplikací je zajištěna detekčními mechanismy na straně klienta, avšak není dokonalá.
Cílem této práce je navrhnout a naimplementovat automatický skript, který bude pravidelně z dat chování uživatelů vypočítávat přesnější kategorizaci.

Postupujte v těchto krocích:
1. Proveďte analýzu komponenty Gaming Mode a souvisejícího backendového prostředí. Nastudujte použití potřebných nástrojů třetích stran.
2. Zanalyzujte strukturu zasílaných dat a jejich způsob uložení na backendu. V případě potřeby navrhněte změny.
3. Naimplementujte a zaintegrujte skript vyhodnocující chování uživatelů.
4. Výskyt anomálií v datech analyzujte a pokuste se odhalit jejich příčiny. V případě potřeby vytvořte další nástroje pro usnadnění analýzy.
5. Popište způsob testování doručení výsledné kategorizace na klientskou aplikaci.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 16. září 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Kategorizace aplikací na základě uživatelského chování v komponentě Gaming Mode

Tomáš Vybíral

Katedra Softwarového inženýrství
Vedoucí práce: Ing. Petr Kolář

10. ledna 2020

Poděkování

Tímto bych chtěl poděkovat všem lidem, kteří mi pomohli s touto prací. Jmenovitě potom bych potom rád poděkoval svému vedoucímu Petru Kolářovi za veškeré rady, které mi poskytl. Dále bych rád poděkoval týmům, které stojí za komponentou Gaming Mode a systémem Burger, že mi dokázali vždy všechno v nejlepším světle vysvětlit. Rád bych také poděkoval své přítelkyni a mé rodině, která mě donutila psát tuto práci a stále mě emocionálně podporovala.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 10. ledna 2020 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Tomáš Vybíral. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Vybíral, Tomáš. *Kategorizace aplikací na základě uživatelského chování v komponentě Gaming Mode*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato bakalářská práce se zabývá automatickou kategorizací aplikací na základě uživatelského chování. Popisuje způsob získávání těchto informací a navrhuje a implementuje způsob jejich vyhodnocování. Zabývá se také použitím testovacích nástrojů k analýze příchozích dat.

Klíčová slova Spark, Big Data, Python, Jupyter, NoSQL, Hry, Avast

Abstract

This bachelor thesis is dealing with automatic categorization of applications based on user behavior. It describes means of getting these information and design and implements their evaluation. It also deals with using testing utilities to analyze incoming data.

Keywords Spark, Big Data, Python, Jupyter, NoSQL, Games, Avast

Obsah

Úvod **1**

1 K pozadí této práce **3**
1.1 O komponentě Gaming Mode 3
1.1.1 Funkcionalita 3
1.2 Odesílání a zpracovávání dat 8
1.2.1 Infrastruktura na zpracovávání dat 8
1.2.2 Plánování úloh 9
1.3 Nástroje pro vývoj nad Burgerem 9
1.3.1 SQL příkazy v NoSQL databázi 9
1.3.2 Spark 9
1.3.2.1 O prostředí Jupyter 10
1.3.2.2 Interní knihovna pro práci s daty 10
1.3.3 Spark vs. MapReduce 11

2 Zpracovávání příchozích dat **13**
2.1 Struktura dat 13
2.1.1 Ukázka 14
2.1.2 Odesílaná data 15
2.2 Návrh kategorizačního skriptu 17
2.2.1 Algoritmus pro získávání aplikační kategorie 18

3 Implementace algoritmu **21**
3.1 Načítání dat z minulého dne 21
3.2 Sčítání příchozích dat 23
3.3 Přiřazení kategorie 25
3.4 Ukládání 26
3.5 Načítání starých dat 27
3.6 Nastavitelné konstanty 28

3.7 Bezpečnost 28
3.8 Nasazení 29

4 Testování **31**
4.1 Řesené problémy 31
4.2 Kontrola dat 32
4.3 Vyhledávání v datech 34
4.4 Analýza dat 34

Závěr **37**

A Použitá literatura a zdroje **39**

B Seznam použitých zkratek **41**

C Obsah přiloženého média **43**

Seznam obrázků

1.1 Proces přidávání aplikací do Gaming Mode komponenty 5
1.2 Hra OpenTTD (Open Transport Tycoon Deluxe) byla přidána do interního seznamu aplikací (aktuální verze, proto je zmíněna Do Not Disturb komponenta) 6
1.3 Část GUI ve kterém může uživatel přidávat aplikace ručně. Zároveň obsahuje seznam aplikací pro které se komponenta aktivuje (aktuální verze, proto je tu opět zmíněn Do Not Disturb komponenta) 6

2.1 Diagram zjednodušeného algoritmu rozhodování o kategorii aplikace (chybí porovnávání vůči všem druhům přidání) 19

3.1 Katalog datových kategorií systému Burger 23

4.1 Ukazka časového průběhu počtu kategorizovaných aplikací (aktuální verze, proto obsahuje i sloupeček presentation_count, který ovšem je důležitý pouze pro komponentu Do Not Disturb)) 33
4.2 Ukázka vyhledávání pomocí vyhledávácích nástrojů (vyhledávám podle id webového prohlížeče Google Chrome) 35

Úvod

V určitých případech, kdy antivirus provádí sken nebo jednu z naplánovaných úloh, tak je možné, že bude počítač těmito úlohami vytížen. To se ovšem může stávat, pokud uživatel bude hrát hry. Většina dnešních antivirů implementuje nějaký způsob, kterým by umožnila hráčům pozastavit náročné úlohy antivirové ochrany. Tyto antiviry implementují jednoduché řešení, kdy je nutné, aby uživatel ručně přidal aplikaci, pro kterou dojde k utlumení aktivit antiviru. V antivirovém programu Avast stejnojmené firmy k tomu slouží komponenta Gaming Mode.

V původním stavu komponenta Gaming Mode v antivirovém programu Avast byla vždy závislá na interním seznamu aplikací, které jsou považovány za hry a které za hry považovány nejsou. V případě, že daná aplikace běží, a již byla zdetekována, jsou utlumeny interní úkony Avastu (naplánované skeny jsou odloženy, vyskakovací okna se neobjevují). Navíc vývojáři v Avastu šli o kousek dál a přidali další vylepšení pro hráče her (vypne se notifikační a aktualizační servisa systému Windows, vypnou se veškeré nesystémové úlohy z Task Manageru včetně Update a Reboot úloh, které jsou také vypnuty během běhu aplikace). Problém byl, že interní seznam aplikací bylo nutné ručně neustále aktualizovat.

Hlavním cílem této práce je navrhnout a následně implementovat automatický skript, který by měl o určitých aplikacích rozhodovat, zda se jedná o hry nebo ne. K těmto rozhodnutím má docházet na základě automaticky sbíraných informací od uživatelů. Z tohoto hlavního cíle jsem se rozhodl si práci rozdělit na tyto úkoly:

- Zjistit, jak funguje komponenta Gaming Mode a jak funguje proces automatického přidávání.
- Zjistit, jak a kam se posílají uživatelská data. Jakým způsobem se dá s nimi pracovat a jak tento proces automatizovat.

- Navrhnout a implementovat řešení, které by využilo znalosti z předchozích bodů a které by řešilo výše zmíněný problém.
- Použít získaná data k analýze chování komponenty Gaming Mode.

Samotná práce je rozdělena na čtyři různé části. V první části popíšu, jak funguje komponenta Gaming Mode a jak funguje systém, na který se odesílají. V druhé části se zaměřím na samostatná odesílaná data a na návrh agregačního algoritmu. Ve třetí kapitole se zaměřuji na implementaci celého systému a pokouším se do podrobností dovysvětlit části kódu. V poslední kapitole se potom zaměřuji na testování vytvořeného řešení a zároveň se zmiňuji o tvorbě testovacích nástojů. V té samé kapitole se zmiňuji, co se mi pomocí těchto nástrojů podařilo zjistit o chování komponenty Gaming Mode.

Toto zadání jsem dostal jako první netestovací úlohu, když jsem začal pracovat v Avastu. Přišlo mi tak zajímavé, že jsme se s kolegou rozhodli, že se bude jednat o mou bakalářskou práci.

Kapitola 1

K pozadí této práce

V této kapitole vysvětlím prostředí a základní pojmy mé práce. Vysvětlím, co je Gaming Mode a proč již není aktuálně součástí antivirového programu Avast a AVG, i když byla na začátku vývoje tohoto projektu.

1.1 O komponentě Gaming Mode

Gaming Mode (dále již jako GM) byla komponenta antivirového programu Avast a AVG (jedná se o dva různé produkty jedné společnosti[1]). Každý větší antivirus implementuje nějakou podobnou komponentu, aby antivirus neubíral výkon ostatním procesům (např. při plánovaném skenu počítače). Jako příklad lze uvést například antivirový program BitDefender[2]. V Avastu se rozhodli, že ji implementují s přidanou hodnotou pro hráče.

V současné době byla komponenta přejmenována na Do not Disturb a lehce upravena, aby fungovala i na neherních aplikacích. Tato skutečnost, ale není pro tuto práci důležitá, proto se jí nadále nebudeme zabývat.

1.1.1 Funkcionalita

Po startu systému a nastartování hlavní antivirové servisy se po jedné minutě aktivuje GM komponenta. Toto časové omezení bylo záměrné, aby ostatní komponenty měly dostatečné množství času a prostředků k úspěšnému a rychlému startu.

Při inicializaci komponenty se sesychronizují data potřebná k fungování komponenty. Pro začátek řekněme, že se jedná o interní databázi her a seznam aktivovatelných pravidel (dovysvětleno dále). Pokud se nejedná o první spuštění komponenty, tak se ještě načtou uložená data. Zde je důležité zmínit seznam přidaných aplikací, pro které se GM aktivoval, a seznam již dříve testovaných aplikací.

Následně po spuštění začne komponenta sledovat nově spuštěné aplikace. Toto je možné kvůli zvýšeným pravomocím antivirové servisy, která se při

instalaci antiviru registruje do systému Windows, který ji potom udělí zvýšená práva.

Pokud uživatel spustil libovolnou aplikaci, jejíž proces v libovolném okamžiku vytvoří nové okno, a tato aplikace není v seznamu aplikací, pro které se GM aktivoval, a zároveň není v seznamu již testovaných aplikací, tak se komponenta pokusí přidat tuto aplikaci do zmíněného seznamu.

V prvním kroku komponenta hledá exe soubor procesu v interním databázi. Nejdříve se podívá zda se přidávací proces nemá přerušit (testuje je zda se nejedná o aplikaci o které víme, že není hrou – tzv. herní blacklist) a pokud ne tak se potom testuje zda aplikace není jedna ze známých her (tzv. herní whitelist). Tato databáze se (jak již bylo zmíněno) pravidelně stahuje při startu komponenty.

K vyhledávání aplikací v databázi se používá systému Wildcard od Microsoftu. Jedná se jednoduchý způsob porovnávání textu s libovolným vzorem[3]. Syntaxe systému Wildcard je (například oproti regulárním výrazům) naprosto jednoduchá. Pro pozitivní nalezení aplikace musí dojít ke shodě mezi cestou ke spustitelnému souboru aplikace a libolným vzorem v herním whitelistu.

Pokud došlo ke shodě, tak je aplikaci přiřazeno unikátní id. Aktuálně se začala v Avastu vytvářet inciativa sdíleného zdrojového kódu pro všechny produkty a proto k výpočtu byla použita hashovací funkce převzatá z komponenty Avast Cleanup (v dnešní době se jedná o samostatný software). Tento hash je počítán z registrů systému Windows a to přesně z odinstalačního klíče pro danou aplikaci.

Pokud se již tento hash nachází v seznamu přidaných her je aplikace (resp. cesta ke spustitelnému souboru) přidána do seznamu pod toto id. V tomto okamžiku jsou testovány všechny cesty ke spustitelným souborům. Zjišťuje se, zda dané soubory existují, v případě že ne, se tyto cesty smažou.

Pokud id v seznamu není, je vytvořen nový záznam s vypočítaným hashem a cestou k dané aplikaci.

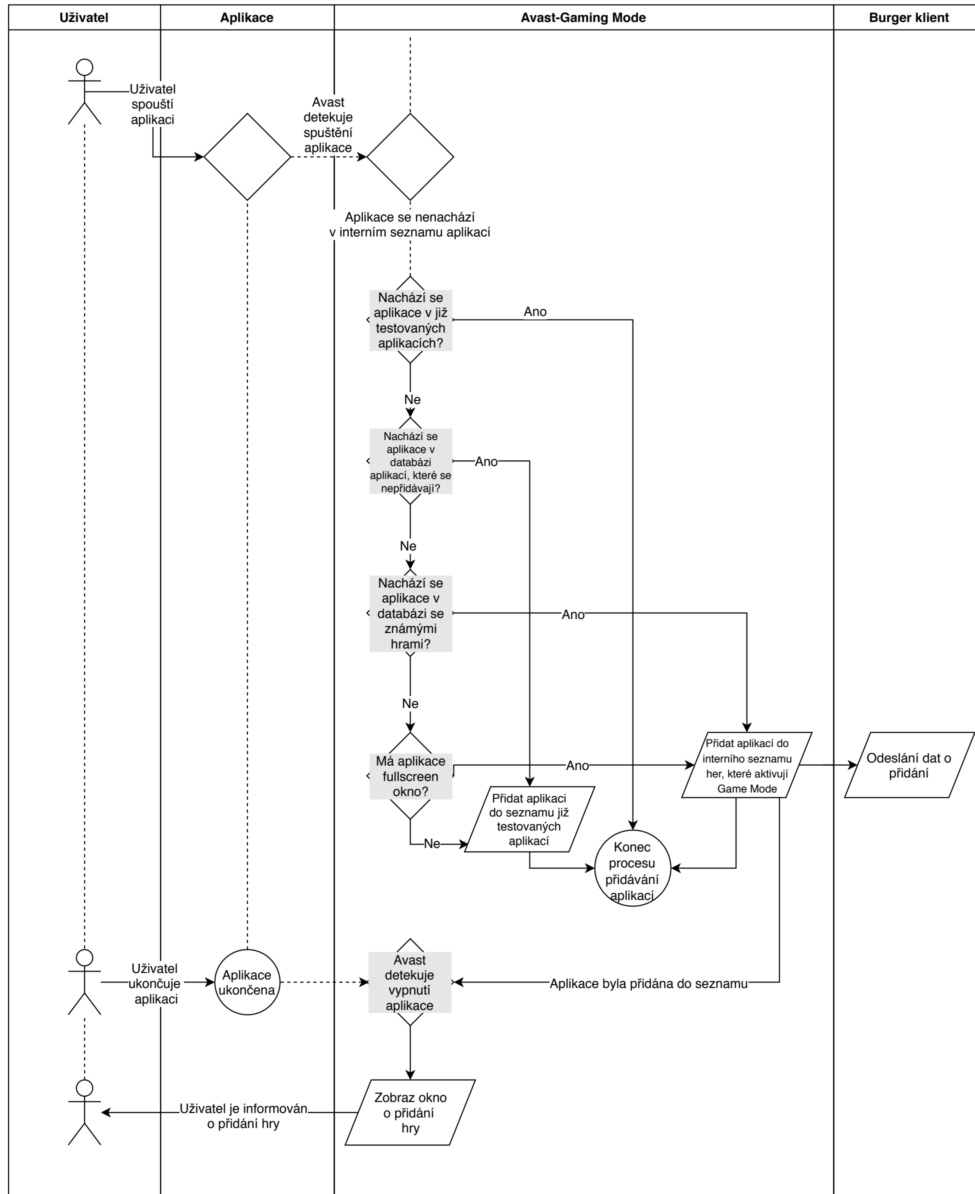
Pokud aplikace nebyla nalezena v interní databázi jsou potom veškerá její okna otestována zda nejsou ve fullscreen módu. Testuje se, zda mají interně nastavený flag `FULLSCREEN` pozitivně a nebo okno postrádá veškeré okraje a je zobrazeno přesně přes celou obrazovku. Pokud je takto libovolné okno detekováno, postupuje se stejným způsobem jako u detekce vůči databázi.

Po ukončení aplikace se zobrazí upozornění, že hra byla automaticky detekována a že byla přidána do seznamu GM aplikací (obrázek 1.3).

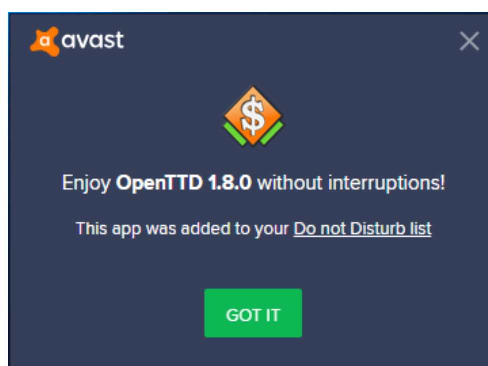
Pokud došlo k pozastavení procesu přidávání a aplikace nebyla přidána do interního seznamu aplikací pro které se GM spouští, tak je přidána do seznamu testovacích aplikací, aby při dalším spuštění aplikace nedošlo k novému testování.

Pokud aplikace nebyla přidána automaticky, má uživatel možnost si aplikaci přidat do seznamu ručně přes GUI antivirového programu. V tom samém okně lze také vidět veškeré aplikace, které se nachází v seznamu aplikací, pro které by se GM aktivoval.

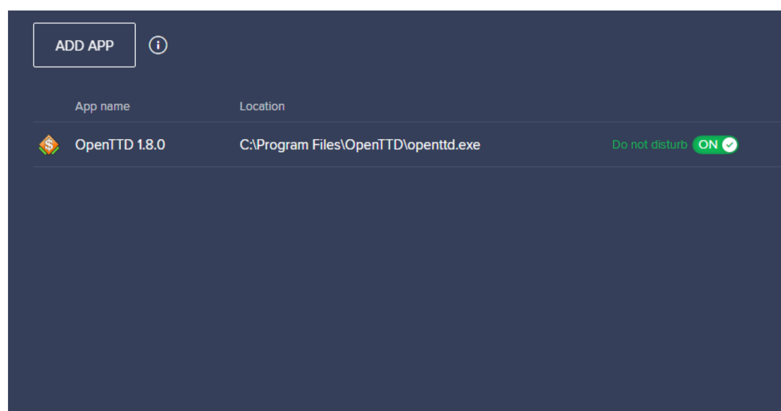
Přidávání aplikace



Obrázek 1.1: Proces přidávání aplikací do Gaming Mode komponenty



Obrázek 1.2: Hra OpenTTD (Open Transport Tycoon Deluxe) byla přidána do interního seznamu aplikací (aktuální verze, proto je zmíněna Do Not Disturb komponenta)



Obrázek 1.3: Část GUI ve kterém může uživatel přidávat aplikace ručně. Zároveň obsahuje seznam aplikací pro které se komponenta aktivuje (aktuální verze, proto je tu opět zmíněn Do Not Disturb komponenta)

Při libovolném přidání aplikace se odešlou anonymizovaná data o přidání aplikace. Tato obsahují:

- id aplikace
- způsob, jakým byla aplikace přidána (detekce databází, fullscreen detekce, přidána ručně uživatelem)
- cestu ke spustitelnému souboru, který započal přidávání aplikace
- kopii záznamu v seznamu aplikací GM (aby bylo vidět zda byl vytvořen nový záznam, nebo jestli byla aplikace přidána do již existujícího záznamu)

Více informací k odesílání dat se nachází v kapitole 1.2.

Přidané aplikace je možné v jakémkoliv okamžiku odebrat pomocí GUI a aplikace již následně nebude spouštět žádná pravidla a Avast se nebude snažit o její znovupřidání do komponenty.

V případě že aplikace se již v seznamu GM aplikací nachází, tak se v okamžiku jejího spuštění aktivuje sada pravidel. Seznam pravidel je volně rozšiřitelný a je definován souborem, který byl stažen při statu aplikace (jak již bylo zmíněno). Tato pravidla jdou případně vypnout nebo zapnout. K tomu lze použít jednu z těchto dvou možností:

1. Odstraněním ze seznamu pravidel, který se synchronizuje pravidelně při startu komponenty. Velice vhodné pokud je potřeba zamezit fungování pravidla napříč verzemi.

2. Interním konfiguračním souborem. Výhodou tohoto řešení je, že je možné upravit dle potřeby pro jednotlivé verze nebo operační systémy. O synchronizaci tohoto konfiguračního souboru se stará antivirus.

Zde je seznam nejdůležitějších implementovaných pravidel:

DisableAvNotification Zakáže anitiviru zobrazovat jakákoliv vyskakovací okna.

PauseAvBackgroundTasks Zastaví již spuštěné úlohy antiviru a zároveň zakáže spouštění dalších naplánovaných úloh.

PauseSystemBackgroundTasks Toto pravidlo vypne všechny v systému naplánové úlohy, ale které nejsou systémové (tedy autorem není Microsoft).

PauseWindowsUpdate Vypne Windows Update servisu a zároveň přeruší Windows Update úlohu v Task Manageru.

DisableWinUpdateAutoReboot Pokud je naplánovaný restart systému, tak dojde k jeho přerušení.

DisableDrawOverWindow Pokud se jiný proces snaží vytvořit okno, které se nachází na vyšší vrstvě, než aktivní herní okno, tak potom je herní okno přesunuto na vyšší vrstvu než okno neherního procesu.

DisableWindowsNotification Vypne notifikační servisu systému Windows, tudíž nejsou zobrazovány žádné dialogy.

KeepGameInForeground Pravidelně zamyká okno hry, tak aby žádné jiné okno se nestalo samovolně aktivním oknem.

SetHighPerformanceMode Nastaví Power Manageru systému Windows na Hight Performance Mode. Tím se zvýší možné zatížení hardwaru na maximum.

SetHightPriority Nastaví vysokou prioritu procesu hry.

Poté co se aplikace ukončí se všechna tato pravidla vypnou a jejich efekty se revertují.

1.2 Odesílání a zpracovávání dat

Společnost Avast má systém na anonymní sběr dat od uživatelů, který je používán napříč jejími produkty. Tato data jsou určena k analýze uživatelského chování. Tento systém využívá vlastní Big Data řešení, které vzniklo z důvodu ušetření financí za poplatky za cizí cloudová řešení (např. AWS[1]) a aby zamezil používání analytických nástrojů třetích stran (např. Google Analytics[2,3]) a musel kvůli tomu zpřístupnit tato data pro jiné lidi než zaměstance firmy. Zároveň se začali vyskytovat problémy s používáním Google Analytics, jelikož tato služba se stata pod náporem většího množství dat nepoužitelná. Tento systém je také pro firmu lépe rozšiřitelný. Název tohoto systému je Burger.

Veškerá data jsou anonymizována, ale i přesto je nutné rozlišovat od koho tato data přišla aby nebyla některá data započítávána vícekrát. Proto je každý uživatel identifikován speciální uživatelským id (jedná se o jednosměrné UID).

K reprezentaci dat na straně klienta se používá technologie Google Protocol Buffer. Tento systém umožnuje vytvořit univerzální vzor[4], pro který lze vygenerovat příslušné datové struktury pro velké množství programovacích jazyků (oficiálně jsou podporovány tyto jazyky: Java, Python, Objective-C, C++, Dart, Go, Ruby, C#, JavaScript, PHP[5]). Tento vzor není problém rozšířit a zachovat tím podporu starších verzí softwaru. Toto rozšíření je umožněno pouze nepovinnými poly. Toto znamená, že pokud bylo nějaké pole označeno jako povinné a v nové verzi již není podporované, tak je nutné jej vyplnit nějakou hodnotou. Proto bylo v Avastu zavedeno, že veškeré vzory budou mít všechny prvky označeny jako nepovinné.

Aby nedocházelo k zahlcení severu jsou nejdříve data shromažďována lokálně a po uplynutí pravidelného intervalu[4] se takto nashromážděná data odešlou na analytické servery.

1.2.1 Infrastruktura na zpracovávání dat

Celá beckendová infrastruktura je založena na open source technologii Apache Hadoop[6]. Tato technologie umožnuje vytvářet výpočetní clustery, které umožňují rychlé zpracování velkého množství dat. Zároveň tato technologie implementuje vlastní souborový systém HDFS[5][7], který je optimalizován na velké

[1] Amazon Web Services – cloudové výpočetní řešení od firmy Amazon. Umožňuje vytvářet a spouštět přes internet přístupné servery dle definice uživatele

[2] Analyticky systém od firmy Google.

[3] V době psaní Avast ještě v některých částech tento systém používal

[4] v době psaní se jednalo o 1 hodinu

[5] Hadoop Distributed File System

množství dat. Zároveň takto uložená data jsou potom lépe přístupná pro výpočetní cluster. Pro přístup k datům tato technologie implementuje vlastní verzi způsobu dotazování nad daty (tzv. MapReduce)[8] Ovšem tento systém není běžným uživatelům v Avastu přístupný. Veškerý způsob dotazování nad daty je postaven na systému plánování úloh, který je také Hadoopem implementován[9].

Zpracování dat od klientů je řešena pomocí Apache Kafka. Tento systém je navrhnut tak aby byl schopen zpracovávat neustálý tok dat [10]. Tento systém je nasazen na vícero serverech. Ty zpracovávají data a ukládají je do dočasných adresářů. Tyto adresáře jsou 1 za den zpracovány a jejich data jsou uložena do speciálních adresářů. Data jsou takto řazena podle typu dat a dle dne jejich odeslání.

Samotné ukládání odeslaných dat řeší služba Apache Cassandra. Tato samostatná služba běží nad HDFS a funguje jako NoSQL databáze, která ukládá nová data uloží [11].

1.2.2 Plánování úloh

Pro plnou automatizaci všech skriptů se používá systém Azkaban. Ten umožňuje vytvářet úlohy a plánovat jejich pravidelné spouštění. Také umožňuje vytvářet mezi jednotlivými úlohami závislosti[12].

1.3 Nástroje pro vývoj nad Burgerem

V této části proberu možnosti vývoje nad prostředím Burger. Uvedu možnosti jak získávat data a jak spouštět vlastní dotazy nad jednotlivými daty.

1.3.1 SQL příkazy v NoSQL databázi

Pro prohlížení HDFS a spouštění SQL příkazů nad uloženými daty je v Avastu použit systém Hue[13]. Toto webové rozhraní dokáže přečíst SQL příkaz a potom pomocí několika dalších nástrojů spustí uživatelem napsaný dotaz nad daty.

Prostředníka mezi SQL dotazem a datovým úložištěm (kde data byla uložena jako NoSQL databáze) slouži Apache Hive. Tento nástroj umožňuje spouštění SQL dotazů nad daty uložené v HDFS. Interně překládá uživatelské dotazy do MapReduce kódu, který je následně spouštěn.

1.3.2 Spark

Dalším způsobem jak v Avastu získávat datové výstupy z uložených dat je Apache Spark[14]. Tento systém je více podobný MapReduce v Hadoopu než Apache Hive. Umožňuje totiž sestavovat dotazy nad databázemi pomocí klasických programovacích jazyků. Zároveň názvy metod, které jsou přístupné

z datových struktur, se podobají SQL příkazům. Je tedy jednodušší pro použití pro SQL programátora, než Hadoop MapReduce. Tento systém je postaven nad Hadoopem a je plně implementován s Hadoop Yarnem. Umožňuje programovaní v libolném jazyce, který běží v JVM (v Avastu – Scala, Java, JPython[6,7]).

Pro vývoj ve Sparku v Avastu se dají použít 2 možnosti:

1. Pomocí ssh[8] přístupu vyvíjet rovnou v clusteru a spouštět kód přímo tam. Tato možnost je ale velice nebezpečná. Proto tato možnost není doporučena pro začátečníky.

2. Využít PySpark a použít vývojové prostředí Jupyter, které podporuje vývoj pro Spark. Tato možnost je mnohonásobně bezpečnější a doporučená pro začátečníky.

1.3.2.1 O prostředí Jupyter

Prostředí Jupyter umožnuje vyvíjet programy skrz webové prostředí v tzv. Jupyter Noteboocích. Tyto notebooky umožnují kód rozdělit do logických celků po tzv. buňkách, které lze spouštět v libovolném pořadí[15]. Ke každé z těchto buněk patří (kromě zdrojového kódu) také datová část, kde je určeno jaký jazyk se v dané buňce nachází a zároveň samotný editor umožňuje uživateli vkládat vlastní datové struktury ve formátu JSON pro jednotlivé buňky. Samotný notebook je ukládán ve formátu JSON, což umožnuje jejich editaci i bez prostředí, které dokáže tyto notebooky otevřít. Ale i tak používají vlastní příponu souboru `.ipynb`.

Historicky se tento projekt jmenoval IPython[16] a umožňoval spouštět kód napsaný v programovacím jazyce Python ve webovém vývojovém prostředí. V dnešní době ale umožnuje interpretovat celou řadu jazyků[17], a to i ty které normálně nejsou interpretovatelné (jako např. C++, …).

V Avastu je toto prostředí použité pouze pro vývoj scriptů v PySparku (používá se kernel `sparkmagic` – resp. jenom v kombinacích s programovacím jazykem Python), který je napojen na Spark modul Burgeru.

1.3.2.2 Interní knihovna pro práci s daty

Pro Spark byla v Avastu vytvořena speciální knihovna, která ulehčuje přístup k datům. Normálně je nutné sparku říci, kde jsou data uložena (toto by bylo nutné pro kombinaci Hive + SQL), ale díky této knihovně stačí zadat tabulku kterou chcete (v tomto případě se načtou veškerá data v tabulce za předchozí den). Dále lze získat data z tabulky omezena dnem jejich odeslání.

[6] Implementace programovacího jazyka Python v Javě

[7] Pro kombinaci JPython + Spark se často používá název PySpark a jelikož je nutná samostatná implementace, tak má i vlastní dokumnetaci

[8] Secure Shell

Tato knihovna zároveň implementuje několik šikovných funkcí pro práci se soubory na HDFS a některé často používané funkce.

Knihovna byla primárně vytvořena pro programovací jazyk Scala. Ale byl vytvořen wrapper pro jazyk Python, který volá definice těchto funkcí přímo v JVM.

1.3.3 Spark vs. MapReduce

Pro obě technologie jsou v Avastu nabízena dobrá vývojová prostředí.

Pro práci s databázemi se hodí používat SQL, ale jelikož v tomto případě nejedná o SQL databázi může se jednato o zbytečnou přítěž a může dojít ke zbytečnému zpomalení z důvodu překladů dotazů.

Dle [18] je MapReduce bezpečnější, ale může být až stokrát pomalejší (spíš jenom desetkrát pomalejší, protože v základním nastavení v Avastu jsou data ukládána na disk a ne přímo do paměti). Ovšem dle [19] je MapReduce schopen být stejně rychlý nad velkým množství dat, pokud má k dispozici dostatečné množství paměti RAM. Ovšem tento zdroj také říká, že Spark je pro rychlé zpracovávání dat lepší než MapReduce.

První zdroj[18] také zároveň uvádí, že je Spark v zásadě jednoduší, ale to v našem případě není pravda, protože my bychom používali SQL příkazy, které by do MapReduce překládal Apache Hive.

Nakonec jsem se rozhodl, že v této práci použiju Spark framework a to také díky tomu, že pro něj je v Avastu vytvořena knihovna, která ulehčuje přístup k jednotlivým uloženým datům a obecně vyšší rychlost.

Kapitola 2

Zpracovávání příchozích dat

V této kapitole se budu podrobněji zabývat odesílanou strukturou dat, která přicházejí od klienta. Následně navrhnu řešení na zpracovávání dat.

Je také dobré zmínit, že se v této i další kapitole v některých případech budu odkazovat na interní firemní dokumentaci k systému Burgeru, která bohužel není veřejně přístupná.

Dále bych chtěl zmínit, že vývoj a návrh agregačního skriptu byl postupný a některé náležitostí nebyly v první veřejně použité verzi tohoto skriptu. Pokud narazíte na část kód, která není zmíněna v této kapitole, je to protože jsem se rozhodl danou část přidat na základě testování nebo jiných okolností.

2.1 Struktura dat

Jak již bylo zmíněno v kapitole 1.2 je pro ukládání a posílaní dat se používá Google Protocol Buffer formát. Každá datová struktura je definována v textovém souboru. Pro definici se používá podobný formát, jakým by se definovala libovolná struktura v programovacím jazyce C nebo C++. Jedinou změnou je, že je nutné určit pořadí v jakém mají být data ukládána a je nutné předznamenat každou proměnnou speciálním klíčovým slovem:

required Povinné pole. K úspěšnému odeslání dat je nutné tuto položku vžy vyplnit. Pokud není uvedeno žádné z těchto klíčových slov, tak se automaticky počítá, že pole je povinné. Jak již bylo zmíněno v kapitole 1.2, tak se tato vlastnost v datových strukturách v Avastu nevyužívá.

optional Nepovinné pole. Data se odešlou, i pokud nebudou obsahovat žádná relevantní data.

repeated Alternativa ke klasickému poli. Data se odešlou i za předpokladu, že tato datová struktura bude prázdná.

2.1.1 Ukázka

V této části ukážu jednoduchou ukázku definice Protocol Bufferu a vysvětlím na něm syntaxi definice datových struktur. Tato ukázka nemá nic společného s touto prací a slouží čistě k vysvětlení základních principů.

Takto nějak by mohl vypadat jednoduchý Protocol Buffer definiční soubor:

```
syntax = "proto2";

package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

Pro ukázku byl použit soubor z oficiální dokumentace Protocol Buffer[20].

Jako první se v souboru definuje syntaxe, kterou má kompilátor použít při parsování tohoto definičního souboru. V tomto případě (a ve všech následujících případech) je použita `proto2` syntaxe. Dále ještě existuje `proto3` syntaxe, ale tou se v této práci nebudu zabývat a v následující části vám popíšu pouze `proto2` syntaxi.

Následuje definice názvu balíčku. Tento název se potom použije při generování zdrojového kódu. Například pro programovací jazyk Java by se vytvoří tento balíček:

```
package tutorial;
```

Pro jazyk C++ by se vytvořil:

```
namespace tutorial {
```

A tak dále.

V další části definičního souboru se již nachází samostatná definice datových struktur. Každá datová struktura začíná klíčovým slovem `message`. Po něm následuje název datové struktury, složené závorky a potom obsah dané datové struktury.

Tyto struktury jsou potom následně přeloženy do struktur pro daný programovací jazyk (ve většině případů budou přeloženy na datový typ `class`).

Každá proměnná je v ukázce definována stejným způsobem: `<povinost parametru> <typ> <název> = <pořadí v datové struktuře>;`

V ukáce v datové struktuře `Person` je definována datová struktura `PhoneNumber` uvnitř definice struktury `Person`. Toto říká kompilátoru, že struktura `PhoneNumber` již nebude použita v žádné jiné struktuře a kompilátor se podle toho přizpůsobí. Zároveň si můžeme také všimnout definice výčtové hodnoty (klíčové slovo `enum`). Ve struktuře `PhoneNumber` je vidět definice základní hodnoty pomocí klíčového slova `default`.

2.1.2 Odesílaná data

V této části rozeberu odesílaná data a jaké infromace se v jednotlivých polích odesílají.

```
package asw.gamingmode.proto;

option java_package = "com.avast.analytics.proto.blob";

option java_outer_classname = "GamingMode";

enum Category {
    NONE = 0;
    GENERAL = 1;
    GAME = 2;
}

message FileVersion {
    optional uint32 major = 1;
    optional uint32 minor = 2;
    optional uint32 build = 3;
    optional uint32 private = 4;
}

message FileInfo {
```

```
    optional string product_name = 1;
    optional string company_name = 2;
    optional FileVersion product_version = 3;
}

message DisabledRule {
    optional uint32 rule_id = 1;
}

message Application {
    optional string id = 1;
    optional string renamed_name = 2;
    optional string name = 3;
    optional string publisher = 4;
    optional Category category = 5;
    optional bool enabled = 6;
    repeated DisabledRule disabled_rule = 7;
    repeated string executable_name = 8;
    optional FileInfo file_info = 9;
}

message SubmitEntry {
    optional Application application = 1;
    optional string file_path = 2;
}

message SubmitGame {
    optional SubmitEntry submit_entry = 1;
    optional uint32 detection_version = 2;
    optional FileVersion gaming_mode_version = 3;
}
```

Vstupním bodem kompletní datové struktury je struktura `SubmitGame`. V té se kromě instance struktury `SubmitEntry` se odesílají i data o verzi interní databázi her (pole `detection_version`) a verzi Gaming Mode komponenty (pole `gaming_mode_version`).

Další úrovní je datová struktura `SubmitEntry`. V té se spolu se strukturou `Application` posílá i cesta k poslednímu spuštěnému souboru patřícímu pro danné id.

Nejnižší strukturou je `Application`. Tato struktura je tvořena přímo z infomací uložených v seznamu aplikací, pro které se aktivuje Gaming Mode komponenta. V té se posílají nejdůležitější informace, které budu zpracovávat. Mezi nejdůležitější patří tato pole:

id Obsahuje vypočítané id z cesty ke spustitelnému souboru.

name Název aplikace.

executable_name Seznam cest ke spustitelným souboru aplikací přidaných pod dané id.

category Kategorie pod kterou byla aplikace zdetekována.

Některá pole již nejsou používána (jako `disabled_rule` ve struktuře `Application`), ale i přesto musí zůstat v definičním souboru, protože by se tím porušila zpětná kompatibilita, ale díky tomu, že se jedná o nepovinná pole, tak není nutné probýrat jejich obsah.

2.2 Návrh kategorizačního skriptu

Na začátku bylo nutné si vyčlenit jednotlivé cíle tohoto skriptu.

Zároveň je nutné zmínit, že v této době jsme s kolegy došli k rozhodnutí, že bude dobré spouštět jednou denně (podrobnosti k tomuto rozhodnutí v kapitole 3.1).

Prvním cílem bylo získat správná data, která ještě nebyla započítána. Rozhodl jsem se, že vždy použiji data z předchozího dne. Toto mi zaručí, že data budou vždy kompletní za daný den a pokud je zpracuji pouze jednou, budou zaručeně ještě nezpracována.

Tato data je potom nutné následně zpracovat. Je například nutné se zbavit duplicitních dat[9], sečíst ruční přidání od uživatelů a smazání hry ze seznamu podporovaných her uživateli. Tato data budou sloužit k určování zda se jedná o hru či ne.

Následně je nutné načíst stará data a aktualizovat hodnoty o nově získaná data z přechozího dne. Tímto získáme nejnovější hodnotu pro výpočet kategorie, která bude aplikaci přiřazena.

Při výpočtu se tedy na základě nových dat rozhodneme jakou z následujících kategorií přiřadíme danné aplikaci:

NONE Skript nemá dostatečné množství informací pro určení kategorie.

GAME Skript rozhodl, že se jedná o hru a tato hra by se měla přidat do seznamu aplikací pro které se má Gaming mode aktivovat.

GENERAL Skript se rozhodl, že aplikace není hrou a není ji tedy nutné již nadále testovat a přidávat do seznamu her, pro které se má Gaming Mode spouštět.

Na konec se nová data uloží a přepíší se tím stará data, aby takto získané nové výsledky byly možné následně použít při dalším spuštění skriptu.

[9] v případě, že by se někdo pokoušel ovlivnit data a posílal by pořád dokola stejná data

2.2.1 Algoritmus pro získávání aplikační kategorie

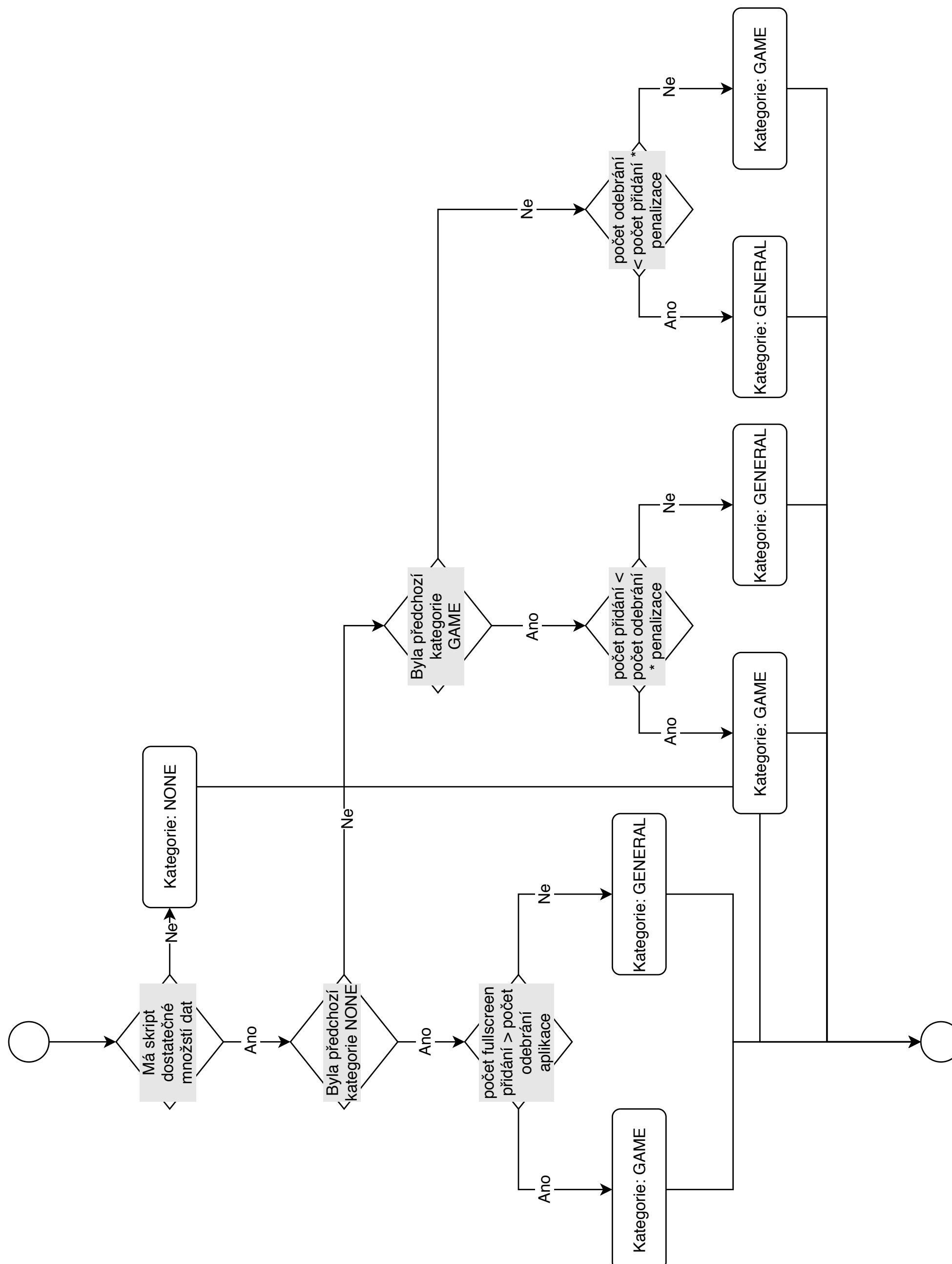
Na začátku jsme se rozhodli, že nejjednodšší bude pouze porovnávat hodnoty přidání a odebírání pro danou aplikaci a dle toho se rozhodnout, zda se jedná o hru nebo ne. Ovšem velmi brzo jsem si uvědomil, že je zbytečné určovat kategorii pro id aplikací o kterých je minimalní záznam v datech. A tudíž jsem se rozhodl, že budu kategorii určovat pouze u aplikací, které mají již určitý počet přidání a odebrání a je tedy možné určit s vyšší přesnotstí jakou kategorii vybrat. Pro tyto aplikace je právě určena kategorie `NONE`.

Tento návrh přišel později při vývoji a to hlavně, protože se nám v databázi během testování začali objevovat id aplikací, která měla pouze minimalní počet přidání a přišlo mi nesmyslné přiřazovat těmto aplikacím kategorii z důvodu nedostatku informací.

Následující omezení také vzniklo až při zkoumání výsledků z databáze. V případě aplikací, které měli větší množství dat, se začal objevovat problém, že stačilo pár naštvaných lidí, kteří nechtějí používat Gaming Mode, ale na místo toho aby komponentu odinstalovali, tak pravidelně mazají všechny přidané aplikace. Což opravdovým hrám způsobilo, že jim byla přiřazena kategorie `GENERAL`. Proto jsem začal zpracovávat všechny druhy přidání aplikací a tento počet všech přidání poměrově porovnávám se součtem ručních přidání a odebráním aplikací. Pokud není tento poměr dostatečný a aplikaci by měla být změněna kategorie `GENERAL` z kategorie `GAME`, tak se tato změna neprovede. A opačně pokud má aplikace kategorii `GENERAL` a tento poměr přidání je dostatečný, tak je aplikaci přiřazena kategorie `GAME`.

Následně jsem měl problém, že data, která mají přibližně stejný počet ručních přidání a odebrání aplikací ze seznamu, často mění kategorii, protože stačil lehký výkyv na jednu stranu. Proto jsem přidal podmínku, že při změně kategorie je novým hodnotám, které podorují novou kategorii, přidána penalizace. Více informací v kapitole 3.3.

Celý algoritmus je také znázorněn na obrázku 2.1.



Obrázek 2.1: Diagram zjednodušeného algoritmu rozhodování o kategorii aplikace (chybí porovnávání vůči všem druhům přidání)

Kapitola 3

Implementace algoritmu

K implementaci algoritmu jsem se rozhodl použít prostředí Jupyter a Jupyter Notebooky a to především z toho důvodu, že programovací jazyk Python je mi bližší než programovací jazyk Scala. A také, že pokud bude někdy někdo na mojí práci navazovat, tak si z osobního pohledu myslím, že programovací jazyk Python je mnohonásobně jednodušší pro úplné začátečníky.

Rozhodl použít prostředí Spark. Jak již bylo zmíněno v kapitole 1.3.3, jedná se ve většině případů o rychlejší variantu a i v našem případě by měla být rychlejší než druhá možnost (prostředí Hue a technologie MapReduce).

Při implementaci bylo nutné postupovat pomalu a kontrolovat si jednotlivé kroky, jestli fukce dělá, to co jsem zamýšlel, aby dělala.

Proto jsem si rozdělil celou práci do několika dílčích kroků. Při implementaci jsem využil rozdělení Jupyter Notebooků na jednotlivé buňky a díky tomu jsem mohl každou část rozdělit do jednotlivých logických celků (resp. buněk).

3.1 Načítání dat z minulého dne

Pokud bych používal čisté prostředí Spark, musel bych začít definicí kontextu, ve kterém bude skript pracovat. Toto naneštěstí řeší již zmíněná interní knihovna[10] `spark2.utils`. Po zavolání

```
import spark2.util
```

je uživateli zpřístupněna proměná `spark`, která již obsahuje předdefinovaný kontext. Tento kontext lze následně konfigurovat. V mém případě jsem musel povolit, aby Spark používal `CROSS JOIN`.

Data, která chci načítat jsou ukládána každý den složek, které identifikují jejich typ a den odeslání dat. Opět pokud bych používal klasický Spark, tak bych musel kontextu, říci kterou složku použít. Knihovně `|saprk.utils|`stačí

[10] Kapitola 1.3.2.2

předat typ a časové (datumové) rozmezí a uživatel dostane všechna data. Ovšem k aktivování těchto funkcí je nutné importovat jeden submodul této knihovny. Proto můžeme původní import nahradit:

```
from spark2.utils import dataframe_utils
```

I v tomto případě je vytvořena proměnná `spark` se správným kontextem. Ta se totiž tvoří v každém modulu této knihovny.

Nyní se proměnné `spark` zpřístupnila funkce

```
spark.catalogData
```

a

```
spark.catalogDataAct
```

.

V obou případech je nutné funkci předat typ dat, která chceme načíst. V prvním případě ještě musíme přidat časový rozsah dat, která chceme načíst. Ten je definován dvěmi datumy, které je nutné zapsat ve formátu `YYYY/MM/DD`. K formatování lze použít integrovanou funkci jazyka Python:

```
str_date = "{:%Y/%m/%d}".format(date)
```

Pro zjištění typu dat, které je potřeba najít jsem použil datový katalog, který přesně určuje pro která data je určena která složka v systému. Z katalogu jsem vyčetl, že kategorii, kterou potřebuji je `60.1_1` (viz. sloupeček HDFS fodler na obrázku 3.1).

Pro načtení dat, bylo potom nutné zavolat:

```
gm = spark.catalogData("60.1_1")
```

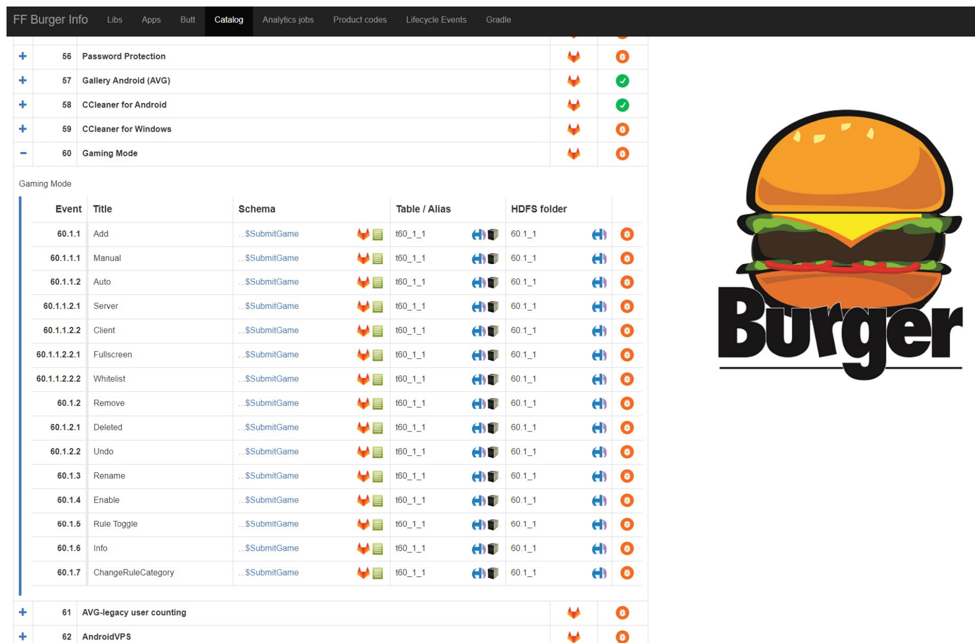
. Po tomto zavolání bude proměnná `gm` obsahovat instanci objektu |DataFrame|, která obsahuje odkaz na uložená data v systému.

Ke kontrole, že se nějaká data načetla je možné použít funkce `show`. Ta zobrazí prvních 20 (defaultní nastavení ve Sparku[21]) řádků tabulky jako textovou ascii tabulku. Tato tabulka je nepřehledná a pro přesnou kontrolu dat je nepoužitelná. Pro lepší a přehlednější zobrazení dat lze zavolat:

```
gm.limit(50).toPandas()
```

Nejdříve limitujeme data na prvních 50 prvků a potom následně konvertujeme instanci objektu `DataFrame` na strukturu z frameworku Pandas, kterou umí Jupyter překresil do hezké tabulky automaticky. Ovšem je nutné limitovat vypsání pouze na prvních 50 prvků, jelikož pokud zvolíme větší hodnotu (nebo vůbec nelimitujeme počet prvků), tak Jupyter vypíše nejdříve prvních 25 prvků tabulky. Potom vypíše tři tečky a následně posledních 25 prvků tabulky. Což nemusí být záměrem (většina mých příkladů).

Po první dobrém zobrazení, jsem věděl, že se mi podařilo nějaká data načíst a je tedy možné je prozatím zpracovávat.



Obrázek 3.1: Katalog datových kategorií systému Burger

3.2 Sčítání příchozích dat

V první části jsem se rozhodl, že nejdříve získám počet manuálních přidání her do Gaming Mode komponenty. K získání takovýchto dat jsem potřeboval zjistit typ odesílané události. K tomu mi opět posloužil katalog dat a v něm jsem našel, že manuální přidání má typ `60.1.1.1`(viz. obrázek 3.1). Tuto informaci bylo nutné porovnat se sloupečkem `event.type`. Ten ovšem obsahoval datový typ pole a ne textový řetězec, se kterým jsem počítal. Jelikož Spark neumí porovnávat dvě různá pole, zda obsahují stejné prvky, rozhodl jsem se, že hodnotu ze sloupečku změním na řetězec. K tomu mi poslouží systém uživatelsky definovatelných funkcí. Díky tomu jsem schopen pro každý prvek spustit funkci nad daným sloupečkem a následně změnit jeho hodnotu. Vytvořil jsem tudíž funkce:

```
def event_to_string(event_array):
    mp = map(lambda ai: str(ai), event_array)
    return ".".join(list(mp))

event_to_string_udf = udf(event_to_string)
```

Funkcí `udf` jsem konvertoval obyčejnou funkci jazyka Python na funkci, kterou je možné ve Sparku spustit. V normálním případě je nutné kromě funkce definovat i návratový typ (jelikož to nelze poznat z hlavičky funkce v jazyce Python), ale jelikož je `StringType()` je defaultní hodnota, tak to v tomto případě není nutné. Tento příkaz je součástí balíčku

```
pyspark.sql.functions
```

.

Jak již bylo zmíněno v kapitole 2.2, tak je nutné kotrolovat, zda někteří uživatelé se nesnaží ovlivnit data neustálým odesíláním jedné a té zprávy dokola. Proto se pro každou řádku bude počítat pouze první řádek s výskytem unikátní hodnoty ve sloupečku `identity.hwid`, který obsahuje systémové hwid počítače z kterého byla data odeslána.

Dále bylo nutné získat id aplikace, pro kterou je nutné data sečíst. Tato informace se nachází ve sloupečku `submitGame.submit_entry.application.id`. Tuto informaci jsem zjistil z datového diagramu, který je vysvětlen v kapitole 2.1.2

Nyní jsem byl schopen plně sestavit dotaz nad daty pomocí tohoto příkazu:

```
manual_adds = gm.where(
        event_to_string_udf(col("event.type")) == "60.1.1.1")\
.select("submitGame.submit_entry.application.id",
        "identity.hwid").distinct()\
.groupBy("id").agg({"id" : "count"})\
.withColumnRenamed("count(id)", "add_count")
```

V něm je nejdříve testován sloupec `event.type` zda se rovná o správný typ události. Následně jsou z tabulky vybrány pouze 2 sloupečky (příkaz `select`) a jsou ponechány pouze jejich unikátní kombinace (příkaz `distinct`). Potom dojde k sečtení všech přidaní pro jednotlivá id. Nakonec je nutné přejmenovat vzniklý sloupec se součtem dat, protože Spark má problémy s používání závorek v názvu sloupečku při použití příkazu `select`, protože v tomto okamžiku Spark parsuje klasické SQL a předpokládá, že jde o volání funkce a jelikož žádná taková fuknce pod tímto jménem neexistuje, tak dojde k vyvolání výjímky.

V této ukázce je také krásně vidět, jak Spark používá názvy method podobné SQL příkazům.

Při počítání odebrání z Gaming Mode seznamu jsem si počínal naprosto stejně jenom jsem adekvátně změnil hodnoty a názvy proměnných a parametrů.

Po získání jednotlivých dat jsem se rozhodl, že data vložím do jedné společné tabulky. To se provede příkazem `join`. Jelikož chceme zachovat i data která se nemusí nacházet v obou tabulkách je nutné tedy provést tzv. `FULL JOIN`. Ten se provede přidání parametru do funkce `join`. Jelikož ovšem

Spark automaticky nedoplňuje spojené tabulky o defaultní hodnoty typu sloupečků, je nutné ještě před spojení přejmenovat id jednotlivých tabulek. To jsem provedl opět příkazem `withColumnRenamed`.

S těmito daty jsem potom mohl spustit následující dotaz.

```
counts = manual_adds.join(deletes,
                          col("deletes_id") == col("adds_id"),
                          "full")\
.withColumn("counts_id", F.coalesce(col("changes_and_adds_id"),
                                    col("deletes_id")))\
.fillna(0, ["add_count", "game_count"])\
.select("counts_id", "add_count", "delete_count")
```

V tomto příkazu dojde nejprve ke spojení tabulek a to pomocí referencí na sloupečky s id, které vytvořím příkazem `col`. Následně je vytvořen nový sloupeček novým id. Ten je vytvořen spojením řádků z obou tabulek (příkaz `coalesce`). V tomto kontextu `F` je reference na balíček `pyspark.sql.functions`. Následně jsou pomocí metody `fillna` naplněny prázdné hodnoty[11] hodnotou `0`. Nakonec jsou vybrány pouze sloupečky, které jsou potřeba.

Stejným způsobem jsem přidal výpočet součtu všech druhů přidání aplikací.

Tímto jsem získal výsledná data, díky kterým jsem byl schopen vypočítat kategorii.

3.3 Přiřazení kategorie

Pro určení kategorie jsem se opět rozhodl použít uživatelsky definované funkce. Je sice pravda, že algoritmus je tak jednoduchý, že by šlo výsledku docílit kombinací řádků jednotlivých příkazů, které by šlo vypočítat samostatně pro každou kategorii, ale přišlo mi, že pokud použiji nejdříve mnou definovanou funkci, tak udělám v algoritmu menší množství chyb.

V nadcházející ukázce jsou použita čísla, která reprezentují konstany pro výpočet algoritmu. V kapitole 3.6 jsou tyto nahrazeny globalními proměnnými, aby byla konfigurace skriptu jednoduší.

```
def pick_category(add_count, all_adds_count,
                  delete_count, actual_category):
    if add_count + delete_count < 25:
        return "NONE"
    if actual_category is None or actual_category == "NONE":
        if add_count > delete_count:
```

[11] None/NULL/...

```
            return "GAME"
        else:
            return "GENERAL"
    if actual_category == "GENERAL":
        if add_count * 0.5 > delete_count
           or max(all_adds_count, add_count) * 0.05 > delete_count:
            return "GAME"
    else:
        if delete_count * 0.5 > add_count
           and delete_count > max(all_adds_count, add_count) * 0.05:
            return "GENERAL"
    return actual_category
```

Pro spuštění výpočtu potom slouží ve skriptu:

```
category = total_new.withColumn('category',
                                category_criterion(
                                    col('add_count'),
                                    col('all_adds_count'),
                                    col('delete_count'),
                                    None))
```

Kde `category_criterion` je proměnná, ve které je uložena transformovaná funkce `pick_category`.

3.4 Ukládání

V této fázi mám data, která obsahují kategorii aplikace. Nyní je nutné jenom uložit na disk. K ukládání dat existuje speciální datová struktura, která je ovšem součástí každé instance struktury `DataFrame`. Proto k uložení dat stačí zavolat jednoduše
`<instance_DataFrame>.write.<typ_souboru>(<umístění dat>)`. Tedy použil jsem k ukládání tuto funkci:

```
category.write.parquet(
    "/projects/gaming_mode/adds_and_removes/category")
```

Pro testovací účely jsem ovšem nepoužíval složku `projects`, ale svou uživatelskou složku, tedy `/users/tomas.vybiral/`.

K uložení jsem se rozhodl použít stejnou datovou strukturu, ve které jsou uložena načítaná data, tedy příkaz `parquet`.

V tomto okamžiku se ukládal každý řádek jako samostatný soubor, což není optimální pro celkovou velikost dat na disku, jelikož HDFS je uzpůsoben v naší konfiguraci na větší soubory. Toto se dá ovšem napravit jednoduchou augmentací příkazu a to na tuto verzi:

```
category.coalesce(1)\
        .write.option("maxRecordsPerFile", 10000)\
        .parquet("/projects/gaming_mode/adds_and_removes/category")
```

Příkaz `coalesce` v tomto případě interně nastaví, že se všechny řádky mají chovat jako jednotný celek. Pokud bychom použili pouze tento příkaz, tak se může stát, že skript začne padat, protože Spark by se při ukládání snažil načíst veškerá data do paměti a v případě, že se jedná o velké množství řádků, hrozí že procesu dojde paměť. Proto je ještě nutné nastavit maximální počet řádků (záznamů). V případě, že je tento limit je překonán, tak Spark začne data zapisovat do dalšího souboru. K nastavení této hodnoty slouží v kódu část `.write.option("maxRecordsPerFile", 10000)`.

3.5 Načítání starých dat

V tomto stavu se data pouze načtou, zpracují a následně přepíšou staré výsledky. Ovšem bylo by dobré použít stará data k připočítání k nově sečteným a kategorii spočítat ze společných dat.

Datový soubor jsem načetl pomocí příkazu:

```
total = sqlContext.read.parquet(
            "/projects/gaming_mode/adds_and_removes/category")
```

Proměnná `sqlContext` je globální proměnná, která je přímo vytvořena prostředím Spark.

Po sečtení data z minulého dne se nově data spojí (příkaz `join`) a jednotlivé výsledky sečtou. Pro každé 2 sloupečky, které se mají sečíst jsem vytvořil nový sloupeček ve které se výsledný součet nachází. Jelikož jsem opět použil `FULL JOIN`, tak jsem musel opět doplnit prázdné hodnoty některých řádků.

Při ukládání dat ovšem nastal problém, jelikož Spark poznal, že původní soubor z daty je otevřený, tak nebylo možné jej přepsat. To jsem ovšem vyřešil pouhým přejmenováním souboru a následným odstraněním původního souboru a přejmenování nového souboru na jméno původního. K tomu jsem použil další část interní knihovny, která umožňuje jednoduchou kontrolu a manipulaci se soubory.

Zároveň jsem si v tuto chvíli uvědomil, že by nebylo špatné vytvořit si nejakou datovou zálohu pro případ, že by skript selhal a poškodil by tím nová data. Toho jsem opět docílil jednoduchou manipulací s výslednými soubory (mám uloženo vícero souborů a označuji si starší data). V aktuálním stavu zálohuji pouze jeden den zpět, ale rád bych tuto hodnotu navýšil alespoň na pár dní.

3.6 Nastavitelné konstanty

V tomto stavu skript dělal přesně to, co jsem očekával. Ovšem pokud jsem chtěl udělat libovolnou změnu v hodnotách, tak jsem musel poctivě procházet celý kód kontrolovat každou změnu.

Proto jsem se rozhodl, že spoustu kódu udělám nastavitelným a většinu hodnot přesunu do proměnných na začátek kódu. Tyto hodnoty nejsou potom v celém kódu měněny, lze je tedy považovat za konstanty.

Touto změnou jsem výsledný kód hodně zpřehlednil a zjednodušilo se ladění jednotlivých hodnot pro výpočty.

3.7 Bezpečnost

Jak jsem se již snažil nastínit v kapitole 3.5, snažil jsem se skrpit upravit, tak aby v případě chyby, nedošlo ke ztrátě dat. Proto jsem se rozhodl, že v případě, že dojde k chybě a data nejsou uložena, tak se skript sám pokusí o načtení zazálohovaných dat. Nově se skript pokusí načíst z data ze zazalohované verze, pokud novější neexistuje a pokud data vůbec neexistují, tak se vytvoří nová prázdná datová struktura (toto rozhodnutí jsem učinil co nám na testovacím prostředí začali úmyslně pravidelně mazat veškerá data).

Následně pokud některé sloupečky v načtených datech neexistují, tak jsou vytvořeny a doplněny defaultními hodnotami.

Zároveň mi přišlo jako dobrý nápad, v případě že se minulé spuštění programu nepovedlo, tak se pokusit znovu počítat data o která jsem v tom dnu přišel. Proto jsem uvedl změnu, která umožňuje načíst data i z neúspěšných dnů. Zda se podařilo či nepodařilo zjištuji posledním dnem kdy došlo ke změně na uložených datech. K této kontrole ovšem jsem nemohl použít interní knihovnu, protože tuto funkci neimplementuje. Nahlédl jsem proto do implementace této knihovny a zjistil jsem, že pro práci se soubory používá program `hdfs`. Z dokumentace jsem vyčetl i že tento program dokáže přesně to co jsem potřeboval a proto jsem se ho rozhodl využít. K tomu ve skriptu používám knihovnu `subprocess` a funkci `check_output`, která spouští proces a vrací jako výstup textový řetězec, který obsahuje výstup ze spuštěného procesu. Ve skriptu jsem si vytvořil novou funkci:

```
def get_last_updated_date(path):
    if file_utils.exists(path):
        res = re.search(
            r"^(?P<year>[0-9]*)-(?P<month>[0-9]*)-(?P<day>[0-9]*)",
            subprocess.check_output(
                ["hdfs", "dfs", "-stat", "\%y", path])\
                .decode("utf-8"))
        return date(year=int(res.group("year")),
                    month=int(res.group("month")),
```

```
                    day=int(res.group("day")))
    return date.today() - ONE_DAY
```

Tato funkce získá chtěné datum za pomoci výše zmíněné funkce a regulárních výrazů. Tato funkce vrací správné datum (nebo včerejší datum, pokud soubor neexistuje) ve struktuře `date` z knihovny `datetime`.

Následně jsem musel upravit získávání nových nezpracovaných dat. Samozřejmě jsem k tomu mohl použít v kapitole 3.1 zmíněnou funkci `catalogData`, které se dát definovat jaká data se mají načíst. Ovšem v firemní dokumentaci systému Burger jsem se dočetl, že je lepší načíst jednotlivé dny samostatně a potom je pomocí funkce `join` sjednotit, protože Spark je potom schopen lépe paralelizovat. Naimplementoval jsem tedy obě řešení a na testovacích datech jsem vyzkoušel jaký (pokud vůbec nějaký) rozdíl v čase tato změna udělá. Provedl jsem několik měření v průběhu dne (abych vyzkoušel různá zatížení systému Burger) a ve všech případech jsem viděl velké zrychlení a to až o polovinu času kratší běh. Toto řešení bylo sice náročnější na implementaci, ale rychlejší algoritmus se mi líbil víc. Proto jsou jednotlivá data získávána pomoci smyčky, která sbírá data z každého dne a postupně je spojuje dohromady.

3.8 Nasazení

Po několika měsíčním testování přišel čas na nasazení skriptu a jeho automatizaci. Z Jupyteru lze vyexportovat spustitelný Python skript, který jsem použil ke spouštění výpočtů. K nasazení jsem využil prostředí Azkaban, kde jsem jednoduše vytvořil projekt `gaming_mode` a za pomoci týmu, který zpravuje systém Burger, se mi podařilo nastavit, aby skript byl spouštěn pravidelně.

Pro plnou fukčnost systému bylo týmem, který spravuje systém Burger, ještě vytvořeno REST API, pomocí kterého si antivirus dokáže získat informace o danné aplikaci, kterou konkrétně v tom čase testuje. Ovšem toto REST API nezískáva data z mnou uložené databáze, ale byl vytvořen další automatizovaný program, který moji databázi překonvertuje a zkopíruje na interní úložiště serveru, kde je REST API nainstalované.

Zároveň bych chtěl v této části zmínit, jak byla upravena automatická detekce aplikací v komponentě Gaming Mode, aby využívala mnou vytvořenou databázi. Nově si Gaming Mode vypočítá id aplikaci jiži na začátku procesu přidávání a potom co zkontroluje aplikaci vůči interní databázi, zkontroluje aplikaci pomocí dotazu na REST API. Podle odpovědi potom komponenta buď aplikaci přidá jako hru, zruší proces přidávání nebo bude v tomto procesu nadále pokračuje.

Kapitola 4

Testování

V této kapitole se zaměřím na metody, které jsem použil k testování skriptu. Zároveň se na začátku zmíním jaké problémy jsem musel řešit. V závěru se ještě rozepíši o analýze příchozích dat a jaké nástroje jsem k tomu převážně používal. Zároveň také chci povídat jaké chyby se mi podařilo díky této analýze nalézt v komponentě Gaming Mode.

4.1 Řesené problémy

Při tvorbě testovacích nástrojů a při testováním samotném jsem řešil dva problémy.

Prvním problémem bylo, jak data přehledně zobrazovat. Jelikož jak jsem již zmínil v kapitole3.1, tak výpis tabulky funkcí `show` je velice nepřehledný. V té kapitole také zmiňuji, že lze použít trik, kterým data překonvertuji na datovou strukturu z frameworku `Pandas`. Ten ovšem umožnuje vypsání prvních 50 prvků tabulky a neumožnuje vypsat více dat, což by mohl v některých výpisech využít.

V dokumentaci jsem objevil, že v Jupyter noteboocích lze vykreslovat HTML jako výstup jednotlivých buňek[22]. Tento způsob byl již součástí původního systému IPython (proto se používá knihovna `IPython.display` a z ní fukce `HTML` v kombinaci s fukcí `display`). Rozhodl jsem se tedy využít tento způsob vyžít, a že si napíši generátor HTML tabulek, který bude mít na vstupu data a počet řádků na kolik se má tato tabulka vykreslit.

K některým analýzám jsem použil vykreslování grafů za pomocí knihovny `matplotlib`, která je integrována i v rámci Jupyteru.

Druhým problémem bylo kopírování kódu. Abych mohl použít v jiném notebooku některé části jiného kódu musel jsem je (alespoň ze začátku) kopírovat, jelikož Jupyter Notebook je ukládán jako JSON, který neumí Python otevřít jako modul. Ovšem v oficiální dokumentaci jsem objevil část kódu, která registruje vlastní načítací skript pro Jupyter Notebook. Problém se vy-

skytl v tom, že tento kód byl dlouhý a musel se spouštět před začátkem každého skriptu. Brzy jsem ovšem ve firemní dokumentaci našel zmínku o tom že lze Jupyteru říci aby vzal libolný soubor a načetl ho jako část kódu (použil stejný interpret jako pro zbytek notebooku). Tedy jsem kód pro importování přesunul do souboru `loader.py` a následně jsem přidal na začátek každého notebooku, který potřebuje načítat jiné notebooky, toto:

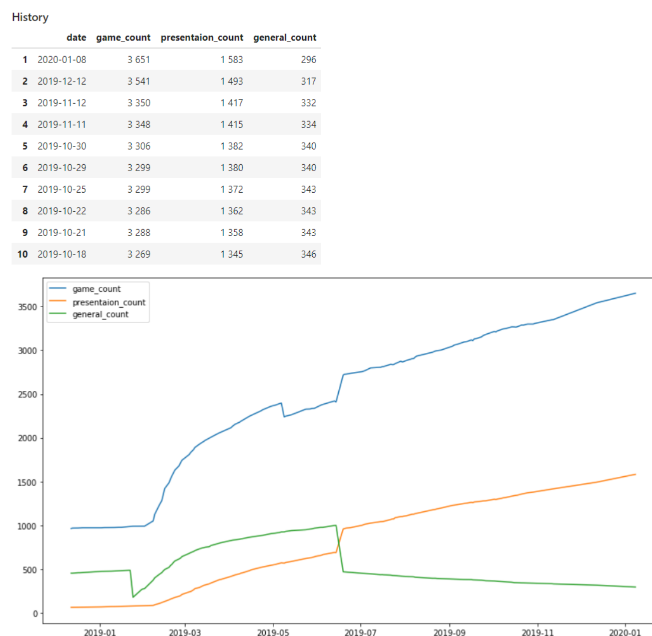
```
\%run -i loader.py
```

Bohužel tento způsob fungoval pouze při prvním spuštění jednotlivých buňek. Pokud jsem chtěl spustit některou buňku vícekrát, tak jsem musel nechat restartovat jádro, které se spustilo pro daný notebook. Zároveň jsem si všiml, že nefungovalo používání uživatelsky definovaných funkcí z takto importovaných modulů. Spark totiž vyžaduje, aby se případné takovéto moduly vždy registrovali do jeho interního systému. Bohužel pro vstup do této funkce chtěl cestu k souboru s modulem. Rozhodl jsem se tedy, že pomocí kódu, který parsuje notebooky, budu ukládat výsledný Python kód do samostatného souboru. Ten potom zaregistruji do Sparku. Tímto se také vyřešil způsob, že nešli spouštět buňky více než jednou. Při druhém spuštění buňky se jednoduše znovu načte ze souboru, který se vygeneroval. Bohužel to znamená, že pokaždé když udělám změnu v notebooku, tak musím vygenerovaný soubor smazat, protože Pythoní interpret by ho našel dřív než by se pokusil zparsovat příslušný notebook. Zároveň mi přišlo zbytečné načítat každou buňku, tak jsem kód upravil tak aby načítal pouze buňky které mají v metadatech buňky tag `importable`. Tyto metadata se dají upravovat přímo v editoru Jupyter.

V důsledku této implementace jsem se rozhodl, že načítat se budou pouze dva notebooky. Ten který obsahuje mnou vytvořený agregační skript (`backend_calculation.ipynb`) a potom druhý (`my_utils.ipynb`), který obsahuje nejpoužívanější funkce, které jsem použil k výpočtu dat. Zároveň také obsahuje mojí funkci, která vypíše pro daná data hezkou tabulku. Také bych chtěl zmínit, že tato malá knihovna obsahuje funkci, která načte data, přidá k ním data z dnešního dne a potom našledně vytvoří graf všech historických dat a následně nová data uloží.

4.2 Kontrola dat

Už během vývoje jsem často používal jako kontrolu dat to že jsem si data zobrazoval v tabulkách a potom následně kontroloval jestli se dělo to co má. Bohužel do výsledného skriptu nebylo možné tyto části zařadit, jelikož skript potom následně začal padat a množství dotazů nad spočítanými daty se zvýšil tak že jich bylo víc než samotného kódu, který počítá výsledná data. Rozhodl jsem se proto tento skript rozdělit na dva. Jeden pouze počítá data a druhý slouží k zobrazování jednotlivých dotazů nad daty.



History

	date	game_count	presentaion_count	general_count
1	2020-01-08	3 651	1 583	296
2	2019-12-12	3 541	1 493	317
3	2019-11-12	3 350	1 417	332
4	2019-11-11	3 348	1 415	334
5	2019-10-30	3 306	1 382	340
6	2019-10-29	3 299	1 380	340
7	2019-10-25	3 299	1 372	343
8	2019-10-22	3 286	1 362	343
9	2019-10-21	3 288	1 358	343
10	2019-10-18	3 269	1 345	346

Obrázek 4.1: Ukazka časového průběhu počtu kategorizovaných aplikací (aktuální verze, proto obsahuje i sloupeček presentation_count, který ovšem je důležitý pouze pro komponentu Do Not Disturb))

Skript, který zobrazuje pracidelně dotazy nad daty se jmenuje `GameModeNewBackendProgress.ipynb`. Tento skript nejdříve načte stará data a potom následně spustí různé dotazy nad daty a vypíše tabulky s výsledky (každá tabulka jeden dotaz). V tuto chvíli jsem rozhodl že by bylo dobré si také ukládat nejčastější jméno aplikace, kterou posíláme a také že by bylo dobré u jednotlivých řádků zobrazovat změnu, která proběhla za daný den. Proto jsem v agregačním skriptu přidal získávání nejpoužívanějšího jména a nově tam také ukládám výpočty za minulý den. Zároveň tento skript také počítá počet aplikacích, kterým byla přiřazena kategorie `GAME` a `GENERAL`. Tato data jsou potom zobrazena v grafu, který slouží, v případě výkyvů, k detekci chyb v kategorizačním algoritmu (viz obrazek 4.1).

Postupem času jsem z tohoto skriptu odstranil zbytečné dotazy a tudíž nyní obsahuje jenom mnou nejčastěji používané a chtěné dotazy. Mezi tyto dotazy například patří: nejvíce ručně přidávané hry, nejvíce mazané aplikace, nejvíce celkově přidávané hry, ... Jako poslední bych rád zmínil poslední dotaz v tomto skriptu, který zobrazí náhodně řádky ze všech dat. Tento dotaz vznikl poté co jsem si uvědomil, že údaje v horních částech tabulek se většinou nemění a jenom občas dojde k změnám v dolních částech tabulky. Tento dotaz mi potom umožnil náhled i do ostatních částí dat.

Pro monitorování příchozích dat jsem si vytvořil skript

`progress_in_time.ipynb`, který načítá různé údaje o předchozích datech ke komponentě Gaming Mode. Každá kategorie má svů vlastní účel jako například: počet denních uživatelů, počet denních přidání (rozdělené podle druhu přidání), ... Veškerá data jsou potom zobrazena jako grafy, které ukazují časový průběh jednotlivých údajů. To mi ukazuje jakékoliv prudké výkivy v datech, které mohou být poteciální problémy.

4.3 Vyhledávání v datech

Ve všech tabulkách, které jsem zobrazoval byla vždy vidět jenom data, která jsem vypočítal a nebylo možné zjistit o danné aplikaci více informací dokud jsem si nenapsal speciální dotaz nad těmito daty. Z tohoto důvodu vznikl skript `searching_utilities.ipynb`. Tento skript původně obsahoval spoustu jednoduše konfigurovatelných dotazů. Ze začátku toto bylo zvladatelné, ale jakmile začalo druhů dotazů přibýcat, tak se skript stal nepřehledným. Proto jsem rozhodl, že skript předělám. V současném stavu se dělí na dvě části. Zároveň bych chtěl zmínit, že skript pracuje většinou s daty z předchozích dvou týdnů.

V první části je uživatel schopen vyhledávat v databázi jednotlivá data podle id. Ve druhé části skript vezme libolný textový řetězec a podle něho dokáže v datech vyhledávat. Může se například jednat o vyhledávání pomocí jména aplikace nebo naźvu spustitelného souboru aplikace.

K zpřehlednění jsem využil knihovny `ipywidgets`, která umožnuje v Jupyter Notebook vytvářet jednoduché prvky uživatelského rozhraní a ovládat je pomocí jazyka Python. Díky tomu jsem každému dotazu mohl vytvořit speciální tlačítko s popiskem jeho funcionality. Každý dotaz si vezme důležitá data z vstupního pole, které musí být před stistknutím tlačítka vyplněno (ukázka na obrázku 4.2).

4.4 Analýza dat

Po nasazení skriptu a spouštění ověřování jsem se chtěl zjistit jak se skript bude chovat po tom co budou přístupná data od všecdních uživatelů. Bohužel hned na začátku jsem zjistil, že nám skoro vůbec nepřicházejí žádná data, která o tom, že nějaká aplikace byla přidána pomocí kontroly ze serveru. Toto mi přišlo divné, jelikož skript už byl několik měsíců nasazen a já sám jsem viděl již některým aplikacím přiřazenou kategorii. Zároveň se nám ozvali, že na REST API nechodí moc dotazů a že se jim to také nezdá. Nakonec se ukázalo, že byl pro server, na kterém běželo REST API, špatně nastavený firewall a tedy žádné dotazy, které nebyli z interní sítě, se nemohli na server dostat.

Po nasazení skriptu a následné aktivace detekce podle výsledných dat jsem z nejčastěji ručně přidávaných aplikací vyčetl, že jednou z nejčastěji přidáva-

Hash: 6140b271179559e39af9315af073

Calc adds | Show names | Show paths to exe file | Show events for id | Search in backend ... | Count add duplicities

All adds for 6140b271179559e39af9315af073e239378966f9

	id	executable_name	detection_version	publisher	name	major	minor	total_adds	whitelist_adds	server_adds	fullscreen_adds	manual_adds
1	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	8	53 896	53 881	0	15	0
2	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	4	2 362	2 330	0	32	0
3	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	5	2 132	2 105	0	27	0
4	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	2	1 304	1 286	0	18	0
5	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	6	1 250	1 238	0	12	0
6	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	3	1 243	1 216	0	27	0
7	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	7	1 007	1 000	0	6	1
8	6140b271179559e39af9315af073e239378966f9	chrome.exe	13		Google Chrome	19	5	950	950	0	0	0
9	6140b271179559e39af9315af073e239378966f9	chrome.exe	14		Google Chrome	19	6	754	753	0	1	0
10	6140b271179559e39af9315af073e239378966f9	chrome.exe	12		Google Chrome	19	4	746	745	0	1	0
11	6140b271179559e39af9315af073e239378966f9	chrome.exe	13		Google Chrome	19	4	707	707	0	0	0
12	6140b271179559e39af9315af073e239378966f9	chrome.exe	14		Google Chrome	19	5	529	523	0	6	0
13	6140b271179559e39af9315af073e239378966f9	chrome.exe	11		Google Chrome	19	3	511	511	0	0	0
14	6140b271179559e39af9315af073e239378966f9	chrome.exe	14		Google Chrome	19	7	384	384	0	0	0
15	6140b271179559e39af9315af073e239378966f9	chrome.exe	8		Google Chrome	19	2	340	340	0	0	0
16	6140b271179559e39af9315af073e239378966f9	chrome.exe	12		Google Chrome	19	3	323	322	0	1	0
17	6140b271179559e39af9315af073e239378966f9	chrome.exe	11		Google Chrome	19	2	279	276	0	3	0
18	6140b271179559e39af9315af073e239378966f9	chrome.exe	14		Google Chrome	19	4	146	141	0	5	0
19	6140b271179559e39af9315af073e239378966f9	chrome.exe	13		Google Chrome	19	3	114	110	0	4	0
20	6140b271179559e39af9315af073e239378966f9	chrome.exe	15		Google Chrome	19	9	100	100	0	0	0
21	6140b271179559e39af9315af073e239378966f9	chrome.exe	12		Google Chrome	19	2	94	93	0	1	0
22	6140b271179559e39af9315af073e239378966f9	chrome.exe	13		Google Chrome	19	2	90	89	0	1	0
23	6140b271179559e39af9315af073e239378966f9	chrome.exe	14		Google Chrome	19	2	85	85	0	0	0

Obrázek 4.2: Ukázka vyhledávání pomocí vyhledávacích nástrojů (vyhledávám podle id webového prohlížeče Google Chrome)

nou hrou je webový prohlížeč Google Chrome. Lidé používali Gaming Mode i na jiné aplikace než na hry. Proto došlo k přepracování komponenty Gaming Mode na komponentu Do Not Disturb. Do této komponenty byla přidána podpora pro neherní aplikace. Mezi tyto aplikace můžeme počítat webové prohlížeče, video přehrávače, prezentační aplikace, atd.

Dalším problémem, které ho jsem si všiml bylo, že aplikace často měnili přiřazené id a proto byli uživatelé obtěžováni znovu objevujícíme si okny o přidání hry do seznamu. Toto se ukázalo, že jse způsobeno tím, že některé aplikace uvádějí v odinstalačním klíči i verzi svého produktu a pokud dojde k aktualizaci aplikace, tak tato aplikace se znovu zdetekuje. Tento problém nakonec vývojáři komponenty opravili tím, že se aktuálně komponenta snaží odstranit verzi z odinstalačního klíče aplikace.

Za zmínku ještě stojí problém, kdy se pod jedním id nacházelo vícero aplikací. Toto je zapříčeněno špatným párováním aplikací s odinstalačními klíči. Ovšem problém nemusí být pouze v tomto případě. Je také možné, že aplikace, která se detekovala, nemá odinstalační klíč a komponenta spáruje aplikaci s cizím klíčem. Vůči tomuto chování byl vývojáři komponenty přidán nový test, kde se id aplikace testuje proti známému seznamu. Pokud dojde k pozitivní detekci, tak je aplikace přidána bez id. Toto ovšem není optimální řešení, protože toto se může stát pro jakoukoliv aplikaci. Proto se tato oprava používá pouze v případech, kdy jsme si jisti, že se jedná opravu o id neherní aplikace (např. `eplorer.exe`, …).

Závěr

Cílem této práce bylo vytvořit automatický systém pro kategorizaci aplikací, který ke své činnosti bude používat data získaná od uživatelů pro komponentu Gaming Mode.

Čtenář mohl zjistit, jak komponenta Gaming Mode funguje. Dozvěděl se, jak dochází k detekci aplikací a jak může uživatel ovlivnit při jakých aplikacích se budou aktivovat prvky komponenty Gaming Mode, které snižují dopad na výkon počítače. Zmínil jsem, také jak funguje aktivace prvků, které zlepšují hráčův zážitek ze hry, a také jaké prvky se přesně spouštějí a co dělají. Zároveň jsem pospal, jakým způsobem dochází ke sbírání uživatelských dat. Touto částí jsem splinil jeden ze svých dílčích úkolů.

V další části jsem se potom zaměřil na celý systém, kam se data odesílají a v jakém formátu jsou odesílána. Následně jsem popsal celý systém, na kterém jsou data uložena, a jak je možné s těmito daty na tomto systému pracovat. Uvedl jsem také výhody a nevýhody jednotlivých možností a následně jsem popsal jedno z webových vývojových prostředí, které jsem potom následně použil pro vývoj svého projektu. Také jsem se v jedné části zaměřil na specifikaci jedné z knihoven, kterou jsem při vývoji použil a která byla vyvinuta přímo ve firmě pro interní účely. Tímto jsem také splnil druhý ze svých dílčích úkolů.

Dalším cílem bylo navrhnout a implementovat řešení problému. V této části jsem se kromě těchto bodů také zaměřil na popis odesílané datové struktury a popis způsobu definice struktur Google Protocol Buffer a použití v kontextu komponenty Gaming Mode. V návrhové části jsem potom nastínil zamýšlený průběh celého skriptu. Následně v implementační části této práce jsem podrobně rozebral vývoj zamýšleného kódu a jednotlivé kroky, které jsem učinil. Zároveň jsem rozebral použití různých prvků systému Spark.

Poslední cíl bylo testovat mé řešení a následně použít data, která jsem získal od uživatelů k analýze funkčnosti komponenty Gaming Mode. V této části jsem vyřešil problém importovatelnosti Jupyter Notebook. Dále jsem popsal vytvoření testovacích nástrojů a vytvořil jsem krátký popis o tom, co

jednotlivé nástroje dělají. V úplně poslední části jsem popsal, jakým způsobem jsem využil tyto testovací nástroje k analýze dat, a zmiňuji některé problémy, které jsem objevil díky těmto nástrojům, a jak je týmy v Avastu řešily.

Splněním všech dílčích úkolů, které jsem si vytvořil, jsem zároveň splnil svůj hlavní cíl.

Tato práce mi umožnila si zvýšit znalost o databázových systémech. Hodně jsem se dozvěděl o NoSQL databázích a jak fungují. Také jsem si uvědomil, co je všechno možné, pokud člověk dokáže vytvořit správný dotaz nad daty.

Příloha **A**

Použitá literatura a zdroje

1. Vznikne antivirový gigant, pražský Avast kupuje původně brněnskou AVG. *iDnes*. Dostupné také z: https://www.idnes.cz/ekonomika/domaci/ceska-antivirova-firma-avast-kupuje-rivala-avg-za-32-miliard-korun.A160707_101955_ekonomika_lve.
2. Bitdefender Game Mode – How to configure. *BitDefender* [online] [cit. 2019-12-10]. Dostupné z: https://bdantivirus.com/bitdefender-game-mode/.
3. Wildcards in Windows. *Jeremy Kuhne's Blog*. Dostupné také z: https://blogs.msdn.microsoft.com/jeremykuhne/2017/06/04/wildcards-in-windows/.
4. *Protocol Buffers | Google Developers* [online]. Google Developers [cit. 2019-12-10]. Dostupné z: https://developers.google.com/protocol-buffers.
5. API Reference | Protocol Buffers. *Google Developers*. Dostupné také z: https://developers.google.com/protocol-buffers/docs/reference/overview.
6. *Apache Hadoop* [online]. The Apache Software Foundation, © 2006-2019 [cit. 2019-12-10]. Dostupné z: https://hadoop.apache.org/.
7. HDFS Architecture Guide. *Hadoop Documentation* [online] [cit. 2019-12-10]. Dostupné z: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
8. *What is MapReduce? How it Works - Hadoop MapReduce Tutorial*. Guru99, © 2019. Dostupné také z: https://www.guru99.com/introduction-to-mapreduce.html.
9. Apache Hadoop YARN. *Hadoop Documentation*. Dostupné také z: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

10. *Apache Kafka*. Apache Software Foundation, © 2017. Dostupné také z: https://kafka.apache.org/.
11. *Apache Cassandra*. Apache Software Foundation, © 2016. Dostupné také z: http://cassandra.apache.org/.
12. *Azkaban* [online]. Azkaban [cit. 2019-12-10]. Dostupné z: https://azkaban.github.io/.
13. *Hue - The open source SQL Assistant for Data Warehouses*. Cloudera, Inc., © 2019. Dostupné také z: https://gethue.com/.
14. *Apache Spark™ - Unified Analytics Engine for Big Data* [online]. The Apache Software Foundation, © 2018 [cit. 2019-12-10]. Dostupné z: https://spark.apache.org/.
15. *Project Jupyter | Home* [online]. Project Jupyter, © 2019 [cit. 2019-12-10]. Dostupné z: https://jupyter.org/.
16. *Jupyter and the future of IPython — IPython*. IPython development team. Dostupné také z: https://ipython.org/.
17. Jupyter kernels. *GitHub*. Dostupné také z: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.
18. MapReduce vs Apache Spark- 20 Useful Comparisons To Learn. *Educba* [online] [cit. 2020-01-08]. Dostupné z: https://www.educba.com/mapreduce-vs-apache-spark/.
19. Spark vs. Hadoop MapReduce: Which big data framework to choose. *ScienceSoft* [online] [cit. 2020-01-08]. Dostupné z: https://www.scnsoft.com/blog/spark-vs-hadoop-mapreduce.
20. Protocol Buffer Basics: Python. *Protocol Buffers* [online] [cit. 2020-01-08]. Dostupné z: https://developers.google.com/protocol-buffers/docs/pythontutorial#defining-your-protocol-format.
21. pyspark.sql module documentation. *PySpark 2.4.4 documentation* [online] [cit. 2020-01-08]. Dostupné z: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.show.
22. Module: display documentation. *IPython 7.11.1 documentation* [online] [cit. 2020-01-08]. Dostupné z: https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html#IPython.display.HTML.

Příloha B

Seznam použitých zkratek

GM Gaming Mode

GUI Graphical User Interface

AWS Amazon Web Sevices

HDFS Hadoop Distributed File System

JVM Java Virtual Machine

ssh Secure Shell

SQL Structured Query Language

RAM Random Access Memory

JSON JavaScript Object Notation

HTML Hypertext Markup Language

Příloha C

Obsah přiloženého média

Všechny soubory jsou zároveň uloženy na: `https://gitlab.com/Tomcus/bakalarska-prace-src`

```
 README.md ........................ popis adresářů ve formátu Markdown
 src
  ├─ impl .................................. zdrojové kódy implementace
  └─ thesis .................... zdrojová forma práce ve formátu LaTeX
 text ......................................................... text práce
  └─ thesis.pdf ........................... text práce ve formátu PDF
```