VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020 Petr Muzikant

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

VYUŽITÍ DATABÁZÍ PŘI ANALÝZE MALWARU REPREZENTOVANÉHO GRAFEM CHOVÁNÍ
USE OF DATABASES FOR THE ANALYSIS OF MALWARE REPRESENTED BY BEHAVIORAL GRAPH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
Petr Muzikant

VEDOUCÍ PRÁCE
SUPERVISOR
doc. Ing. Jan Hajný, Ph.D.

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Petr Muzikant ***ID:*** 200517
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Využití databází při analýze malwaru reprezentovaného grafem chování

POKYNY PRO VYPRACOVÁNÍ:

Práce je zaměřena na zpracování a ukládání grafových dat, které reprezentují stav procesů v operačním systému a interakce mezi nimi, jak na neinfikovaných, tak infikovaných zařízeních. Cílem bakalářské práce je implementace softwaru pro vyhledávání zajímavých podgrafů v těchto datech, jejich klasifikace pomocí interních systémů firmy Avast a jejich následný import do zvoleného databázového systému. Databázové řešení by mělo být zvoleno s ohledem na množství uchovávaných dat a umožňovalo rychlé grafové analytické dotazy. Výsledné řešení by mělo být otestováno v praxi v rámci interních systémů firmy Avast. Prototyp by měl zvládat extrakci grafových dat ze vstupních XML souborů, pročištění grafu od osamocených nebo nedosažitelných uzlů, vyhledání podgrafů v závislosti na uživatelsky definované metrice, klasifikaci takto vzniklých podgrafů pomocí interních systému firmy Avast.

DOPORUČENÁ LITERATURA:

[1] Basic Graph Algorithms [online]. [cit. 2019-10-02]. Dostupné z: http://jeffe.cs.illinois.edu/teaching/algorithms/book/05-graphs.pdf

[2] Angles, Renzo. (2012). A Comparison of Current Graph Database Models. Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012. 171-177. 10.1109/ICDEW.2012.31.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: doc. Ing. Jan Hajný, Ph.D.
Konzultant: Miroslav Drbal (Avast Software s.r.o.)

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Bakalářská práce se zabývá problematikou behaviorální detekce, její hlavní komponentou zvanou behaviorální štít a zpracováním telemetrických dat generovaných tímto štítem. Zvláštní pozornost je také věnována tématu vícevláknové aplikace, díky kterému bylo možné výsledek práce plně optimalizovat. Hlavním cílem práce je výběr databázového řešení (s ohledem na povahu dat) a implementace nástroje pro zpracování dat umožňující rozšiřitelnou filtraci. Práce je rozdělena na teoretické úvody do problematik a na dvě praktické části týkající se výběru databázového systému (na základě měření) a samotného nástroje (včetně jeho optimalizace). Před samotným závěrem je také přiložena krátká kapitola o problémech, které se během práce vyskytly a jejich řešení. Práce probíhala ve spolupráci se společností Avast Software s.r.o.

KLÍČOVÁ SLOVA

behaviorální detekce, behaviorální štít, databáze, graf, grafová, chování procesu, relační

ABSTRACT

This Bachelor's thesis deals with problematics of a behavioral detection, its main component called behavioral shield and processing a telemetry data flowing from this shield. Special attention is also paid to a multithreaded application since it was used to optimize the results. Main goal of the thesis is selection of the right database system given the nature of the data and implementation of a tool allowing data processsing and extensible filtering. Thesis is divided into theoretical introductions of mentioned problematics and into two practical parts about the selection of database system (based on measurements) and the tool for processing and filtering data (including its optimization). At the end is also included short chapter about the problems during the implementation with theirs solutions. Bachelors thesis took place in the cooperation with Avast Software s.r.o.

KEYWORDS

behavioral detection, behavioral shield, database, graph, process behaviour, relational

MUZIKANT, Petr. *Využití databází při analýze malwaru reprezentovaného grafem chování*. Brno, 2020, 106 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: prof. Ing. Jan Hajný, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Využití databází při analýze malwaru reprezentovaného grafem chování“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Moc rád bych poděkoval svému konzultantovi panu Ing. Miroslavu Drbalovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Také bych rád poděkoval vedoucímu práce panu doc. Ing. Janu Hajnému, Ph.D. za poskytnutí příležitosti vykonávat tuto práci ve spolupráci se společností Avast Software s.r.o. Poslední poděkování patří rodině, přítelkyni a přátelům za podporu a motivaci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Mezi způsoby detekce škodlivých aplikací se mimo jiné řadí také metody založené na monitorování a klasifikování chování programu přímo za jeho běhu – tzv. „behaviorální detekce“. Jejich součástí je komponenta zvaná „behaviorální štít“, která průběžně kontroluje běžící aplikace v systému a případně je na základě jejich škodlivého chování detekuje a označí jako malware (*malicious software* – škodlivý software). V rámci této kontroly probíhá také zaznamenávání telemetrických dat (především informací o procesech a jednotlivých interakcí mezi nimi), která slouží pro budoucí vývoj detekce (např. jako trénovací data pro strojové učení). Pro tyto účely je nutné data ukládat.

Obecně se k uchovávání informací nejčastěji využívají databáze. V dnešní době existuje několik různých modelů a jejich implementací, které se mezi sebou liší převážně v logice zpracování informací. Správnou volbou modelu a stanovením vhodné struktury uložených informací lze výrazně zrychlit operace vykonávané nad daty. Vzhledem k povaze telemetrických dat, která behaviorální štít generuje, je příhodné pro jejich ukládání využít model grafové databáze.

Cílem této práce je nejprve nastudovat problematiku grafových databázových systémů (v kontrastu s relačními) a zvolit několik jejich zástupců. Následně vytvořit nástroj pro zpracování dat, umožňující základní (ale i rozšiřitelnou) filtraci a převedení dat do formátu, ve kterém je bude možné importovat do zvolených databázových systémů. Poté je třeba provést měření na testovacích datech a podle předem zvolených metrik vybrat nejvhodnějšího zástupce grafové databáze. V závěrečné fázi se implementuje živé spojení nástroje s touto databází pro import dat v reálném čase.

Práce je rozdělena tematicky na úvod do problematiky, včetně teoretických částí o databázových řešení, behaviorální detekci a vícevláknových aplikacích. Za nimi následují kapitoly popisující návrh a průběh implementace grafové databáze a nástroje pro zpracování dat. Součástí návrhu grafové databáze je také popis vybraných distribucí grafových databází na trhu a jejich srovnání. Následně jsou v práci uvedeny problémy, které se během implementace vyskytly a jejich řešení. Celá práce a její výsledky je v závěru shrnuta.

1 Definice problému

Jak bylo zmíněno v úvodní části, behaviorální štít je program, který průběžně sleduje a vyhodnocuje běh jednotlivých procesů. Od roku 2017 je takový program také součástí antivirového programu od společnosti Avast Software s.r.o. V případě, že proces z pohledu behaviorálního štítu vykazuje podezřelé chování, automaticky učiní příslušná bezpečnostní opatření. Interní reprezentace všech procesů v systému a jejich vzájemné interakce jsou vygenerovány ve formátu XML a zasílány na servery sloužící ke zpracování těchto dat. Ty je však nyní zpracovávají pouze proudově a aktuálně neexistuje systém, který by je dlouhodobě ukládal k analytickým účelům. Hlavní problém tedy spočívá v tom, že v takových datech momentálně nelze efektivně vyhledávat a pracovat s nimi.

Proto je vhodné navrhnout a implementovat takové technické řešení, které by umožňovalo nad mnoha daty provádět rychlé analytické dotazy. Součástí takového řešení by měl nejdříve být nástroj, který vstupní data analyzuje a pročistí na základě uživatelsky stanovených metrik. Následně je importuje do vhodného databázového řešení.

2 Teoretický úvod do relačních a grafových databází

Ke správnému řešení problému je třeba nejprve znát základní principy databází. Tato kapitola se věnuje popisu, základní struktuře a indexaci dvou nejrozšířenějších modelů databázového systému. Na konci kapitoly jsou tyto dva modely navzájem porovnány.

2.1 Relační databáze

Relační databáze je jeden z nejrozšířenějších modelů k shromažďování a evidenci informací. Potřeby těchto činností sahají až do roku 1890, kdy vznikla objednávka na vůbec první „automat na bázi děrných štítků“ [1]. Spolu se vznikem prvního digitálního počítače v roce 1970 zavedl britsko-americký matematik a informatik Edgar Frank Codd pojem „relační model“ [2]. Jako reakci firma IBM vyvinula velice známý a dodnes používaný jazyk SQL [3]. S první komerčně využitelnou databází přichází na trh společnost Oracle v roce 1970 [4].

Pokud hovoříme o databázích dnes, rozlišujeme pojmy „DB“ (báze dat – *database*) a „DBMS“ (*database management system* – systém řízení báze dat). Zástupci komerčních DBMS jsou: MySQL, MSSQL, PostgreSQL, Oracle a další. Jejich podrobnější popis lze najít v [5].

2.1.1 Základní struktura

Základním stavebním kamenem relačních databází jsou vhodně navržené tabulky také nazývané jako **relace**. Ty popisují logickou a fyzickou strukturu dat. Jejich sloupce se nazývají **atributy**, které mají předem určený datový typ a jejich řádky jsou pak **záznamy**, které slouží k ukládání samotných dat. Obecně jedna tabulka představuje definici jedné **entity** a záznamy v tabulce představují jednotlivé **instance** této entity.

V relační databázi pak existují speciální druhy atributů (sloupců), které plní specifické funkce. Tyto atributy nazýváme **klíče**:

- **Primární klíč** – Jedná se o jedinečný (v jedné tabulce se vyskytuje maximálně jednou) atribut, který vždy identifikuje jednotlivou instanci. Většinou je typu Integer[1] nebo typu Guid[2] a má pro sebe vyhrazený jeden sloupec v tabulce. I přestože např. rodné číslo v tabulce registru osob by byl také vhodný

[1] velmi rozšířený datový typ uchovávající celé číslo

[2] datový typ sloužící jako jednoznačný identifikátor

kandidát na primární klíč, bývá běžně uložen jako samostatný atribut a funkci primárního klíče plní jiný sloupec. Ten se zavádí samostatně proto, aby jeho plnění mohlo probíhat zcela automatizovaně. Vzhledem k jeho jedinečnosti se poté dají mezi tabulkami vytvářet relace (viz další bod).

- **Cizí klíč** – Tento atribut se v tabulce definuje jako „odkaz“ na cizí tabulku představující její primární klíč. Díky těmto klíčům lze mezi tabulkami vytvářet několik různých vztahů. Vztah 1:N, kdy se na instanci tabulky A odkazují cizím klíčem několik různých instancí z tabulky B. Vztah 1:1, kdy se instance tabulky A odkazuje cizím klíčem na instanci tabulky B a vice versa. A nakonec vztah M:N, díky kterému lze k libovolnému počtu instancí z tabulky A odkázat libovolný počet instancí z tabulky B (takový případ se nejčastěji realizuje tzv. „tabulkou relací“, která nemá jiné atributy, než cizí klíče odkazující na tabulku A a B) [6][7].

2.1.2 Databázové transakce a systém ACID

Databázové transakce znamenají vykonání jednoduché operace nad daty uložené v DB. Může to být např. vložení nových dat do tabulky, selekce určitých dat, atd. Vzhledem k tomu, že k databázi může mít přístup více uživatelů zároveň, je žádoucí, aby jednotlivé transakce dodržovaly určitá pravidla.

Základy těchto pravidel popsal Jim Gray ve své práci [8] v roce 1981, které však rozšířili a zpopularizovali autoři A. Reuter a T. Härder v roce 1983 [9]. Tato pravidla byla pojmenována jako „ACID“ (*Atomicity, Consistency, Isolation, Durability*):

- **Atomicity (atomicita)** – pravidlo říká, že by databázová transakce měla být nedělitelná, čili se provede buď jako celek nebo se neprovede vůbec. Chrání tak databázi např. před výpadkem proudu uprostřed vkládání dat.
- **Consistency (konzistence)** – vykonávaná transakce může databázi přivést z validního stavu pouze do dalšího validního stavu. Např. vkládaná data by měla splňovat strukturu tabulky.
- **Isolation (izolovanost)** – výsledky neukončené transakce by měly být skryty před ostatními běžícími transakcemi. Jednotlivé transakce by tak měly být na sobě nezávislé.
- **Durability (trvalost)** – výsledek ukončené transakce by měl být správně zaznamenán a úspěšně uložen do databáze.

Z popisu všech čtyř pravidel lze usoudit, k jakému cíli vedou. Jakákoliv operace nad databází musí být buď úspěšná a uložená nebo se její její efekt vůbec nedotkne zachovalých dat. Nemůže se stát, aby se proces v nějaké fázi ukončil a stav databáze zůstal tak, jak je. Za tímto účelem vznikl také mechanismus operací transakcí.

První operace *„begin“* značí začátek transakce a tudíž její izolování od ostatních. Druhá operace *„commit“* značí ukončení transakce, uložení výsledku do databáze a také kontrola, jestli se databáze nachází ve validním stavu. Pokud se v něm nenachází nebo nastala chyba již při konání transakce, přichází na řadu operace *„rollback“*, která ukončuje transakci, odvolává její dosažené výsledky a uvádí databázi do původního stavu. Více informací lze najít v [10], [11] a [12].

2.1.3 Index relační databáze

Index(-ování) databáze je proces, který slouží ke zvýšení efektivity vykonávaných transakcí, převážně však při vyhledávání a procházení již uložených dat. Designér, správce nebo programátor vždy musí nejprve určit, který sloupec (neboli atribut) které tabulky (neboli entity) zvolí k indexování. Jako zpětnou vazbu pro zvýšení efektivity vyhledávání si databázový server může vyžádat větší nároky na paměť a pomalejší vkládání dat do DB.

Index funguje tak, že s jeho vytvořením si server uloží bokem informace, kde v paměti najde příslušnou hodnotu daného atributu. Z tohoto důvodu je vhodné vytvářet index nad tím sloupcem, nad kterým v budoucnu plánujeme vyhledávat (např. pokud se plánuje v databázi vozidel vyhledávat podle barvy auta, naindexuje se sloupec, který v sobě tuto informaci obsahuje). Jakmile potom dojde k transakci pro vyhledání určitých dat, databázový server nebude projíždět tabulku řádek po řádku, ale podívá se do indexu a rovnou bude sahat v paměti po konkrétní instanci.

Předchozí odstavec vypovídá o jedné ze dvou metod indexování – *non-clustered* metoda. Jednotlivé záznamy jsou v souboru uloženy se seřazeným indexovým klíčem (což odpovídá dané hodnotě v daném sloupci) a s ukazatelem na místo záznamu. Při této metodě se právě indexují sloupce, které nejsou primárním klíčem. Další – *clustered* metoda – rozdělí uložená data na menší podskupiny dat, které mají nějakou společnou vlastnost a zároveň jsou data v podskupině seřazená. V případě vyhledání určitých dat pak lze podle společné vlastnosti sáhnout po menší skupině a v ní vyhledávat rychleji. Takové rozdělení na podskupiny může být opakované.

Extrém takového rozdělení je např. indexování na základě B+ stromů. Tato struktura uložená vedle databáze v sobě obsahuje všechna data rozdělena to grafu typu strom. Každý uzel v sobě obsahuje rozmezí dvou hodnot, které porovnáváme. Jednotlivé výstupy pak ukazují, kam po hranách máme jít, pokud se hledaná hodnota nachází v daném rozmezí nebo je vyšší/nižší, než dané rozmezí. Takové rozdělení může být několikrát opakováno až ve finále dojdeme k nejnižší úrovni uzlů, kde jsou už jen samotné hodnoty. K těm jsou přiřazeny ukazatele na reálné záznamy.

Více informací o indexování databáze zde: na stránkách [13], [14], [15] a nebo také v patentu [16].

2.2 Grafová databáze

Předchozí kapitola pojednávala o relačních databázích. Postupem času se začal generovat stále větší objem dat a začala stoupat jejich propojenost (s tím souvisí i postupná ztráta předvídatelnosti jejich struktury ukládání). Vzhledem k rychlému vývoji cloudových služeb a rozsáhlých aplikací, který zapříčinil výrazně větší nároky na rychlost, se začaly vyvíjet tzv. „NoSQL“ databázové řešení[3]. Jejich základní charakteristiku lze vyčíst přímo z názvu – bez využití SQL neboli bez využití relačního modelu. Data jsou tedy v těchto DBMS ukládány jinak, než v tabulkách. To přináší několik výhod: není předem stanové pevné schéma, jsou vysoce rozšiřitelné a k dotazování není potřeba žádný příkaz JOIN.

Mezi reprezentanty NoSQL modelů patří *Key-Value Stores*, *Column Family Stores*, *Document Stores* a ***grafové databáze***. Každý model je určen pro jiný typ datového schématu. Vývojář tak může volit za účelem efektivnějšího ukládání a lepší struktury dat, které si přeje evidovat a spravovat. O prvních třech modelech více informací v [10], [17] a [18].

2.2.1 Základní struktura

Základní struktura grafové databáze používá graf jako reprezentaci dat. Ten je tvořen uzly a hranami, ve kterých jsou uložena data a vztahy (relace) mezi nimi. Tyto objekty mohou mít navíc své atributy, do kterých lze uložit doplňující informace (tzv. *„Property graph“* [19]). Uzel v takovém modelu představuje instanci entity a jeho vlastnosti pomáhají tuto entitu popsat. Hrany slouží k definování kontextu mezi entitami a jejich vlastnosti slouží k identifikaci váhy, důležitosti, typu a kvalitě vztahu. Používají se převážně modely s orientovanými hranami, ale existují i s neorientovanými. Druhý příklad je implementován duplikováním orientované hrany s opačným směrem [10].

Dalším typem grafové databáze je *„Hypergraf“*. Jeho jedinou charakteristikou je schopnost hranou spojit více entit dohromady. Ztrácí pak smysl hovořit o tom, odkud-kam hrana vede, ale které konkrétní uzly spojuje dohromady. Nakonec byl vyvinut graf typu *„RDF Triple“*, který pro ukládání používá RDF dokumenty [20]. Ten se skládá z trojice subjekt-predikát-objekt, která umožňuje vysokou škálovatelnost, avšak pomalejší procházení a vyhledávání v grafu. RDF graf byl vyvinut za účelem získávání informací na internetu.

[3]Tento pojem byl poprvé použit jako název open-source databáze v roce 1988 (Carlo Strozzi), ale svoji popularitu získal až v roce 2009 (Eric Evans) [17].

V roce 2000 byl poprvé prezentován také základní teorém **CAP**. Je to acronym pro *Consistency*, *Availability* a *Partition Tolerance*. Samotný teorém říká, že distribuovaná databáze (např. nějaká webová služba s daty) nemůže garantovat všechny tři vlastnosti zároveň. *Consistency* hovoří o tom, že by se prováděná služba měla provést kompletně nebo vůbec. Odpovídá tak pojmu *Atomicity* v modelu ACID. *Availability* znamená, že jakýkoliv požadavek na zdravý uzel musí vrátit validní odpověď. Čili pokud je služba aktivní, musí naslouchat, přijímat zprávy a zpracovávat je. *Partition Tolerance* pak vyjadřuje schopnost systému fungovat i při ztrátě zpráv či části sítě. Při výběru grafové databáze pak musíme jednu z těchto vlastností obětovat. Zahodit *Partition Tolerance* znamená mít všechna data uložena v jednom jediném grafu. Zahození *Availability* může znamenat větší zpoždění na požadavky do databáze a zahození *Consistency* může a nemusí vést ke kompletnímu defektu uložených dat. Více informací k teorému CAP jsou dostupné v [21].

2.2.2 Index grafové databáze

Grafová databáze disponuje vlastností *index-free adjacency* (volně přeloženo jako bezindexová sousednost), která vyjadřuje, že není třeba využívat globálních indexů k prohledávání v databázi, jelikož vztah mezi dvěma uzly je přímo vyjádřen v samotných uzlech. Každý uzel totiž obsahuje přímé odkazy na své sousedy. Zde přichází jedna z největších výhod oproti relačním databázím – spojování dat, které je zde reprezentováno přímým odkazem, je v relačních databázích velmi časově a výpočetně drahé (kdy se před samotným dotazem musí ještě potřebná data pospojovat dohromady pomocí cizích klíčů). Výhoda tedy spočívá v tom, že takový přechod mezi daty je v grafové databázi již předpřipravený a probíhá v konstantním čase (v podstatě skok z jednoho uzlu na druhý).

Ve většině případů se tato vlastnost implementuje tak, že si každý uzel uchovává ve vlastním seznamu informace o svých sousedech. Každý soused a hrana k němu je označen jednoznačným identifikátorem a všechny tyto seznamy jsou pak uloženy dohromady někde v datovém úložišti (kde a jaký identifikátor je použit, se liší mezi různými DBMS) [19].

Nicméně stále je možné některé atributy uzlů indexovat za účelem zvýšení efektivity hledání prvního uzlu, ze kterého se pak pomocí hran nachází další uzly. Např. se může vytvořit index na název uzlu – při komplexnějším dotazu, u kterého je třeba najít uzly s konkrétním jménem, ze kterých se pak najdou další uzly, se prvně pomocí indexu naleznou uzly podle jména a následně podle hran nalezou ostatní uzly.

2.2.3 Dotazovací dialekty

Vzhledem k tomu, že jednotlivé modely NoSQL databází se vyvíjely za jiným účelem, jejich dotazovací dialekty nejsou stejné. V daném NoSQL modelu navíc začalo vyvíjet více společností a tak existují různé dialekty i mezi různými DBMS. Grafové databáze nejsou výjimkou a tak zde můžeme nalézt řadu různých způsobů, jak napsat dotaz nad databází. V následujících odstavcích bude popsáno pár nejznámějších dialektů z hlediska jejich vývoje a využití, doplněné o jednoduchou ukázku.

Cypher

Jedná se o vysoce populární dialekt používaný primárně v nejznámější implementaci grafové databáze Neo4j. Byl vyvinut tak, aby splňoval potřeby dotazování nad daty uložené v grafové struktuře a zároveň měl podobnosti na již existující SQL. Jeho nejviditelnější vlastností je grafická podobnost na uzly (psané v kulatých závorkách) a hrany (psané v hranatých závorkách uprostřed šipky `->`). Dotazy tak mají připomínat znázornění průchodu grafem.

Pro práci se samotnými uzly a jejich vztahy slouží klíčová slova CREATE, UPDATE nebo DELETE. V případech, kdy je nutné nějaký již existující uzel v databázi nalézt, využijí se příkazy MATCH (nalezení uzlu s konkrétními atributy) nebo MERGE (tento příkaz kombinuje funkci příkazů MATCH a CREATE dohromady). K další filtraci slouží příkaz WHERE. Ukázku dotazovacího dialektu lze najít ve výpise 2.1. Všechny informace o Cypher jazyku lze najít v jeho online dokumentaci: [22].

Výpis 2.1: Ukázka Cypher dialektu

```
MATCH (p:Process {NodeName: "C:\\WINDOWS\\SYSTEM32\\
   SPOOLSV.EXE"})-[:SPAWN]->(x)
WHERE NOT "TRUSTED" IN x.characteristics
RETURN p,x;
```

GraphQL+−

GraphQL+− je dotazovací dialekt vyvíjený společností Dgraph. Jako takový je založený na jazyku GraphQL od společnosti Facebook. Od svého předka se liší v několika přidaných a odebraných funkcí, které byly pozměněny za účelem co nejlépe podporovat dotazování nad daty uloženými v grafové struktuře. Každý dotaz má své vlastní jméno, aby byla zjednodušena identifikace výstupu. V závorce následují podmínky, které musí hledané uzly splňovat a ve složených závorkách pak vlastnosti uzlu nebo hrany, které se mají vypsat. Dotaz pak vrátí úplně všechny jeho nálezy, které odpovídaly podmínkám a vyplní všechny požadované vlastnosti, které doplnit může (tzn. jsou tyto informace v daných uzlech uloženy). Ukázka dotazu v GraphQL+−: 2.2. Více lze vyčíst z oficiální dokumentace: [23].

Výpis 2.2: Ukázka GraphQL+− dialektu

```
{
  query(func: eq(NodeName, "C:\\WINDOWS\\SYSTEM32\\
     SPOOLSV.EXE")) @filter(type(Process)) @cascade {
    NodeName
    spawn @filter(NOT eq(Characteristics, "TRUSTED")){
      uid
      NodeName
      Characteristics
    }
  }
}
```

SPARQL

Tento jazyk byl vyvinut pro data uchovávána v RDF formátu a je uznáván jako klíčová technologie sémantického webu. Existuje zde velká podobnost na SQL jazyk zejména v klíčových slovech dotazu. Stejně jako RDF trojice, i SPARQL se skládá z trojice subjekt-predikát-objekt. Účelem dotazu je pak najít stejné nebo podle podmínek podobné trojice, jako ty, které jsou dotazovány. Existují klíčová slova SELECT – které vrací výsledek podobně jako v SQL, CONSTRUCT – které vrací výsledek už seskládaný do nových RDF trojic a ASK – které vrací logickou hodnotu true/false [24].

Výpis 2.3: Ukázka SPARQL dialektu

```
SELECT ?p,?x
WHERE {?p a~<Process>;
    <NodeName> "C:\\WINDOWS\\SYSTEM32\\SPOOLSV.EXE";
    <SPAWN> ?x;
    FILTER NOT CONTAINS(?x <Characteristics>, "TRUSTED")}
```

2.3 Srovnání relační a grafové databáze

Následující kapitola bude velmi stručná, jelikož všechny její poznatky vyplývají již z předchozích kapitol. Největší rozdíl mezi relační a grafovou databází je v základní struktuře, díky které pak v grafové databázi není třeba při dotazování spojovat zdroje dat podle atributu dohromady. Entitou v relační databázi je definovaná tabulka, kde jednotlivé sloupce představují atributy a řádky představují vytvořené instance entity. V grafové databázi předem definovaná entita není, vytváří se rovnou instance entity (uzel), která obsahuje libovolnou strukturu atributů. Relace v relační tabulce je vyjádřena cizím klíčem jako odkaz na primární klíč jiné tabulky, zatímco v grafové databázi relaci vyjadřuje hrana spojující dva uzly. Právě díky hranám jsou grafové databáze výrazně rychlejší v prohledávání a procházení dat.

Další výhody grafové databáze nad relační nemusí být na první pohled zřejmé. Hodně totiž záleží na případu využití. Obecně by se ale dalo říct, že grafové databáze lépe zastupují objektově-orientované modely. A to nejen při návrhu struktury, ale i v samotném dotazování. S uzly se tak pracuje jako s celými objekty, zatímco v relační databázi je pro výběr komplexnějších dat nutné nejdříve znát data z jiných tabulek.

Dalšími výhodami grafové databáze jsou rychlé reakce na dotazy, efektivní ukládání dat, přístup k nim a vysoká škálovatelnost (avšak s větším nárokem na kapacitu disku). Další výhodou je dynamická struktura ukládaných dat. Zde ale velmi záleží na tom, do jaké míry se data mohou od sebe lišit. Velkou nevýhodou grafových databází je velmi malá rozšířenost mezi vývojáři a programátory, čemuž nepomáhají ani nesjednocené dialekty pro dotazování.

Velmi zajímavé práce pojednávající o rozdílech, výhodách a nevýhodách grafové databáze lze najít v následujícím článku a dvou zdrojích přímo od výrobců grafové databáze: [25], [26], [27].

3 Teoretický úvod do behaviorálního štítu

Na začátku roku 2017 společnost Avast vydala novou verzi svého antiviru, jehož součástí byl i patentovaný a dlouhodobě vyvíjený „behaviorální štít“. Jedná se o program, který sleduje chování spuštěných procesů v počítači a případně řeší situace, kdy program dělá něco potenciálně škodlivého. *„Behaviorální štít můžeme přirovnat k ochrance na velkých veřejných akcích. Stejně jako ochranka hlídá, zda se návštěvníci akce nechovají podezřele nebo nebezpečně, i Behaviorální štít sleduje všechny programy, které běží v počítači a předtím úspěšně prošly vstupní bezpečnostní kontrolou.“* [28].

3.1 Detekce škodlivé aplikace

V dnešní době je většina detekčních mechanismů antivirových programů založena na skenování a porovnávání otisků programu s databází otisků již označených jako škodlivé. Tento mechanismus je využíván jak při skenování již uložených souborů na disku, tak i za běhu programů. Toho vývojáři malwaru využívají a „opevňují“ svoji aplikaci tím, že např. mění strukturu, logiku nebo mění jeho proměnlivou podobu během šíření.

Behaviorální štít však funguje jiným způsobem. Jak už bylo několikrát zmíněno v této práci, namísto podoby programu měří jeho chování. K tomu se využívají různé klasifikační algoritmy. Škodlivost programu se vyhodnocuje pomocí skupiny charakteristik, které jsou přiřazovány na základně chování a jejich vah. Je to z toho důvodu, že obvykle stejné typy malwaru mají stejné charakteristiky chování, přestože mohou být naprogramovány rozdílně. Pomocí dalšího algoritmu se poté tyto charakteristiky zahrnou do výpočtu celkového „skóre“ programu a na základě něj je označen za bezpečný nebo nebezpečný. K této klasifikaci se nejčastěji používá např. Bayesův klasifikátor [29]. K samotnému naučení vah je možné použít vhodné samo-učící se mechanismy (neuronové sítě, genetické algoritmy, atd.).

Je ale třeba mít mechanismus, který bude bránit vytváření falešně-pozitivních případů. Jedním z nich je vytvoření „whitelistu“. Můžeme buď celý program označit jako důvěryhodný, aby na něj nebylo aplikováno měření nebo můžeme u programu specifikovat charakteristiky, které víme, že má mít. Ty následně nebudou počítány do celkového výsledku měření. Dalším mechanismem je specifikace charakteristik, které musí konkrétní malware mít (buď všechny ze specifikovaných nebo minimálně jednu).

3.2 Příklady vybraných charakteristik

Celkový výčet charakteristik se pohybuje v řádech desítek až stovek. V této kapitole je vypsáno jen několik nejnázornějších:

- **SURVIVE_REBOOT** – spustitelný program se nezruší při restartu počítače. Toto však automaticky nemusí znamenat malware, jelikož takhle mohou být naprogramovány i systémové procesy.
- **HAS_BEEN_ORPHAN** – program jako proces nemá žádného předka. Jeho rodičovský proces je již ukončen. Toto je velmi časté chování malwaru.
- **ACTION_USED_NETWORK** – program přistoupil k síti a využil nějaké služby nebo poslouchal na nějakém portu. Také nemusí samo o sobě automaticky znamenat malware.
- **WINDOW_NOT_VISIBLE** – program pracuje na pozadí.
- **PROCESS_IS_HIDDEN** – proces sice pracuje, ale např. ve správci úloh je skrytý. To je umožněno tak, že proces svůj kód vtiskl do paměti správce úloh a upravil tak zobrazení sebe samotného.
- **EXEC_FROM_WINDIR** – program byl spuštěn z průzkumníka Windows.
- **IS_SHADOW** – program má stejný název jako jiný, již důvěryhodný proces, většinou systémový (např. SERVICES.EXE). Rozdíl je v tom, že jsou oba umístěné v jiné části disku.
- **WRITES_TO_REGISTRY_STARTUP** – program se pokusil o zapsání do registru v oblasti start-up.
- **TURNS_OFF_WINDOWS_FIREWALL** – program se pokusil vypnout standardní Windows Firewall.

Jak je z příkladů zřejmé, některé charakteristiky samy o sobě vykazují podezřelé chování, zatímco některé charakteristiky jsou společné s ostatními bezpečnými programy. Úkolem behaviorálního štítu je tyto charakteristiky správně poskládat do skupin, ze kterých je možné vyhodnotit, zda se o malware jedná nebo ne. Informace v této kapitole a spoustu dalších lze najít v samotném patentu behaviorálního štítu [30] a v patentech souvisejících [31], [32].

3.3 Vstupní data

Telemetrická data pocházející z behaviorálního štítu mají formát XML a obsahují informace o jednotlivých procesech a spustitelných souborech. Množství těchto dat je značně velké. K této studentské práci bylo poskytnuto celkově až 100 GB dat ve formě 17 961 XML souborů.

Každý XML soubor má předem stanovenou strukturu. Nejdříve jsou v hlavním elementu *IDPAgentData* specifikovány informace o původu dat (např. identifikátor celého souboru, čas vygenerování, identifikátor uživatele, verze uživatelova OS a verze generujícího zdroje). Následuje element *Nodes*, ve kterém jsou jednotlivé uzly. Tyto uzly mohou být vždy jedním ze tří typů: **proces**, **spustitelný soubor** nebo **virtuálně spustitelný soubor**. Uvnitř každého uzlu jsou ve vnořených XML elementech uvedeny informace blíže popisující daný uzel[1].

Tyto informace definují jednotlivé atributy uzlu. Jednotlivé typy uzlů obsahují několik společných a několik rozdílných atributů. Všechny atributy se dále rozdělují na povinné (např. *PID* a *CreationTime*) u **procesu** a nepovinné, které např. nemusí obsahovat konkrétní hodnotu a mohou tak být prázdné nebo nulové (`<SHA256/>` nebo `<FileSize>0</FileSize>`). Tyto atributy nebyly známé v době odesílání telemetrických dat na server, protože ještě nemuselo dojít ke spuštění těchto souborů a proto byly prozatím z pohledu behavirálního štítu nezajímavé. Ukázkový příklad hlavičky souboru a jednoho uzlu přímo ve formátu XML je k dispozici v příloze A.

3.3.1 Společné elementy

Všechny 3 typy uzlů mají následující atributy společné:

- **NodeID** – celočíselný jednoznačný identifikátor uzlu.
- **NodeType** – textové označení typu uzlu
  (tedy „PROCESS“, „EXECUTABLE“ nebo „VIRTUAL“).
- **NodeName** – textový název uzlu. V případě procesu a spustitelného souboru představuje cestu k souboru, který daný uzel vyvolal. V případě virtuálního uzlu představuje jednoznačný identifikátor.
- **OutEdgeRelationships** – obsahuje jednotlivé vztahy na další uzly. Každý takový vztah má v sobě vložený atribut, který specifikuje jeho název pomocí jednoznačného identifikátoru. Tento identifikátor lze následně přeložit na různé typy vztahů (např. *instance-of*, *spawn*, *delete*, atd.). Následně jeden vztah obsahuje libovolné množství *RelationshipConnection* elementů, jejichž hodnota reprezentuje *NodeID* uzlu, kam daný vztah směřuje.

[1] V této a následujících kapitolách budou vynechány ty elementy, které nejsou jakkoliv relevantní pro účel této práce. Výčty elementů tak nejsou kompletní.

- **InEdgeRelationships** – obdoba *OutEdgeRelationships*, jen vyjadřuje, odkud daný vztah směřuje do tohoto uzlu.
- **Characteristics** – seznam textových řetězců, které byly blíže popsány v kapitole 3.1 a příkladně uvedeny v kapitole 3.2. U těchto charakteristik si můžeme být jisti, že je uzel splňuje.
- **UnknownCharacteristics** – stejné jako předchozí, jen zde nemáme 100% jistotu, že je daný uzel splňuje.

Jednotlivé typy uzlů pak obsahují další – pro ně specifické – elementy.

3.3.2 Elementy charakterizující spustitelný soubor

Executable uzel navíc obsahuje:

- **SHA256** –jednoznačný identifikátor vypočtený z obsahu spustitelného souboru.
- **FileSize** – velikost souboru vyjádřená v jednotkách B.
- **RegistryContext** – obsahuje vícero *RegistryOperationInfo*, které nesou informace o změnách v registru Windows, které daný uzel provedl.
- **Connections** – seznam informací (url, doména, adresa, port a protokol) o posledních n síťových lokací, kam se uzel snažil připojit nebo se připojil. n je libovolně volitelné, běžně však nastavené na hodnotu 5.
- **Detection** – pokud byl daný uzel detekován jako malware, tak tento element obsahuje informace o této detekci. Např. název techniky, subkomponenty nebo algoritmu, který škodlivý program odhalil.

3.3.3 Elementy charakterizující proces

Process uzel navíc obsahuje:

- **PID** – tzv. „Process Identifier“. Toto číslo je používán k identifikaci procesů ve většině operačních systémů.
- **CommandLine** – textový řetězec obsahující celou cestu ke spouštěnému souboru včetně všech parametrů příkazové řádky.
- **CreationTime** – časové razítko ve formátu *hh:mm:ss.fff* vyjadřující čas vzniku procesu.
- **Connections** – viz Connections u **Executable** uzlu.
- **NamedObjects** – seznam jednotlivých pojmenovaných objektů[2], které proces vytvářel nebo otevíral.

[2]Pojmenované objekty jsou systémová primitiva identifikovatelná jménem, sloužící k meziprocesové komunikaci. Např. mutex, atom, aj.

3.3.4 Elementy charakterizující virtuální spustitelný soubor

Uzel typu **virtual** obsahuje převážně stejné elementy, jako uzel **executable** (*SHA256*, *FileSize*, *Detection*). Má však navíc svůj specifický element *VirtualNodeContent*, který odpovídá textovému řetězci z příkazové řádky. *NodeName* pak odpovídá jednoznačnému identifikátoru, který je vypočtený z části tohoto řetězce. Představuje tak spustitelný obsah, který byl předán např. v příkazové řádce a není uložen na disku.

4 Teoretický úvod do vícevláknové aplikace

Vícevláknová aplikace si zakládá na implementaci paralelismu za účelem zvýšení efektivity pomocí tzv. vláken. Vlákno je jakýsi základní prvek, který je vykonáván procesorem. Každý proces může obsahovat jedno (hlavní) nebo více vláken (proto „vícevláknová aplikace“). Implementace více vláken se používá pro současné spuštění více různých úkonů nebo jednoho úkonu nad různými daty, z důvodu urychlení běhu aplikace nebo zpracování dat.

Výhody

Z porovnání konceptů využití více procesů s jedním vláknem versus využití jednoho procesu s více vlákny plynou určité rozdíly. Výhody vícevláknové aplikace jsou: *velká rychlost odezvy na uživatelský vstup* (program může pokračovat přestože nějaká jeho část vyžaduje delší dobu zpracování), *sdílení prostředků* (takže není využito tolik paměti RAM a existuje snadnější, rychlejší komunikace mezi vlákny) a *hospodárnost* (správa vláken je z pohledu režie a časové náročnosti lepší, než správa více procesů).

Nevýhody

Při rozdělení procesu na vlákna se každému vláknu přiřadí vlastní zásobník v paměti. Všechna vlákna však spolu sdílí paměťovou haldu procesu, což vede k vytvoření konceptu sdílených prostředků. Existuje tak riziko stavu zvaného „souběh“. Jedná se o přivedení dat do nekonzistentního stavu vlivem práce dvou různých vláken nad stejnými daty (např. jedno vlákno čte data zatímco druhé vlákno do nich zapisuje).

Další nevýhodou, která souvisí se sdílenými prostředky, je riziko stavu „uváznutí“. Ten nastává při velmi špatném hospodaření se sdílenými prostředky a značí stav, kdy se více vláken trvale zablokují, protože vzájemně čekají na prostředek, který je přidělen jinému vláknu. Podmínky pro vznik uvíznutí jsou:*vzájemné vyloučení* (každý prostředek je používán maximálně jedním vláknem), *držení a čekání* (vlákno, které využívá alespoň jeden prostředek, může požádat o další), *nemožnost odebrání* (prostředky může uvolnit jen vlákno, které jej drží) a *zacyklené čekání* (existuje skupina vláken, které čekají na prostředky držené ostatními vlákny tak, že první vlákno čeká na druhé, druhé vlákno čeká na třetí, atd.). Tyto podmínky se nazývají *Coffmanovy podmínky*[1] a aby došlo k uváznutí, musí platit všechny v daném časovém okamžiku.

Poslední nevýhodou je, že chyba v jednom vlákně může teoreticky způsobit pád celého procesu.

[1]Tyto podmínky poprvé popsal Edward G. Coffman, Jr. v článku [33] v roce 1971.

Synchronizační primitiva

Problém se sdílenými prostředky se řeší tzv. „synchronizací“ vláken. Ta zaručuje, že více vláken nikdy nepřistoupí zároveň ke stejným datům. Tento stav lze zajistit pomocí určitých synchronizačních primitiv. Mezi ně patří např. zámek, semafor, fronta zpráv, monitor a další.

Zámek a semafor fungují na bázi povolení vstupu do kritické sekce (kód, který pracuje se sdílenými prostředky) pouze jednomu vláknu. Ostatní vlákna musí počkat, až se jim vstup do sekce povolí. Fronta zpráv představuje frontu instrukcí, které si vlákna přeji vykonat. Tyto instrukce se nad daty provádí postupně. Monitor představuje třídu, která obsahuje metody pro práci se sdílenými prostředky. V rámci těchto metod jsou implementována jiná synchronizační primitiva.

5 Praktická část – grafová databáze

V následující kapitole je popsán návrh a postup implementace technického řešení grafové databáze a nástroje pro zpracování dat. Po celou dobu práce jsem postupoval na základě znalostí získaných v rámci rešerše a byl jsem do jisté míry ovlivněn radami, názory a zkušenostmi konzultanta Ing. M. Drbala.

5.1 Návrh struktury databáze

Vzhledem k tomu, že už samotná vstupní data představují strukturu grafu (uzly obsahující informace o jejich vztazích s ostatními uzly), bylo by vhodné ji co nejvíce zachovat. Avšak pro účely větší přehlednosti jsem rozšířil strukturu na jednotlivé „sekundární“ pod-uzly. Do této kategorie patří všechny elementy z původního „primárního“ uzlu, které v sobě dále obsahují více elementů, než jen jednu hodnotu (např. *NamedObject* bude pod-uzel procesu). Jednotlivé elementy s hodnotou pak představují příslušné atributy uzlu v DB.

Data přímo z hlavičky vstupního souboru tvoří tzv. „RootNode“ – hlavní uzel – který tyto informace uchovává jako své atributy. Na něj pak orientovanou hranou ukazují všechny primární uzly, které se v daném vstupním souboru vyskytují. Na tyto uzly pak orientovanou hranou ukazují uzly sekundární. Tato struktura je znázorněna v obrázku 5.1.

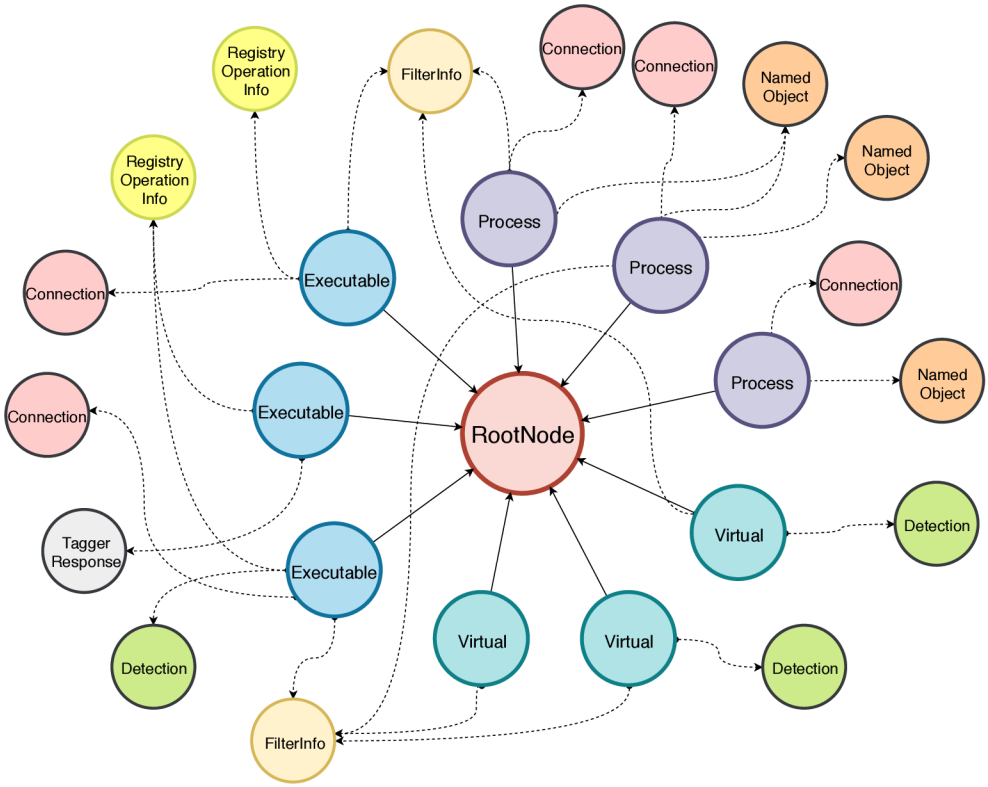
V samotné grafové databázi takových struktur mohou být stovky až tisíce. Každý vstupní XML soubor pak představuje nový *RootNode*. Vztahy v DB jsou vždy orientovány od primárních uzlů k sekundárním uzlům nebo k master uzlu. Je to z toho důvodu, aby každá hrana mohla univerzálně představovat notaci „HAS_SOMETHING“. Takové vyjádření by pak mohlo vypadat v dialektu „Cypher“ například následovně:

- `(Executable)-[HAS_ROOTNODE]->(RootNode)`
- `(Process)-[HAS_CONNECTION]->(Connection)`
- `(Virtual)-[HAS_DETECTION]->(Detection)`

V obrázku z důvodu zachování přehlednosti také chybí jednotlivé vztahy definované v XML elementech *InEdgeRelationships* a *OutEdgeRelationships*. Jejich vyjádření by analogicky vypadalo následovně:

- `(Executable1)-[INSTANCE_OF]->(Process1)`
- `(Process1)-[SPAWN]->(Process2)`

Za povšimnutí také stojí dva dosud nezmíněné uzly (*FilterInfo* a *TaggerResponse*), které se v obrázku 5.1 objevily. Význam těchto uzlů souvisí s filtrováním grafu na základě uživatelsky definovaných metrik. Tímto tématem se blíže zabývá kapitola 6.3.4.



Obr. 5.1: Základní grafová struktura vstupních dat

5.2 Popis vybraných grafových DBMS

Na trhu se grafových databázových systémů vyskytují desítky, požadavkem na tuto práci je však výběr takové, která je *open-source* nebo poskytuje svoji distribuci pod licencí, která je zdarma a lze ji používat pro komerční účely. Výběr probíhal na základě instalace distribucí a následné měření zvolených metrik. Proto jsem musel nástroj pro zpracování dat připravit také na export dat ve více formátech, protože každá distribuce může přijímat rozdílná vstupní data.

Následující vyjmenované a stručně popsané distribuce grafových databází jsem vyhodnotil jako vhodné kandidáty pro splnění zadání této práce. Zároveň také drží relativně vysokou pozici na žebříčku popularity[1].

[1]Měřeno na základě kombinace skóre z počtu uvedení jejich názvu na webových stránkách, obecného zájmu z Google Trends, množství technických dokumentů ohledně dané databáze, počtu nabídek práce, ve kterých je databáze uvedena, počtu zmínění na profesních a sociálních sítí, jako např. LinkedIn nebo Twitter. Více zde: [34]

- **Neo4j**
  - nejpopulárnější, nativní, velice efektivní a rozšiřitelná grafová databáze,
  - komunitní edice je licencována pod *GNU General Public License v3* [35],
  - umožňuje práci na většině operačních systémech a podporuje více způsobů připojení k databázi,
  - díky popularitě je na ni vyvíjeno mnoho uživatelských procedur sloužících k ulehčení práce s daty,
  - dotazovací dialekt Cypher vytvořený primárně pro Neo4j.
- **Dgraph**
  - vysoce rozšiřitelná a relativně nová nativní grafová databáze,
  - možnost přerozdělování prostředků mezi více serverů při plném zachování ACID transakcí,
  - dotazovací dialekt inspirovaný dialektem „GraphQL“ interně vyvíjený společností Facebook,
  - použití licence *Apache 2.0* [36].
- **AgensGraph**
  - oproti předchozím se jedná o tzv. „multi-model“, tedy kombinace grafové, relační a dalších typů databází,
  - nástavba nad populární distribucí relační databáze *PostgreSQL*,
  - dotazovací dialekt lze volně kombinovat mezi dialekty SQL a Cypher,
  - použití licence *Apache 2.0*.
- **ArangoDB**
  - také „multi-model“ - data jsou ve všech případech uložená ve formátu JSON a podle typu databáze se s nimi rozdílně pracuje,
  - zakládá si na využití jednoho dotazovacího dialektu napříč různými typy databází,
  - proprietární dotazovací dialekt AQL velmi podobný běžným programovacím jazykům (např. využití notace *for*),
  - použití licence *Apache 2.0*.
- **TigerGraph**
  - nativní grafová databáze s vysokou podporou paralelizace,
  - jednotlivé dotazy se nepíšou přímo, ale nejdříve se musí „nainstalovat“ (před-připravit, např. naindexovat důležité atributy použité v dotazu) a umožnit tím efektivnější zpracování dotazu,
  - dialekt GSQL velmi podobný jazyku Cypher s větší podporou využití proměnných a práce s nimi v rámci dotazu,
  - vlastní Developer Edition umožňující využít databázi pro jednoho uživatele, jeden graf a na jednom stroji pouze pro nekomerční účely.

- **PostgreSQL**
  - v rámci práce bude také implementována jedna SQL databáze za účelem získání povědomí o rozdílech mezi grafovými a relačními databázemi,
  - výsledky praktického porovnání jsou shrnuty v kapitole 5.3.2.

5.3 Srovnání grafových DBMS

Tato podkapitola je věnována samotnému srovnání grafových DBMS. Všechny vyjmenované databázové systémy jsem nainstaloval podle jejich individuálních dokumentací na virtuální kontejner, který mi byl poskytnut od společnosti Avast (více o kontejneru v kapitole 5.3). Následně jsem zvolil vhodné metriky, změřil výkon jednotlivých databázových systémů a srovnal je. V této práci je uveden konkrétní postup na instalaci pouze toho systému, který se v rámci měření projevil jako nejoptimálnější.

Prostředí pro implementaci DBMS

Jelikož pro účely zpracování velkého množství dat vznikají vysoké nároky na hardware, poskytla mi společnost Avast přístup k virtuálnímu kontejneru s linuxovou distribucí *Ubuntu v. 18.04.3*. Obsahuje 4 fyzické procesory (16 logických procesorů) a 128 GB RAM paměti. Právě na tento kontejner jsem nainstaloval všechny zmíněné distribuce databáze a provedl jejich měření.

Nejdříve jsem kontejner aktualizoval a předchystal pro práci. Základní disk, na kterém je distribuce nainstalována, poskytuje pouze 8 GB volného místa. Proto jsem poprosil svého konzultanta o připojení externího disku do adresáře `/mnt/storage`, který má 4 TB volného místa k dispozici. Kompletní výpis příkazů pro zmíněné operace je vypsán v B.1.

Import dat do DBMS

Před samotným měření jsem musel nejprve data do databází naimportovat. Tento proces probíhal pomocí nativních nástrojů jednotlivých DBMS, které ve většině případů přijímaly data ve formátu CSV (až na databázový systém Dgraph, zde bylo třeba data převést do formátu JSON).

Tohoto jsem dosáhl díky vytvořenému nástroji pro zpracování, který jsem dynamicky modifikoval tak, aby převedl data do správného formátu. Jednotlivé časy importů jsem sice naměřil, nicméně kvůli charakteru práce je zde neuvádím (reálný import do výsledné databáze probíhá na základě TCP připojení, ne pomocí převedení dat do formátu CSV a využití nativního nástroje vybrané databáze).

5.3.1 Zvolené metriky

Do všech databázových systémů jsem naimportoval celkově 100 GB telemetrických dat, která obsahovala 65 129 623 uzlů, 298 341 817 atributů a 139 512 600 vztahů. Stanovil jsem dva analytické dotazy, které mají za úkol databázi otestovat v procházení grafu a hledání cest na základě indexovaných i neindexovaných atributů:

- Dotaz č. 1: *Najdi všechny uzly typu Process, jejichž název je „C:\\WINDOWS\\SYSTEM32\\SPOOLSV.EXE“, který ukazuje hranou typu „spawn“ na uzel, který ve svých charakteristikách neobsahuje řetězec „TRUSTED“. Vrať celou cestu včetně všech uzlů.*
- Dotaz č. 2: *Najdi všechny uzly typu Executable, které ukazují hranou typu „instance_of“ na uzly typu Process. Tyto uzly pak musí libovolným počtem skoků po hranách „spawn“ dojít k uzlu, který hranou „delete“ ukazuje na prvotní Executable uzel. Vrať celou cestu včetně všech uzlů.*

Jako první metriku jsem zvolil rychlost zpracování dotazu. Nad databází bude v budoucnu voláno spoustu analytických dotazů a je žádoucí, aby výsledky byly k dispozici co nejdříve. Vzhledem k velkému množství dat, které se bude do databáze ukládat, jsem jako druhou metriku zvolil velikost místa, kolik daná databáze zabírá na disku. Nakonec třetí metrika je, kolik paměti databázový systém zabírá při zpracování dotazů pro odhad potřebné konfigurace v případě pořizování dedikovaného serveru.

Indexace databázových systémů

Pro efektivnější vyhledávání podle jména uzlu jsem vytvořil index na *NodeName*, jelikož se podle něj v prvním dotazu filtrují počáteční uzly. Vytvoření indexu pozitivně ovlivnila rychlost dotazu, ale negativně velikost zabíraného místa v paměti. V Neo4j se například index vytváří příkazem `CREATE INDEX ON :Process(NodeName)`. Obdobně podle dotazovacích dialektů v dalších systémech.

5.3.2 Rychlost zpracování dotazů

V tabulkách 5.1 a 5.2 jsou uvedené výsledky měření dvou stanovených dotazů. Pro výpočet průměru a směrodatné odchylky jsem vynechal vždy největší a nejmenší hodnoty (označené hvězdičkou). U některých databázových systémů chybí výsledky k druhému dotazu, protože jej nelze v dotazovacím dialektu efektivně vytvořit (databázový engine neumí zpracovávat tzv. „pattern matching“ [37]). Konkrétní dotazovací dialekty a jejich popis lze najít ve výpisech v sekci D.

V rámci implementace a sestavování dotazů jsem označil databázový systém TigerGraph za nevyhovující z důvodů pomalé implementace nových analytických dotazů, neschopnosti „pattern matching“ a možného využití pouze placené verze.

Tab. 5.1: Měření rychlosti zpracování Dotazu č. 1

Pořadí měření	Neo4j [ms]	AgensGraph [ms]	Dgraph [ms]	ArangoDB [ms]	PostgreSQL [ms]
1.	1974*	673	322	5239*	5187*
2.	1029	584	285	1222	3742
3.	472	942	295	1278	3734
4.	291	577	179*	1224	3878
5.	115	402*	339	1087	3872
6.	75	956*	220	1013*	3586
7.	185	857	332	1531	3606
8.	286	894	225	1637	3955
9.	190	659	364*	1299	3664
10.	186	691	279	1629	3411*
11.	178	770	285	1724	3673
12.	67*	787	354	1523	3581
Průměr	298,9	743,4	293,6	1415,4	3729,1
Směr. odchylka	264,61	120,65	42,98	207,18	126

Na začátku měření je převážně u databází Neo4j a AgensGraph změřený čas podstatně vyšší, než zbytek naměřených hodnot. Důvodem je nejpravděpodobněji skutečnost, že databáze byla čerstvě spuštěna po restartu systému, tudíž si během prvního spuštěného dotazu nahrávala do paměti informace o uzlech z dotazu. Tomuto procesu se říká „zahřátí“ databáze. Následné dotazy trvají kratší dobu, protože se může během dotazovaní pro některá data sahat do paměti namísto čtení z disku.

Z tabulek je možné vypozorovat, že nejrychlejší zpracování dotazu měly databáze Dgraph a Neo4j. V Dgraph databázi jsem však nebyl schopen sestavit druhý dotaz, protože není možné specifikovat libovolné opakování hran a využití tzv. „pattern matching“. Proto její výkon není možné změřit pro procházení grafem přes typy uzlu (což není naindexovaný atribut). Ostatní databáze měly horší výsledky v obou případech. Důvodem mohou být např. modely databází AgensGraph a ArangoDB, které nejsou čistě grafového charakteru. O relačním modelu databáze PostgreSQL hovoří následující podkapitola 5.3.2.

Tab. 5.2: Měření rychlosti zpracování Dotazu č. 2

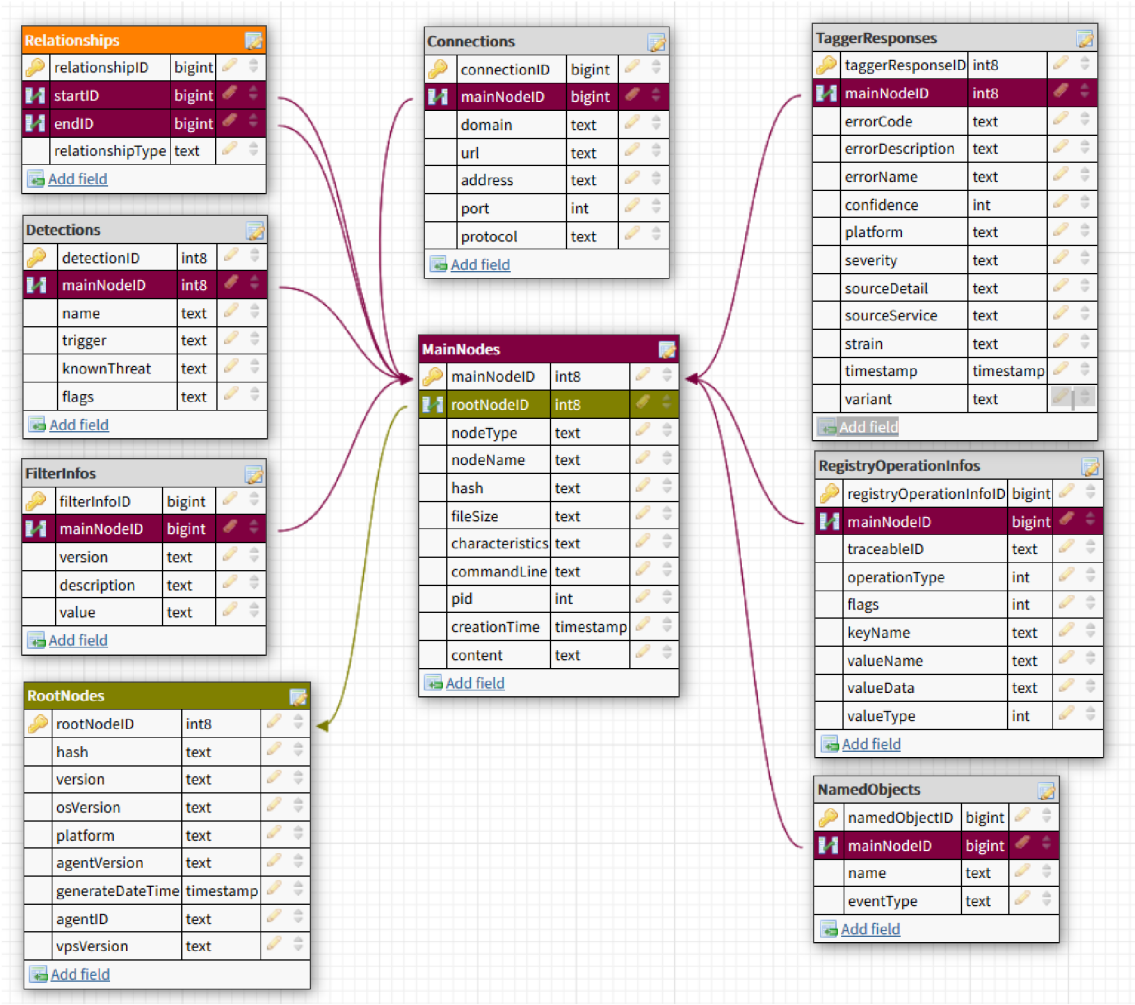
Pořadí měření	Neo4j [ms]	AgensGraph [ms]	PostgreSQL [ms]
1.	138884*	300567*	355405*
2.	7007	20363	231794
3.	6697	20516	230465
4.	6711	20948	233988
5.	6560	21021	232116
6.	6175	20039	249986
7.	6891	20261	22587
8.	6524	19950*	230326
9.	6088*	20344	224487
10.	6434	20177	218540*
11.	6670	20441	238382
12.	6730	21493	227697
Průměr	6939,9	20560,3	232482,8
Směr. odchylka	949,74	429,26	6979,61

Implementace a výsledky měření relační databáze

Pro ukázku, že pro zadání této práce se nejvíc hodí právě databáze grafové, jsem naimplementoval a změřil také jednu relační – PostgreSQL. Pro použití relační databáze bylo potřeba převést grafovou reprezentaci dat na reprezentaci relační (viz obr. 5.2).

Všechny hlavní typy uzlů jsou uloženy v jediné tabulce *MainNodes*. Uvnitř tabulky pak jsou k dispozici všechny možné atributy, které mohou a nemusí být vyplněny podle toho, o jaký typ uzlu se jedná (podle atributu *nodeType*). Hrany grafu jsem implementoval pomocí relací (cizího klíče) na ostatní tabulky. Jednotlivé vedlejší uzly tak mají atribut *mainNodeID*, který je zároveň primárním klíčem tabulky *MainNodes*. Samotné vztahy mezi hlavními uzly jsou v tabulce *Relationships*, která obsahuje identifikátory zdrojových a cílových uzlů a také atribut *relationshipType* označující typ vztahu.

Ve výpise D.5 jsou vypsány oba dotazy, ve kterých si lze všimnout ohromného množství příkazu JOIN (obzvláště v dotazu druhém), který propojuje data z jedné tabulky s daty z tabulky druhé. Právě tato operace je pro databázový server poměrně zdlouhavá, což se negativně projevuje na výsledcích měření. Grafové databáze toto spojování relací dělat nemusí, protože může jednoduše procházet po hranách, které zde představují relaci mezi dvěma uzly (jak již tomu bylo zmíněno v kapitole 2.3).



Obr. 5.2: Schéma vstupních dat v relační databázi

5.3.3 Velikost místa na disku

Tabulka 5.3 zobrazuje velikost využití místa na disku jednotlivých grafových databází (za předpokladu, že jsem do nich nahrál 100 GB vstupních dat). Nejlépe na tom je databáze Dgraph, následují databáze TigerGraph a ArangoDB, poté Neo4j, PostgreSQL a nakonec databáze AgensGraph.

Tab. 5.3: Požadovaná velikost místa na disku

	Neo4j	AgensGraph	Dgraph	TigerGraph	ArangoDB	PostgreSQL
Velikost místa na disku [GB]	29,2	62,1	13,8	19,4	19,1	36

5.3.4 Velikost zabíraného místa v paměti

Tabulka 5.4 zobrazuje hodnoty, kolik místa si jednotlivé procesy spuštěných databází zabraly v paměti při zpracování dotazů. Tyto hodnoty obsahují programový kód databáze, data procesu a také segmenty paměti související s procesem, které byly odloženy do odkládací paměti na disk. Můžeme pozorovat velice podobné výsledky mezi všemi databázovými systémy.

Tab. 5.4: Požadovaná velikost místa v paměti

	Neo4j	AgensGraph	Dgraph	ArangoDB	PostgreSQL
Velikost místa v paměti [GB]	37,638	35,4	39	37,8	36

5.4 Výběr vhodné grafové DBMS a její implementace

Na základě seznámení se se všemi měřenými DBMS, studiu jejich dokumentací, jejich instalace, implementace a samotném výsledku měření jsem se rozhodl použít databázi Neo4j. Tato databáze sice nevykázala nejlepší výsledky při zabírání místa na disku a v paměti – nicméně tyto prostředky se vždy dají dynamicky navýšit.

Naopak Neo4j prokázala velmi dobré výsledky – co se rychlosti zpracování dotazů týče – a navíc jako jedna z mála dokáže v databázi vyhledávat na základně „pattern matchingu“. Dalšími důvody pro výběr tohoto DBMS je také její popularita, s čímž souvisí velmi kvalitně zpracovaná dokumentace, vysoce rozšířená komunita a aktivní vývojářská podpora.

V příloze C.1 jsou uvedeny příkazy, které jsem použil pro instalaci a implementaci databáze. Nejdříve jsem nainstaloval JRE (*Java Runtime Enviroment*). Následně jsem do systému přidá nový repozitář, ze kterého následně lze stáhnout balíček *neo4j*. Nainstalovaná distribuce se chová jako služba, tudíž je možné využívat příkaz `service neo4j start|stop|restart|...` pro ovládání.

Po instalaci jsem spustil nativní Neo4j nástroj zvaný „memrec“ [38], který zobrazuje doporučenou konfiguraci týkající se paměti RAM na spuštěném stroji. Výstup tohoto nástroje jsem zkopíroval do výchozího konfiguračního souboru */etc/neo4j/neo4j.conf*. Výsledná konfigurace je uvedená v příloze C.2.

6 Praktická část - nástroj pro zpracování dat

Samotný nástroj pro zpracování dat jsem vytvořil pro účely přijetí vstupních telemetrických dat, aplikaci filtrů, klasifikace a jejich následný import do vybrané databáze

6.1 Vývojové prostředí a požadavky na nástroj

Pro účely efektivní implementace nástroje jsem vybral objektově orientovaný programovací jazyk C#. Objektový model nejlépe odpovídá grafovému modelu, kde jednotlivé uzly představují instance objektů a vztahy mezi nimi jsou vyjádřeny referencemi na tyto objekty. Dále jsem jazyk zvolil kvůli osobním preferencím a profesním zkušenostem.

Jedním z nároků nástroje je schopnost pracovat s velmi objemnými daty efektivně a co nejrychleji. Dalším nárokem je vysoká stabilita programu z důvodu delší doby jeho běhu. Chyby uvnitř práce vláken nesmí vyvolat pád celého programu. U závažnějších chyb (související s nástrojem jako takovým) je zásadní zaznamenat důvod jejich vzniku.

Pro samotný vývoj, který probíhal na platformě Windows 10, jsem využil programovací prostředí Visual Studio Enterprise 2019, ve kterém jsem vytvořil projekt pro vývoj konzolové aplikace v *.NET Framework* verze 4.6.1. Samotné nasazení nástroje a lokální zpracování dat však kvůli hardwarovým nárokům na paměť běží na linuxové distribuci Ubuntu verze 18.04 (viz kapitola 5.3). Jelikož *.NET Framework* není vyvíjen pro běh na linuxových zařízení, je vhodné pro nasazení aplikace použít multi-platformní nástroj, aby se nemusel pokaždé překládat do formátu kompatibilního s linuxovým strojem.

6.1.1 Mono

Takovým nástrojem je např. program „Mono“. Jedná se o *open-source* projekt přímo určený pro vývoj aplikací na vícero různých platforem. K jeho instalaci na operačním systému Ubuntu jsem nejprve zaregistroval správný certifikát, přidal repozitář do systému a aktualizoval jej. Následně jsem mohl nainstalovat jednotlivé balíčky tohoto programu. Např. základní balíček *mono-devel* sloužící k samotné kompilaci C# kódu, balíček *mono-complete* k instalaci všech základních knihoven (jako např. „System“, „System.Core“), balíček *mono-dbg*, který při pádu programu zobrazí detailnější informace o chybě a další. Příkazy pro instalaci a spuštění programu v Mono jsou uvedeny ve výpisu B.2.

6.1.2 .NET Core

Od roku 2019 společnost Microsfot vyvíjí nový framework zvaný *.NET Core*. Jedná se o open-source, multiplatformního přímého nástupce *.NET Framewok*. Vzhledem ke krátké délce vývoje (a možnými problémy s ní spojené) jsem se rozhodl zůstat u *.NET Framework*. Nicméně předělání projektu do *.NET Core* určitě zůstává jako možnost do budoucna.

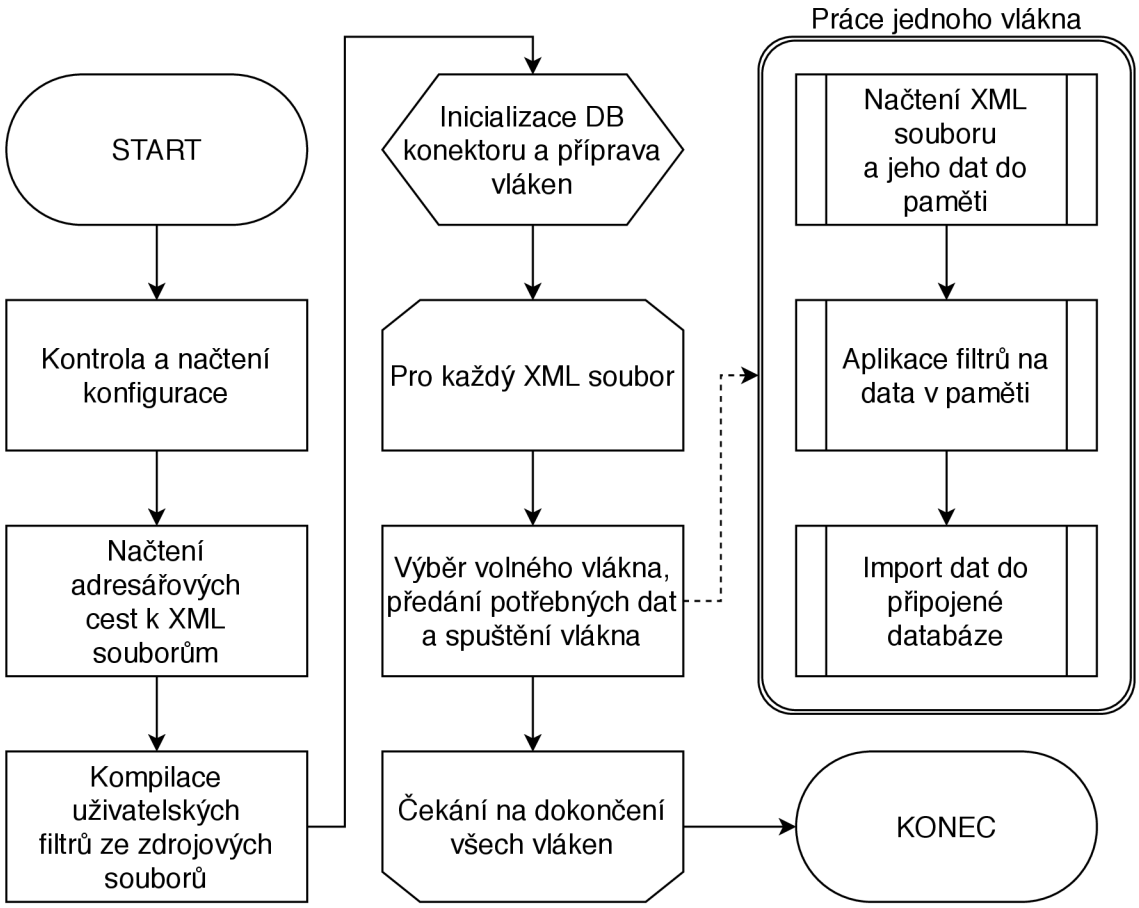
6.2 Specifikace činností nástroje

Vývoj nástroje jsem rozdělil na dvě fáze (resp. verze): konvertování formátu dat pro import do různých DBMS a kompletní zpracování dat včetně jejich importu do výsledné databáze. Protože první fáze byla přechodná a sloužila k měření databázových systémů, bude se tato práce věnovat pouze fázi druhé.

Abych dodržel požadavek na schopnost efektivně zpracovávat data ve velkém objemu a protože jsem dostal k testování stroj s více logickými procesory, využil jsem paralelního zpracování. Na obrázku 6.1 je znázorněný vývojový diagram, podle kterého se program řídí. V hlavní metodě *Main* ve třídě *Program.cs* se nachází základní logika pro inicializaci tohoto postupu. Ostatní třídy a jejich metody jsou využívány pro dosažení dílčích cílů.

Jak lze z diagramu vyčíst, hlavní část programu, která zajišťuje načtení, filtraci a import dat, probíhá paralelně v rámci jednotlivých vláken. Abych zajistil nezávislost nástroje na daném stroji, implementoval jsem konkrétní počet vláken, které se mají vytvořit, jako konfigurovatelnou hodnotu. Uživatel si tak může sám zvolit na základně počtu logických jader, kolik vláken nástroji přidělí (více informací se nachází v kapitole 6.5.4). Tento počet vláken se načítá ihned po startu nástroje zároveň s kontrolou celkové konfigurace.

Následuje získání informací o vstupních souborech (převážně jejich velikost a adresářová cesta k souboru). V projektu jsem vytvořil třídu *XmlFilesHelper*, která využívá nativní knihovnu a její metodu *DirectoryInfo.GetFiles()* dostupnou v .NET Framework. Proces je obalen v try-catch blocích pro zachycení všech typů výjimek. V případě chyby se výjimka zachytí, zahlásí se detail chyby a program se bezpečně ukončí. Více informací o načítání v kapitole 6.3.3.



Obr. 6.1: Vývojový diagram činností nástroje

6.3 Načítání a práce se vstupními daty

Pro popis načítání a filtrace dat je nejprve nutné definovat třídní strukturu. Z pomocí takto definovaných pojmů bude následně možné popsat samotné načítání dat a jejich filtrace.

6.3.1 Základní třídní struktura

Základní strukturu programových tříd jsem rozdělil takovým způsobem, aby odpovídala grafovému modelu uvedenému v kapitole 5.1. Jednotlivé typy entit jsou tak rozděleny na primární a sekundární (*RootNode* je zde klasifikován také jako sekundární). Dále program obsahuje pomocné třídy zajišťující kompilaci externích souborů, kontrolu konfigurace, načítání vstupních souborů, import do databáze, atd. Tato struktura je vypsána v příloze E.1. Její soubory a adresáře jsou blíže popsány v následujících podkapitolách.

Neklasifikované třídy

Projekt obsahuje pět tříd/souborů, které nepatří do žádné z uvedených kategorií. Prvním z nich je automaticky vygenerovaný *Program.cs*. Tato statická třída zajišťuje, že chod programu pevně následuje navržený postup uvedený v kapitole 6.2.

Další dva soubory jsou *packages.config* a *App.config*. První z nich je automaticky generovaný a obsahuje informace o využívaných knihovnách v celém programu, jejich verzi a cílovém frameworku. Druhý neboli aplikační konfigurační soubor slouží jako uživatelské nastavení všech potřebných parametrů. Uživatel předem volí, ze kterých složek se mají načítat vstupní data, počet použitých vláken, přístupové údaje k databázi a další. Program je díky tomu možné pouštět zcela bez vstupních parametrů.

Poslední a nejdůležitější třída je *NodeGraph*. Jedná se o základní stavební jednotku celé třídní struktury a v paměti reprezentuje načtený XML vstupní soubor s telemetrickými daty z behaviorálního štítu. Obsahuje referenci na *RootNode*, kde jsou uvedeny informace z hlavičky souboru a kolekci typu *List<INode>*, která obsahuje jednotlivé uzly načtené ze vstupního souboru.

Adresář ThreadPoolLogic

Předchozí podkapitola uvedla pět neklasifikovaných tříd, nicméně vyjmenovala pouze čtyři. Pátou třídou je *ThreadPoolLogic*, ve které je specifikována samotná logika práce jednoho vlákna.

Uvnitř se nachází struktura *ThreadPoolData*, která obsahuje proměnné potřebné pro chod jednoho vlákna. Jsou to konkrétně: číselný identifikátor vlákna (použitý při logování), cesta ke vstupnímu souboru (pro načtení), logická hodnota určující, jestli se mají na vstupní data aplikovat filtry a nakonec samotný reset-event objekt, díky kterému je možné hlavnímu vláknu signalizovat, že je vlákno k dispozici k další práci.

Třída dále obsahuje metodu *ThreadPoolProcedure*, která implementuje algoritmus podle vývojového diagramu 6.1. Pro jednotlivé operace se využívají metody jiných tříd a každá tato operace je obalena v try-catch bloku. V případě zachycení výjimky se zpráva o chybě zaznamená do předem nachystaného objektu *System.Text.StringBuilder*, jehož obsah se vypíše do konzole až těsně před zresetováním vlákna. Logování tak neprobíhá okamžitě, protože by jinak několik vláken vypisovalo do konzole zároveň a výsledný text by nedával smysl (samotný výpis do konzole je *thread-safe*, je však třeba jej provést pouze jednou za běhu vlákna).

Pro označení vlákna za ukončené jsem použil příkaz:

```
threadPoolData.manualResetEvent.Set();
```

Adresář MainNodes

Do tohoto adresáře patří rozhraní *INode* a základní typy uzlů, které z něj dědí. Toto rozhraní obsahuje všechny společné atributy typů uzlů jako své vlastnosti:

- `enum NodeType {PROCESS, EXECUTABLE, VIRTUAL}` – Při inicializaci primárního uzlu se v konstruktoru příslušné třídy naplní tato zděděná vlastnost na hodnotu podle toho, o jaký primární uzel se jedná. Tato vlastnost slouží k jednoduššímu rozpoznání typu uzlu při procházení kolekce všech uzlů v *NodeGraph*.
- `Dictionary<string, List<Relationship>> OutgoingRelationships` – kolekce obsahující reference na všechny odchozí vztahy na další uzly. Kolekci jsem realizoval pomocí slovníku, kdy klíč je vždy cílový/zdrojový uzel a hodnota představuje seznam vztahů (v některých případech totiž může uzel ukazovat na druhý dvěma různými vztahy).
- `Dictionary<string, List<Relationship>> IngoingRelationships` – kolekce obsahující reference na všechny příchozí vztahy od ostatních uzlů.
- `HashSet<string> Characteristics` – kolekce obsahující veškeré známé charakteristiky uzlu.
- `HashSet<FilterInfo> AppliedFilters` – kolekce obsahující informace o filtrech, které na uzel byly postupně aplikovány. Více o filtrech v kapitole 6.3.4.
- `string CustomID` – Jednoznačný identifikátor nahrazující *NodeID* uzlu. Nahrazení je nutné, protože původní identifikátor je jedinečný pouze v rámci jednoho XML souboru. Při pouhém přiřazení ID uzlům hrozí riziko vzniku kolize, které by bránilo importu uzlů v kolizi. Způsob tvorby tohoto atributu je více popsán v kapitole 6.5.3.
- `string Name` – Název uzlu.

Jednotlivé primární uzly obsahují tyto zděděné atributy a zároveň další pro ně specifické atributy. Pro *Executable* to jsou například (ostatní primární uzly analogicky podle toho, jaké mají specifické atributy):

- `List<RegistryOperationInfo> RegistryOperations`
- `List<Connection> Connections`
- `string Hash`
- `long FileSize`
- `DetectionStruct Detection`
- `TaggerResponse TaggerResponse`

Vlastnosti skalárního typu jako je *string* a *long* představují atributy uzlu, zatímco ostatní proměnné představují referenci nebo kolekci referencí na další objekty obsahující doplňující informace o uzlu.

Adresář SecondaryNodes

Tento adresář obsahuje skupinu tříd, které slouží jako úložiště informací pro uzly primární, které se na ně referencemi odkazují. Každá z nich pak obsahuje pouze definice atributů (jejichž struktura pochází ze vstupních souborů). Jejich naplňování probíhá buď ve třídách primárních uzlů nebo ve vlastní metodě načítání pomocí *XmlReader*, o kterém více hovoří kapitola 6.3.3.

Adresář ExtensionHelpers

Zde se nacházejí třídy, které slouží k definici, správě, načítání a aplikování uživatelských filtrů. O uživatelsky nastavených metrikách pro filtrování grafů je blíže zaměřená kapitola 6.3.4

Adresář Helpers

Pomocné statické třídy obsahující doplňující metody využívané napříč celým projektem. Patří zde např. *FileHelper*, který poskytuje možnosti práce s jednotlivými soubory. Nebo také *NodeGraphHelper*, který pomáhá při vytváření nových podgrafů. Jejich logika nevyžaduje pro charakter této práce bližší specifikaci.

Adresář Neo4j

Tento adresář obsahuje pouze jednu třídu *Neo4jDriver*. Pomocí ní se poté provádí import do připojené databáze. Bližší informace k importování lze najít v kapitole 6.4.

6.3.2 Inicializace vláken a paralelizace nástroje

K paralelnímu běhu nástroje jsem použil kombinaci objektů *ManualResetEvent*, *WaitHandle* a *ThreadPool* (všechny objekty jsou součástí nativní knihovny *System.Threading*). Nástroj funguje na principu přidělování práce v hlavním vlákně dalším – „worker“ – vláknům. Každý takový „worker thread“ obsahuje svůj vlastní *ManualResetEvent*, pomocí kterého dokáže signalizovat hlavnímu vláknu, že je jeho práce ukončena.

Na začátku běhu programu se v hlavním vlákně inicializuje nové pole (o velikosti konfigurovatelného počtu vláken) reset-eventů a pro každý prvek se nainicializuje nová instance objektu *ManualResetEvent*. Následuje iterace pro každý načtený vstupní soubor, uvnitř které se nejdříve získá index vlákna, které má signalizováno, že svoji práci již ukončilo a je k dispozici k dalšímu zpracování souboru. Zjištění volného vlákna lze provést pomocí příkazu `WaitHandle.WaitAny(events)`. Díky získanému indexu je následně možné přiřadit vláknu práci na dalším vstupním souboru.

Poté se vytvoří struktura *ThreadPoolLogic.ThreadPoolData* (objekt a jeho součástí jsou blíže popsány v kapitole 6.3.1), které se předají všechna data, která mají vstupovat do práce jednoho vlákna (např. adresářová cesta k souboru). Součástí těchto dat je také přidružený reset-event, aby se uvnitř vlákna mohl v případě ukončení práce označit za dokončený, což pro *WaitHandle* znamená signál, že jej může znovu vybrat k nové inicializaci.

Nakonec se vlákno spustí pomocí příkazu:

```
ThreadPool.QueueUserWorkItem(
    ThreadPoolLogic.ThreadPoolProcedure, threadPoolData);
```

6.3.3 Načítání dat ze vstupních XML souborů

NodeGraph se pro daný vstupní soubor plní pomocí objektu *XmlReader*. Ten obsahuje metodu *LoadXml*, která jako vstupní parametr přijímá cestu k danému souboru. Na začátku této metody se inicializuje nový *XmlReader* pomocí `XmlReader.Create(xmlFilePath)`. Následně se pomocí metod *MoveToContent()*, *Read()* a *ReadToFollowing()* načtou data pro *RootNode* a doplní se do něj.

Každý uzel v sobě obsahuje statickou metodu *LoadXXX* (např. pro virtuální uzel je to *LoadVirtualNode*). Tyto metody vždy jako parametr dostanou nainicializovaný *XmlReader.ReadSubtree()*, pomocí kterého si načtou potřebné hodnoty, vytvoří instanci objektu sebe samého a tu vrátí. Pokaždé, když objekt potřebuje načíst podřadné uzly, zavolá analogicky tuto metodu toho uzlu a předá mu instanci XML čtečky. Tento mechanismus tak vytváří hierarchický koncept načítání.

V mnoha případech se využívá cyklus *while (xmlReader.Read())*, uvnitř kterého se název elementu posílá do několika násobného bloku *if-else*. Jednotlivé podmínky se ptají na konkrétní atributy uzlu a pokud je najdou, načtou se pomocí *XmlReader.ReadElementContentAsString()*. Příkladné načítání objektu *Connection* je naznačeno v příloze E.1.

6.3.4 Filtrace a klasifikace podgrafů

Vyhledávání a klasifikaci podgrafů jsem implementoval jako načtení zdrojových kódu jednotlivých uživatelsky definovaných metrik do paměti za běhu programu. Díky tomu má uživatel možnost jednoduše podle třídní předlohy *ExtensionBase* psát vlastní „filtry“, které si přeje aplikovat na vstupní data.

Tato abstraktní třída obsahuje dva textové řetězce: *Version*, který uživatel musí definovat sám a *Name*, který se generuje automaticky podle názvu třídy. Dále obsahuje metodu *Apply*, do které uživatel píše samotnou logiku filtru, co se s daty má provést. Další již definovanou metodou je *CreateFilterInfo*, která se může v případě potřeby volat uvnitř metody *Apply*. Díky ní je možné vytvořit novou instanci objektu *FilterInfo* (viz obr. 5.1), který obsahuje stejné atributy jako filtr, ale navíc má uživatelem specifikovanou *Value*, která jednoznačně identifikuje daný podgraf a také *FileHash*, který obsahuje identifikátor XML souboru, ze kterého data pochází. *FilterInfo* se pak přiřazuje různým hlavním uzlům v grafu za účelem ukazatele, kterým konkrétním filtrem uzel prošel.

Díky tomuto mechanismu je možné hlavní uzly seskupovat a klasifikovat. Pro každou skupinu uzlů, které si přejeme klasifikovat, se může vytvořit jeden uzel *FilterInfo*, na který se uzly napojí. Uvnitř hodnoty *Value* je pak uložena informace o dané klasifikaci.

.NET Assembly

Jednou z novinek, které .NET Framework přinesl, je tzv. „Assembly“. Toto nové pojetí výsledného modulu aplikace se skládá z několika různých souborů (*exe*, *dll*, *cs*, atd.) a představuje základní stavební složku každého programu. Obsahuje metadata skládající se z odkazů na další Assemblies, seznamu souborů, identifikace Assembly a z definic typů (jako jsou např. datové typy, třídy). Díky tomu pomáhají vytvářet určitou samostatnost aplikací a jejich nezávislost na registrech.

Pro implementaci filtrů jsem se rozhodl vytvořit svou vlastní, novou Assembly. Důvodem je primárně nezávislost na zdrojovém kódu nástroje. Uživatelé, kteří si přejí změnit filtr, jednoduše přepíšou *.cs* soubor, ze kterého se tato Assembly načítá.

Kompilace a aplikace filtrů

K načtení ze zdrojových kódu od uživatele do paměti jsem vytvořil třídu *ExtensionCompiler*. Obsahuje jedinou metodu, ve které se program nejdříve podívá, jestli existuje adresář s filtry (tento adresář se také načítá z konfigurace) existuje a jestli nějaké zdrojové kódy obsahuje. Jestli ano, pomocí objektu *CSharpCodeProvider* z knihovny *Microsoft.CSharp* se pomocí reflexe načtou všechny uživatelské filtry do paměti v podobě již zmíněného objektu *Assembly*.

Objektu je nejdříve nutné v *CompilerParameters* specifikovat tzv. „ReferencedAssemblies“, což jsou knihovny potřebné ke správné kompilaci filtrů. Nebo se také ještě mohou specifikovat různé parametry kompilace. Např. generování pouze do paměti nebo do souboru s příponou „.dll“, vytvoření „debug“ informace, atd. Nakonec se zavolá funkce pro kompilaci filtrů:

```
CSharpCodeProvider.CompileAssemblyFromFile(
    parameters, filesToCompile);
```

Zkompilované filtry jsou dále podle diagramu 6.1 aplikovány v druhém kroku v rámci práce jednoho vlákna. V této fázi jsou vstupní data uložena v objektu *NodeGraph* (více o tomto objektu v kapitole 6.3.1), který se pošle do statické třídy, kterou jsem nazval „ExtensionPipeline“. Jak již z názvu vypovídá, má za úkol aplikovat filtry v řadě jeden po druhém. Uvnitř se nachází pouze jedna statická metoda *ApplyExtensions*, která přijímá kolekci filtrů a kolekci objektů *NodeGraph*.

Uvnitř metody se všechny filtry proiterují, přičemž v každé iteraci se zavolá metoda *Apply* daného filtru nad vstupní kolekcí *List<NodeGraph>*. Výstupem je znovu kolekce již klasifikovaných podgrafů *List<NodeGraph>*, která slouží jako vstup dalšímu filtru. Tímto způsobem se aplikují všechny načtené filtry a vrátí se kolekce podgrafů, které jsou připraveny na import do databáze.

Klasifikace a tvorba podgrafů

První dva uživatelské filtry byly společností Avast Software s.r.o. specifikovány již na začátku akademického roku. Oba dva nebyly výpočetně ani implementačně nijak náročné a aplikují se na každý uzel v grafu (tzn., že nevytváří žádné podgrafy). *VersionFilter* má za úkol zahodit a odfiltrovat všechny vstupní soubory, jejichž verze je nižší, než `18.x.x`. Druhý filtr *TaggerFilter* má za úkol ke všem *Executable* (viz 6.3.1) uzlům přidat tzv. *TaggerResponse*. To provede tak, že zjistí hodnotu identifikátoru uzlu a tu pošle jako parametr na určitou URL adresu v doméně `tagger.int.avast.com`. Webový server následně vrátí interní informace o uzlu v JSON formátu.

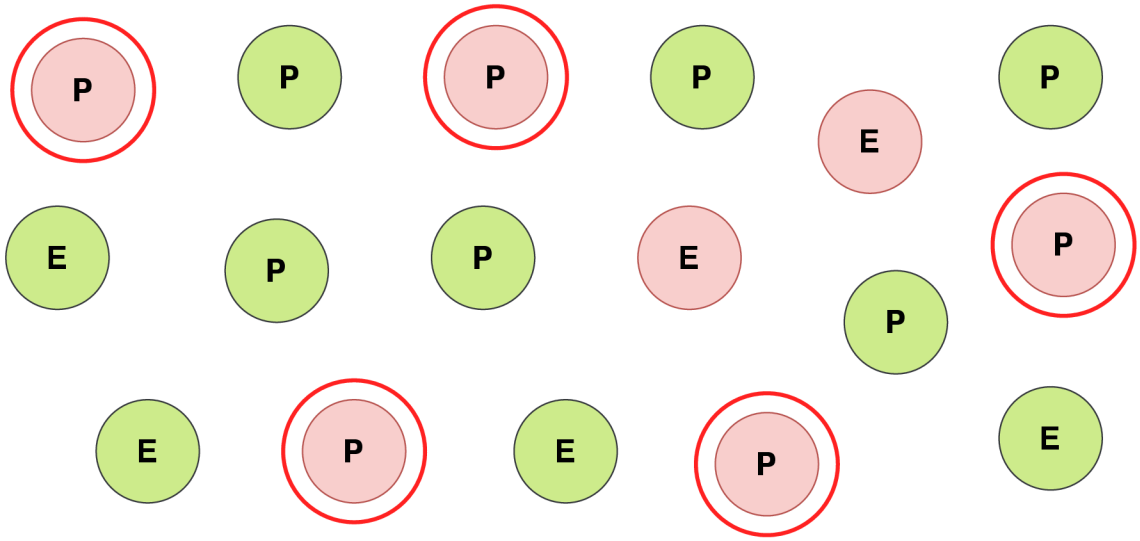
UntrustedSubgraphsFilter

Třetí, více sofistikovaný a důležitější filtr se nazývá *UntrustedSubgraphsFilter*. Jeho úkolem je najít všechny procesy, které neobsahují charakteristiku „TRUSTED“ a z nich pomocí vztahů na ostatní uzly vytvořit nový podgraf. Výsledkem je tedy množina podgrafů, které znázorňují nedůvěryhodné chování programu.

Z takových procesu je tedy možné se dopátrat např. ke spustitelnému souboru, který jej vyvolal, nebo je také možné v rámci podgrafu vidět, jak daleko vliv procesu dosáhl. Můžeme např. pozorovat řetěz vyvolávání nových procesů, na jehož konci

je původní spustitelný soubor smazán. Díky těmto informacím můžeme analyzovat, jak se různé škodlivé programy chovaly, jak fungují a podle toho ještě více „opevnit“ behaviorální štít (např. generace behaviorálních signatur pomocí strojového učení).

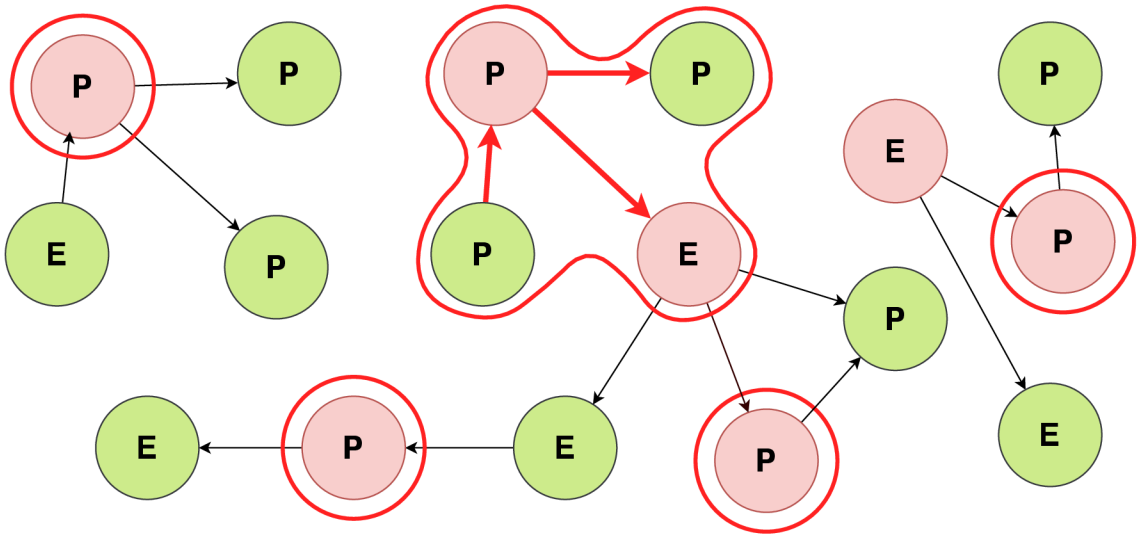
Nejprve se ve filtru získá prvotní množina všech procesů, které neobsahují charakteristiku „TRUSTED“. Víme totiž, že součástí každého výsledného podgrafu bude vždy minimálně jeden takový proces. Tyto uzly označíme jako „startovací body“ pro naše hledání. Viz obr. 6.2.



Obr. 6.2: Selekce nedůvěryhodných procesů

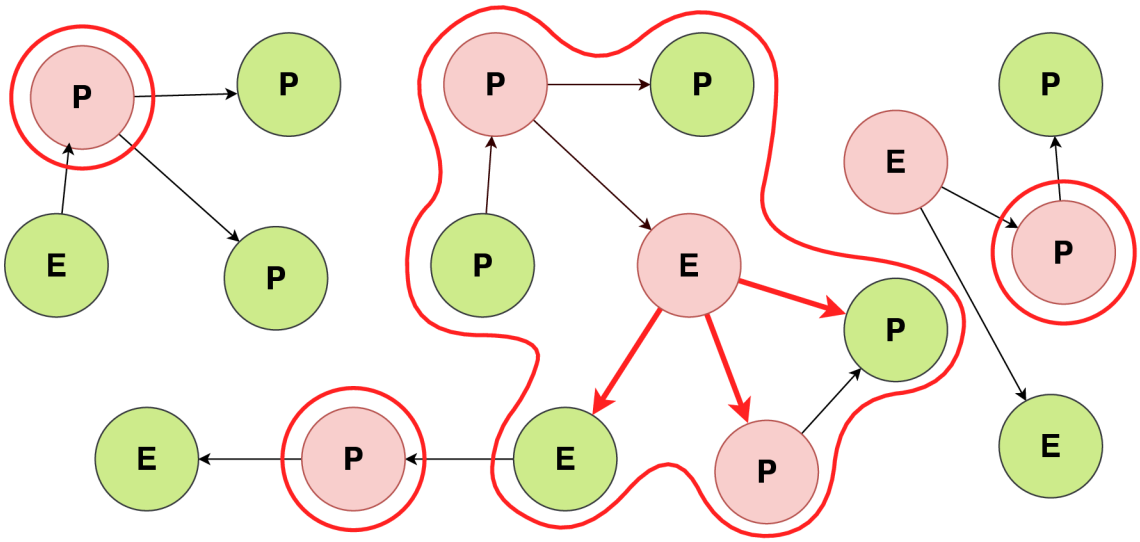
Pro každý nový startovací bod se nainicializují všechny potřebné proměnné, včetně instance třídy *NodeGraphHelper* – ta pomáhá vytvořit nový, prázdný *NodeGraph*, přidávat do něj uzly a vztahy na ostatní již přidané uzly. Také se naincializuje tzv. „subgraphList“, který představuje frontu pro prohledávání grafem. Použil jsem kolekce *List* namísto *Queue*, protože pro účely napojování vztahů v novém podgrafu potřebuji zachovat původní procházené uzly a jejich vztahy.

Následně se vezme startovací bod a provede se expanze jeho odchozích i příchozích vztahů. Cíl takového vztahu se vždy dynamicky přidá do podgrafu pomocí *NodeGraphHelper* a do *subgraphList*. Na obrázku 6.3 je znázornění expanze prvního startovacího bodu.



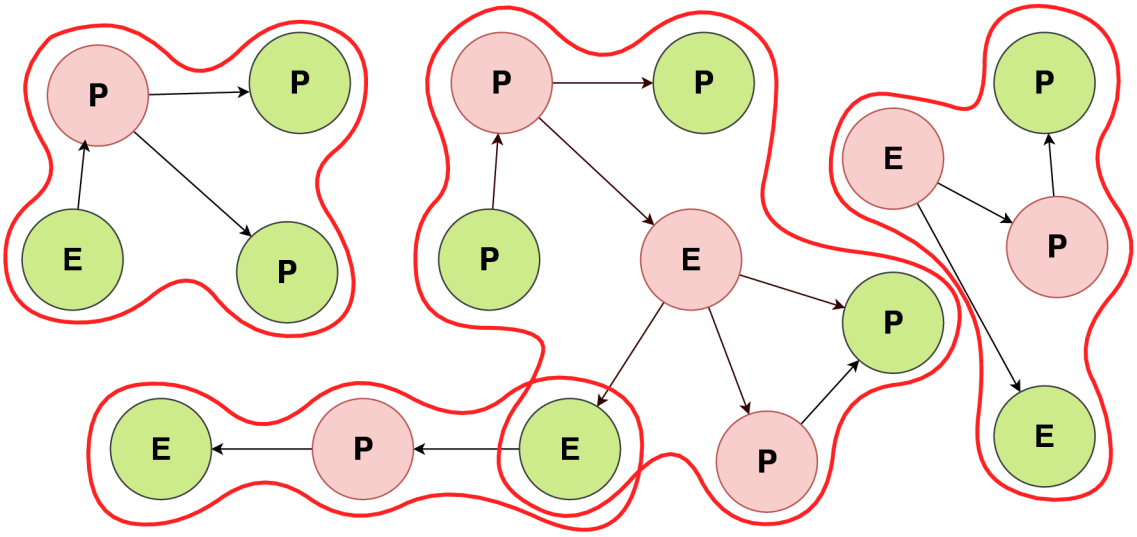
Obr. 6.3: Expanze nedůvěryhodného procesu

Poté se pomocí cyklu a zvyšujícího se indexu vezme vždy další uzel uvnitř podgrafu (čili další prvek v seznamu). Pokud je daný uzel „untrusted“, tak se rozexpanduje podobně jako v předchozím kroku. Pokud uzel charakteristiku „TRUSTED“ obsahuje, nechá se v podgrafu, ale k expanzi nedojde. Nutné podotknout, že pokud se touto expanzí dojde k dalšímu nedůvěryhodnému procesu, tak se tento proces vymaže ze startovacích bodů. Pokud bychom totiž v dalších iteracích startovacích bodů narazili právě na tento uzel, vytvořil by se nám ten stejný podgraf znovu. Tento proces je znázorněn v obrázku 6.4.



Obr. 6.4: Opakovaná expanze podgrafu po nedůvěryhodných uzlích

Zároveň v celém algoritmu funguje mechanismus kontroly, zda byl daný uzel již navštívený. Proto při pokusu o expanzi dalšího nedůvěryhodného uzlu, který však v podgrafu již je, se operace přeruší. Na obrázku 6.5 je možné vidět již finální podoba grafu rozděleného na podgrafy. Je možné se všimnout, že některé „trusted“ uzly jsou součástí více podgrafů. Tato skutečnost je možná a dokonce i pravděpodobná, jelikož expanze podgrafu přes důvěryhodné uzly neprobíhá.



Obr. 6.5: Finální podoba rozděleného grafu

Při psaní algoritmu jsem se silně inspiroval BFS algoritmem pro vyhledávání v grafu. Nicméně vzhledem k povaze zadání filtru jsem byl donucen provádět určité změny. Např. implementace konkrétně tří prahů, při kterých se tvorba podgrafu musí zastavit, aby daný podgraf nebyl až přebytečně moc velký. Je to práh expanze, kdy při překročení konfigurovatelného počtu expanzí následné expanze již neprobíhají. Dále je to práh uzlů, kdy je možné specifikovat maximální počet uzlů v jednom podgrafu a nakonec je to práh názvů uzlů. Zde se musí zachovat podmínka, že konkrétní název uzlu se v podgrafu může vyskytnout maximální tolikrát, kolikrát jej práh specifikuje.

Všechny prahy expanze představují předčasné ukončení tvorby podgrafu – proto jsem byl donucen implementovat další mechanismus: expanze podgrafu bez přidávání uzlů. Pokud totiž expanze skončí před prohledáním celého grafu, může se stát, že některý nedůvěryhodný proces, který by jinak byl součástí tohoto podgrafu, zůstane stále mezi startovacími body. A to by vedlo znovu k dvojitému vytvoření tohoto podgrafu. Proto se v této speciální expanzi pouze odstraňují tyto uzly z kolekce startovacích bodů, dále se však nic nepřidává.

6.4 Import dat do databáze

Dalším a posledním krokem práce jednoho vlákna je import do vybrané databáze. K tomuto kroku jsem nejprve využíval veřejnou komunitní knihovnu *Neo4jClient* (projekt je k dispozici na [39]) od uživatele Readify (publikovanou pod licencí *Microsoft Public License*). Nicméně v průběhu akademického roku vyšla nová verze databáze Neo4j (update z v3.5 na v4.0). Následkem byla nekompatibilita mezi knihovnou a databází. Proto jsem byl donucen použít nativní knihovnou *Neo4j.Driver* (k dispozici na [40]) pro připojení k databázi přímo od vývojářů Neo4j, která již kompatibilní byla. Aby bylo možné využívat knihovnu asynchronně, nainstaloval jsem také doplňující knihovnu *Neo4j.Driver.Simple*, která poskytuje přetěžení metod v hlavní knihovně.

6.4.1 Popis knihovny Neo4j.Driver

Díky této importované knihovně se může v projektu využívat instance objektů, jako je konektor *IDriver*, relace *ISession* a transakce *ITransaction*. Lze si povšimnout, že v názvech se objevuje konvence pro rozhraní. Je tomu tak, protože autoři knihovny dodrželi tzv. „Dependency inversion principle“ (více informací na [41]).

IDriver představuje hlavní přístupový bod k databázi. Spravuje všechna připojení k databázi a stará se o jejich nejvyšší logiku (např. zahájení, ukončení spojení nebo řešení jeho pádu). *ISession* můžeme definovat jako kontejner pro jednotlivé transakce, které jsou v něm prováděný za sebou. *ISession* může vždy hostovat pouze jednu *ITransaction* v jednom okamžiku. Tento transakční objekt pak představuje jednu nebo více atomických operaci nad databází, uvnitř kterých se specifikují konkrétní dotazy. Obsahuje také metody *Run()*, *Commit()* a *RollBack()*, které slouží k spuštění dotazu, potvrzení všech změn provedených uvnitř transakce nebo navrácení databáze do stavu před započetím transakce.

V momentě, kdy se uvnitř relace (*ISession*) otevře nová transakce, požádá si relace o volné připojení od konektoru, který jej vybere a přidělí ze své kolekce aktivních připojení. Po skončení transakce relace toto připojení opět uvolňuje. To znamená, že při nečinnosti relace (žádná transakce zrovna není aktivní a program např. vykovává vnitřní výpočet) se nevyužívá žádných síťových prostředků. Více informací o knihovně lze vyčíst z oficiální dokumentace: [42].

6.4.2 Importování podgrafů

Knihovna používá příkaz *GraphDatabase.Driver()* k vytvoření nové instance *IDriver*. Jako parametry přijímá převážně URL adresu serveru a autentizační údaje.

Nicméně lze přidat další parametry jako třeba konfigurační nastavení (timeout, logging, maximální velikost buffer apod.). Tuto metodu jsem implementoval uvnitř metody *Neo4jDriver.Initialize()*, kde se zároveň s vytvořením *IDriver* instance inicializuje také konkrétní počet (podle počtu vytvořených vláken) instancí *ISession* pomocí metody *IDriver.Session()*. Tato metoda se v hlavním programu volá ještě před inicializací všech vláken.

Díky tomu může vytvořené vlákno přistupovat k již vytvořeným relacím a pouze provádět jednotlivé transakce. Při ukončení práce vlákna však relace není ukončena, protože čeká až ji přidružené vlákno bude používat znovu při zpracování dalšího vstupního souboru.

Ve třídě *Neo4jDriver* jsem dále vytvořil metodu *ImportNodeGraphs*. Tato metoda přijímá kolekci podgrafů, které jsou výstupem aplikace všech uživatelských filtrů. Tyto podgrafy se v metodě proiterují a pro každý podgraf se vytvoří nová transakce, uvnitř které se do databáze odešlou dotazy pro vytvoření všech uzlů v podgrafu a vztahů mezi nimi. Pokud všechny exekuce dotazů proběhly úspěšně, zavolá se metoda *tx.Commit()* pro potvrzení a uložení změn. V opačném případě se zachytí výjimka a zavolá se *tx.RollBack()*.

Příklad importu *Executable* uzlu a jeho vztahů je uveden v příloze E.3. Jednotlivé dotazy jsou napsané v jazyce *Cypher*. Obsahují klíčová slova jako je MATCH, MERGE (který představuje notaci MATCH OR CREATE), CREATE, WHERE, FOREACH a RETURN. Tomuto dialektu se již věnovala kapitola 2.2.3, nově však lze ve výpise vidět mechanismus parametrů. Ten je implementován pomocí samotných transakcí, které je přijímají jako druhý parametr v metodě *tx.Run()* (první parametr je samotný dotaz). Uvnitř dotazu se specifikuje parametr s předložkou znaku dolaru (např. „$rootNodeID“) a k transakci se připojí nový anonymní objekt, který atribut „rootNodeID“ specifikuje.

Jelikož byla v kapitole 6.3.4 uvedena skutečnost, že jeden uzel může být součástí více podgrafů, vytvořil jsem pro tyto uzly speciální třídu *NodeState*. Uvnitř jsou logické hodnoty pro import uzlu a import jeho vztahů. Jednotlivé instance stejného uzlu, které jsou obsaženy ve více jak jednom podgrafu, obsahují referenci na objektu typu *NodeState*. Tato reference představuje něco jako komunikaci mezi instancemi, ve které si můžou navzájem specifikovat, zda už byly naimportovány a tím pádem není potřeba je importovat podruhé.

6.5 Optimalizace nástroje

Na konci bakalářské práce jsem se primárně zaměřil na optimalizaci nástroje. Cíl stanovený konzultantem Ing. Miroslavem Drbalem specifikuje, že by nástroj měl zvládnout zpracovávat, filtrovat a importovat nejméně 200 000 XML souborů za 1

den. Z toho plyne požadovaná minimální rychlost cca 2,32 XML souborů za vteřinu. Vzhledem k tomu, že filtry budou psané uživatelem, zaměřil jsem se primárně na optimalizaci načítání a importu dat. V následujících podkapitolách je popsáno několik předmětů optimalizace, kterými jsem se zabýval.

K většině optimalizací jsem využíval vlastní logování hodnot do konzole za běhu nástroje nebo jsem využíval tzv. „profilování“. Výstupem tohoto procesu je detailní, rozsáhlý rozbor běhu programu, ze kterého můžeme vyčíst převážně kolik která metoda alokovala paměti nebo také hlavně, kolik milisekund která metoda trvala. Díky tomuto jsem byl schopen najít místa v projektu, která trvala nejdéle a ty jsem se snažil zoptimalizovat. Jak provést profilování projektu je uvedeno ve výpise E.2.

6.5.1 List vs HashSet/Dictionary

Hned první předmět optimalizace se týká spíše obecného principu, než reakce na logování/profilování projektu. Vzhledem k tomu, že se výpočetní a paměťové prostředky dají vždy dynamicky navýšit, zvolil jsem strategii obětování těchto prostředků ve prospěch lepší časové složitosti (její notaci *Omikron* – tzv. „vyjádření nejhoršího případu“).

Příkladem může být např. kolekce *OutgoingRelationships* každého hlavního uzlu. Jelikož součástí načítání je také oprava vztahů, ukazujících na chybějící uzly (viz kapitola 7), je nutné v této kolekci vyhledávat podle cílového uzlu, na který tyto vztahy ukazují. Proto jsem namísto kolekce typu *List<Relationship>* seskupil jednotlivé vztahy podle cílového uzlu a vytvořil z nich kolekci typu *Dictionary<string, List<Relationship>>*, kde klíč slovníku představuje identifikaci cílového uzlu a jako hodnota ve slovníku je kolekce vztahů ukazující na daný cílový uzel.

Namísto procházení seznamu a filtrování podle cílového uzlu (složitost O(n)) je tedy možné vyhledat požadovanou kolekci pomocí klíče ve slovníku (složitost O(1)). Dalším příkladem může být např. kolekce *Characteristics*, na kterou se ve filtru *UntrustedSubgraphsFilter* často ptáme, jestli obsahuje řetězec *TRUSTED*. Proto jsem kolekci implementoval jako *HashSet*, čímž jsem také složitost O(n) zmenšil na O(1).

6.5.2 If-else vs switch

Hned při první profilaci jsem si všiml, že po značnou dobu běhu nástroje probíhá metoda označená jako *<PrivateImplementationDetails>:ComputeStringHash* (až 15 % celkového běhu aplikace). Zprvu jsem si myslel, že se jedná o vytváření hashe použitého v kolekcích *HashSet* a *Dictionary*. Nicméně metodou pokus-omyl jsem zjistil, že tuto funkci využívá právě *switch* použitý při načítání dat ze vstupních XML souborů. Proto jsem tedy switch nahradil dlouhým if-else blokem, který postupně porovnává vstupní řetězec s řetězci, kteří byli definovaní v jednotlivých

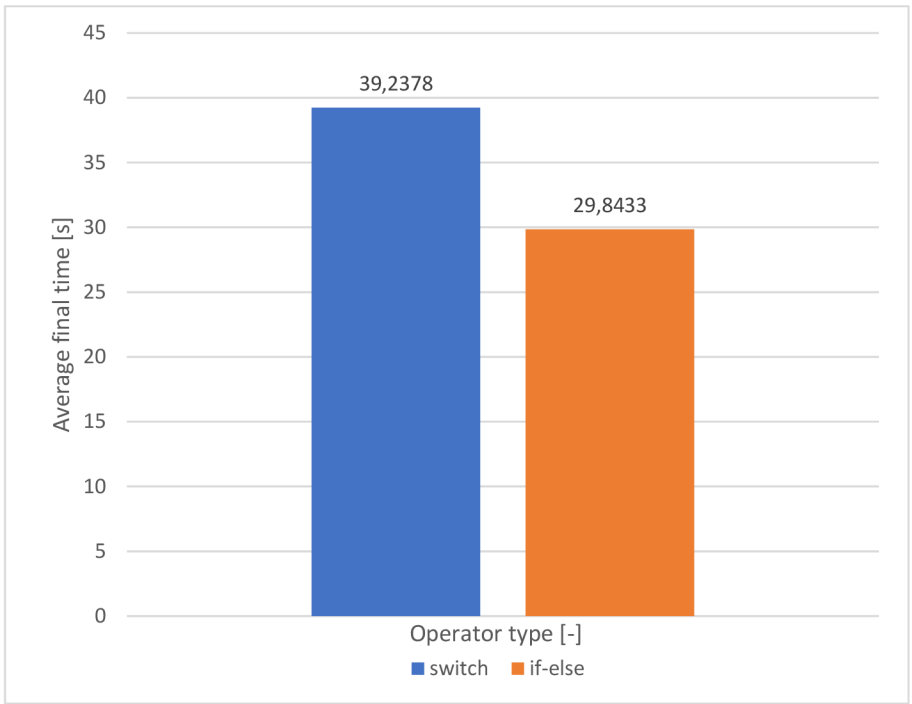
case. V nové profilaci se již počítání hashe nevyužívá. Namísto toho se objevila nová metoda *String.op_Equality*, která však nyní zabírá okolo 5 % celkové doby běhu nástroje.

Toto chování notace *switch* je odůvodněno základním chováním C# kompilátoru. Ten si při velkém počtu případů k porovnání nejprve interně vytvoří slovník, pomocí kterého poté přistupuje k jednotlivému *case*. Mezi dvěma způsoby načítání dat jsem také provedl měření, při kterém jsem importoval 400 XML souborů (různých velikostí).

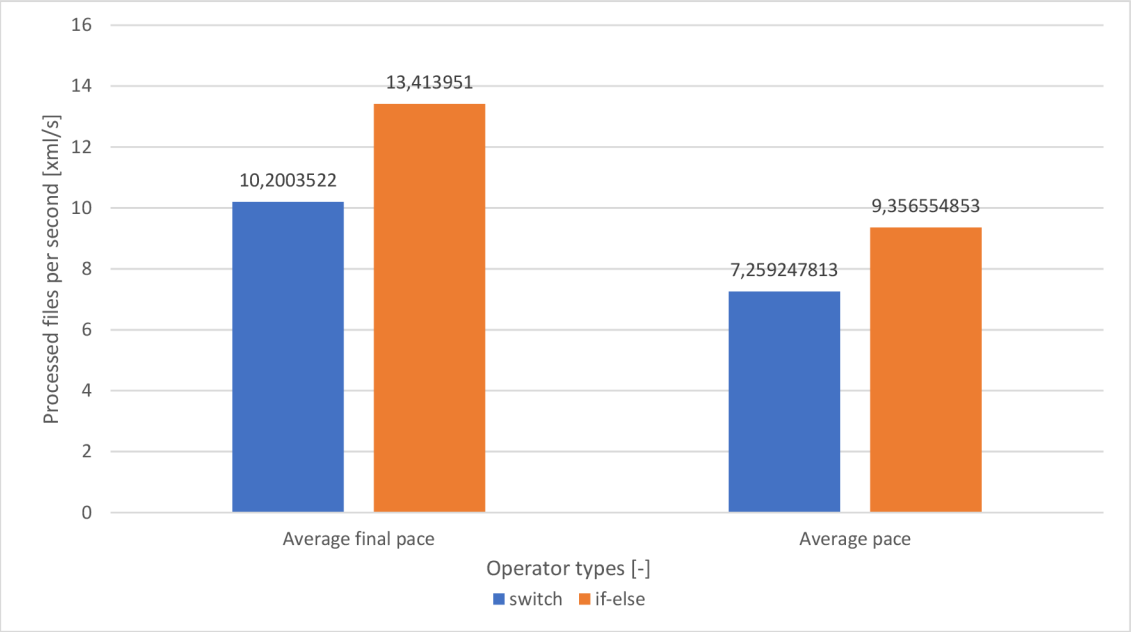
Měřené veličiny jsou *Final time [ms]* (jak dlouho celkově trvalo načíst, zpracovat a naimportovat soubory), *Pace [xml/s]* a *Final pace [xml/s]*. Obě veličiny představují rychlost načítání, nicméně se liší v době naměření – *Pace* se zaznamenává průběžně pro každý soubor, zatímco *Final pace* se zachytí pouze na konci. *Final pace* tak odpovídá vzorci:

$$\frac{400 * 1000}{Final\ time}$$

Měření jsem provedl celkem 12-krát pro oba způsoby. Z naměřených *Pace* jsem nejdříve vytvořil aritmetické průměry, abych u všech veličin dostal přesně 12 hodnot. Následně jsem vždy odstranil nejmenší a největší hodnotu (pro odstranění případných anomálií) a vytvořil aritmetický průměr z deseti zbývajících hodnot. Tyto průměry jsem zaznamenal do názorných grafů 6.6 a 6.7.



Obr. 6.6: Vliv typu operátoru (if-else vs switch) na výsledném čase.



Obr. 6.7: Vliv typu operátoru (if-else vs switch) na výsledných průměrných rychlostech.

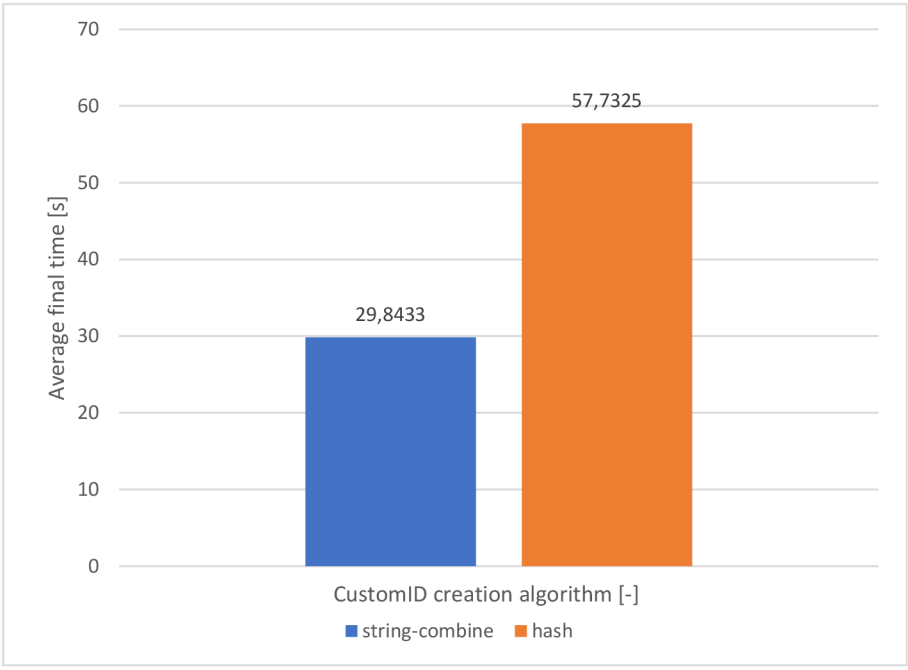
Z výsledků měření a také z profilování jsem došel k závěru, že využití bloků *if-else* je v rámci optimalizace efektivnější.

6.5.3 CustomID jako výsledek hashovací funkce

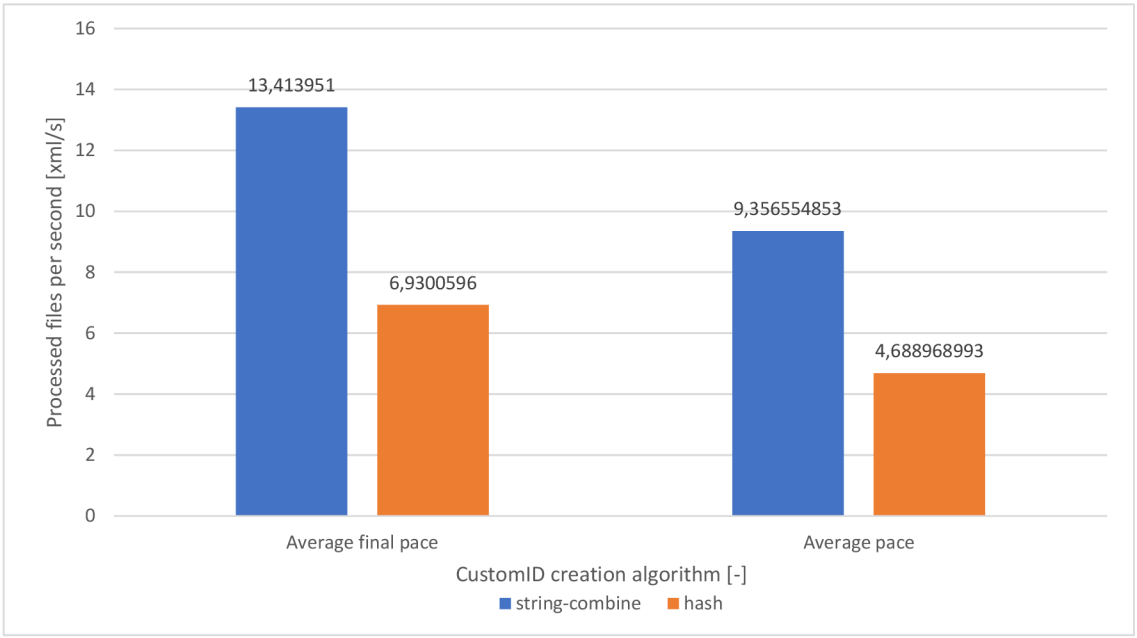
Jak bylo nastíněno již v kapitole 6.3.1, *CustomID* je atribut, který jednoznačně identifikuje uzel v celé grafové databázi. Původní *NodeID* je identifikátor v rámci jednoho souboru, proto jsem uznal za vhodné tuto hodnotu zkombinovat s názvem samotného XML souboru (ten představuje hodnotu hash funkce, na jejímž vstupu je obsah celého souboru).

Přišlo mi vhodné zachovat formát *CustomID* v podobě hashe, proto jsem integroval tzv. *xxHash* [43]. Tato funkce na vstupu přijímala původní identifikátor uzlu a jednoznačný identifikátor souboru. I přestože má být tento algoritmus jeden z nejrychlejších hash funkcí (až 5,4 GB/s [44]), vypozoroval jsem po jeho integraci zpomalení. Toto zpomalení bylo znatelné i v rámci profilování.

Alternativou pro vytvoření *CustomID* je tedy pouze spojení řetězců dohromady s podtržítkem sloužícím jako oddělovač (např. „572_0adf18bcdc50f86312e888e46e2c08e5ad9903bcde2e5041b0f86f40ab8da599“). Provedl jsem tedy stejné měření jako v předchozí kapitole, jenom tentokrát jsem měřil dva způsoby tvorby atributu *CustomID*. Z výsledků měření v grafech 6.8 a 6.9 plyne, že implementace hashovací funkce má negativní vliv na optimalizaci nástroje. Proto jsem se rozhodl zůstat u skládání řetězců.



Obr. 6.8: Vliv použitého algoritmu pro generování CustomID na výsledném čase.



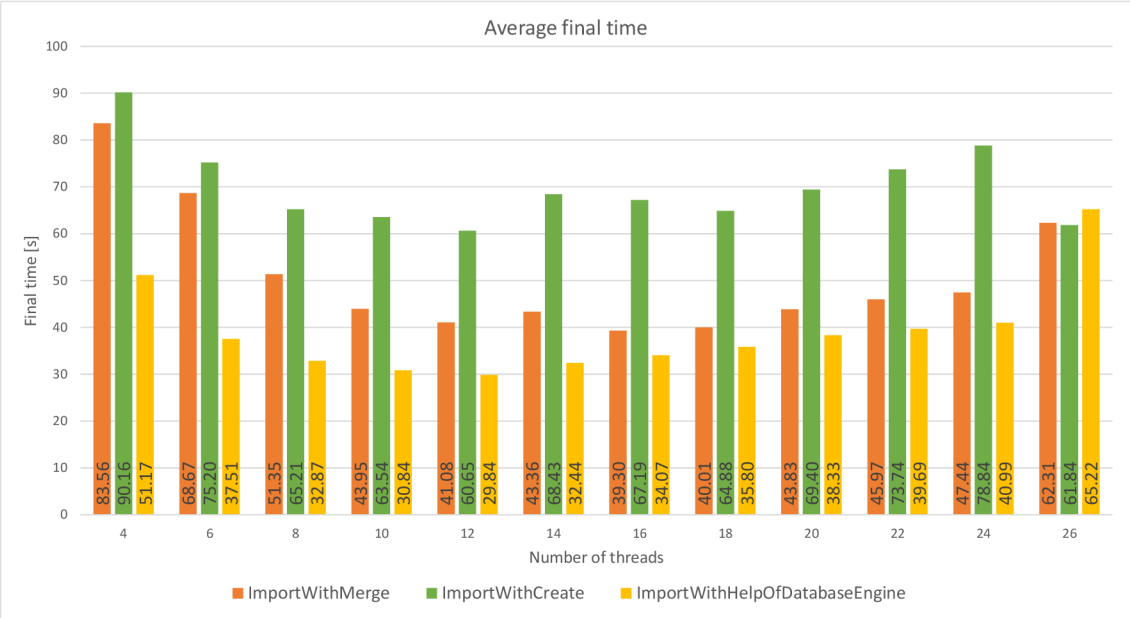
Obr. 6.9: Vliv použitého algoritmu pro generování CustomID na výsledných průměrných rychlostech.

6.5.4 Konfigurovatelný počet jader a logika importu dat

Posledním tématem optimalizace je možnost dynamicky měnit počet jader, s čímž úzce souvisí i logika importu dat. Při implementaci importu dat jsem navrhl až 5 metod, které se prakticky mezi sebou liší hlavně postupem a formou dotazů. Během vývoje jsem však zjistil, že kvůli různým strukturálním změnám dvě z nich nebudou nadále funkční. Proto se tato kapitola bude věnovat pouze třem:

- *ImportWithMerging*
  - logika spočívá v iteraci všech kolekcí a jejich vnořených kolekcí (např. iterace všech *Executable* uzlů a vnořená iterace jejich *Connections* kolekce, atd.)
  - v každé této iteraci se pomocí transakce volá dotaz pro vytvoření daného uzlu a napojení na jeho hlavní uzel
  - 100 % iterací se tedy provádí přímo v běhu nástroje
  - hlavní uzly se však vytváří pomocí příkazu MERGE, který podle *CustomID* buď najde daný uzel nebo jej vytvoří zároveň s ostatními parametry
  - na konci importu jednoho hlavního uzlu se projedou všechny jeho odchozí vztahy a naimportují se do databáze – pokud uzel s cílovým *CustomID* v databázi ještě nebyl naimportován, vytvoří se, ale bez parametrů (proto se v předchozím bodě využívá MERGE)
- *ImportWithCreating*
  - tato metoda je velmi podobná té předchozí
  - namísto vytváření hlavního uzlu pomocí MERGE se však vytvoří pomocí příkazu CREATE, který by podle oficiální dokumentace měl být efektivnější
  - všechny vztahy mezi uzly se vždy naimportují až když už jsou všechny hlavní uzly v databázi
  - nevýhodou však je, že při importu vztahu musíme nyní podle *CustiomID* najít nejen cílový uzel, ale i uzel zdrojový (hledání uzlu podle CustomID se provádí pomocí indexu)
- *ImportWithHelpOfDatabaseEngine*
  - v předchozích metodách byla veškerá iterace prováděna pouze nástrojem
  - tato metoda využívá příkazů FOREACH pro „přesun“ některých vnořených iterací na databázový engine
  - nástroj tak určitou část práce ponechá na databázi a může se rychleji věnovat dalším uzlům
  - jelikož databáze běží na stejném systému jako nástroj pro zpracování dat, začal jsem se zabývat vztahem mezi výkonem databáze a počtem přidělených vláken

Jak již bylo zmíněno, k dispozici jsem měl virtuální kontejner obsahující 16 logických jader. Začal jsem tedy provádět měření jednotlivých metod, během kterého jsem v konfiguračním souboru měnil počet přidělených vláken. Tento počet začal na 4 vláknech a po dvou se postupně zvyšoval až na 26. V tomto rozsahu jsem měl možnost pozorovat vztah efektivity nástroje ku počtu přidělených vláken. Pro měření jsem použil 1000 XML souborů, které jsem celkem 12-krát naimportoval vždy pro jedno nastavení počtu vláken. Výsledky (zpracované stejným způsobem jako v předchozích podkapitolách) jsou znázorněny v grafech 6.10, 6.11, 6.12.

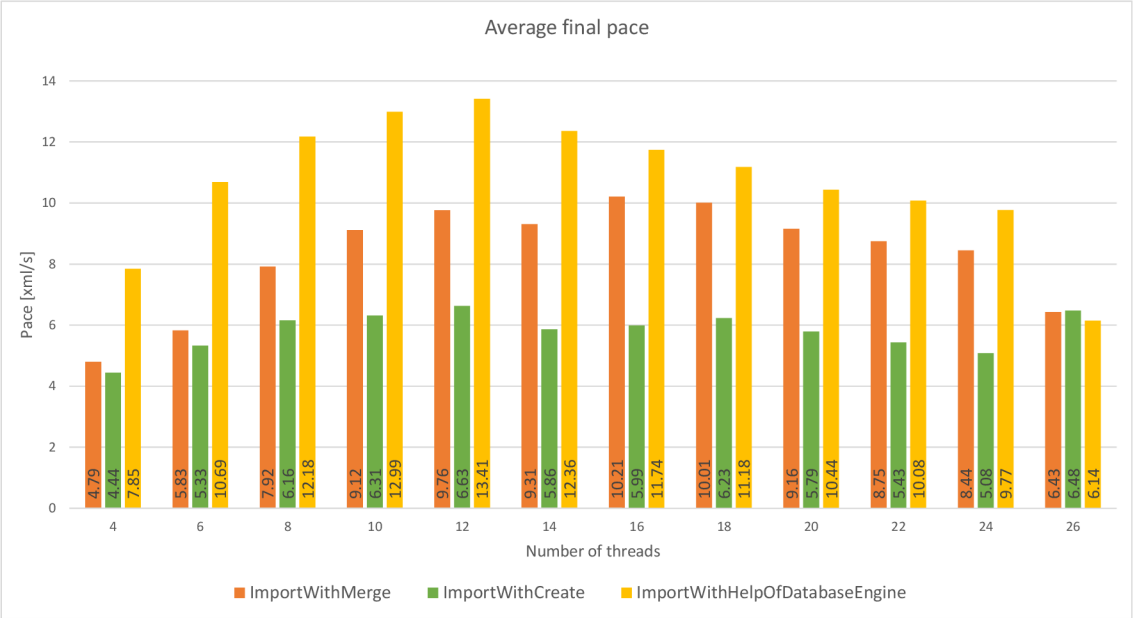


Obr. 6.10: Vliv metody importu a vlákna na výsledném čase.

Z grafů je možné vypozorovat hned několik poznatků. Metoda *ImportWithCreate* měla nejhorší výsledky téměř při všech počtech vláken. Naopak na tom byla metoda *ImportWithHelpOfDatabaseEngine*. Z toho plyne, že využití databázového engine k částečnému importu dat se vyplatí a tudíž jsem tuto metodu jako finální řešení importu dat.

Nutno znovu podotknout, že počet logických jader na stroji byl 16. To znamená, že při nastavení 16 vláken teoreticky připadá jeden logický výpočetní prostředek na jedno vlákno. Proto můžeme pozorovat u metody *ImportWithHelpOfDatabaseEngine* maximální výkon u 14 vláken a ne u 16. Je to z toho důvodu, že samotný databázový engine také potřebuje výpočetní prostředky ke svému běhu a ke své části importu. Při vyšších počtech vláken výkon začíná klesat, protože plánovač úloh na systému začíná být zahlcován požadavky od více vláken, než je logických jader k dispozici.

Obr. 6.11: Vliv metody importu a vlákna na výsledné průměrné rychlosti.



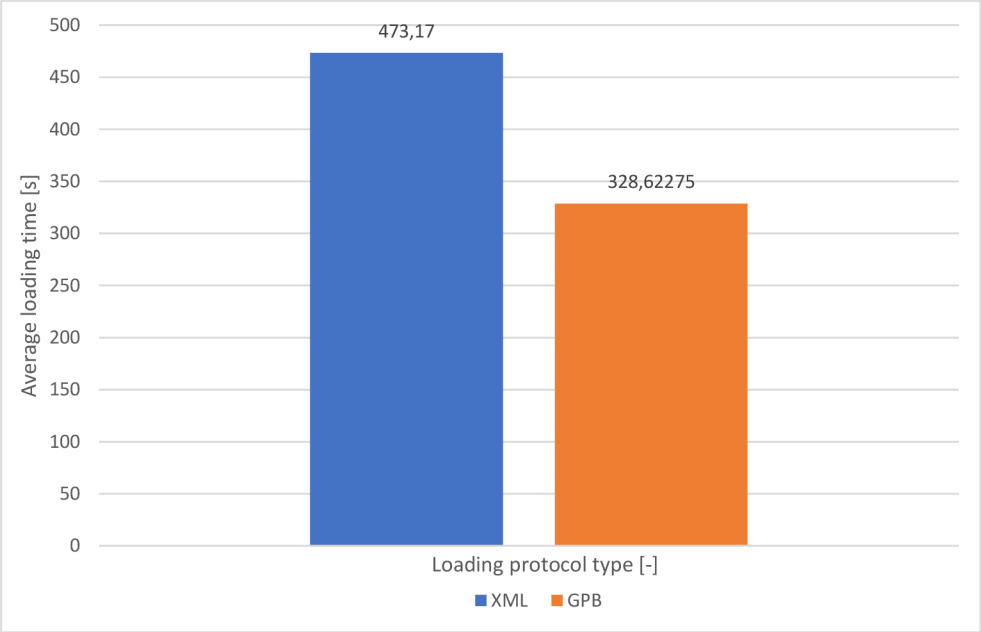
Obr. 6.12: Vliv metody importu a vlákna na výsledné konečné rychlosti.

6.5.5 Google Protocol Buffers

Tato metoda pro serializaci a deserializaci dat je přímý konkurent formátu XML. Téměř na konci akademického roku provedl nejmenovaný zaměstnanec společnosti Avast Software s.r.o. měření, ze kterého vyšlo najevo, že načítání binárních dat ve formátu Google Protocol Buffer (výstupní soubor serializace) a jejich deserializace vykazuje mnohem efektivnější výsledky, než načítání XML souborů pomocí objektu *XmlReader*.

Problém však je, že behaviorální štít nyní umí generovat data pouze ve formátu XML. Generování již serializovaných *GPB* souborů by se na něj muselo nejprve naimplementovat. Jako „Proof Of Concept“ jsem si připravil stejných 400 XML souborů a nechal jsem z nich externím nástrojem vytvořit binární *GPB* soubory. Následně jsem ve svém nástroji dočasně pozastavil načítání dat pomocí *XmlReader* a vytvořil nový způsob načítání dat do *NodeGraph* z těchto souborů.

Poté jsem provedl nové měření. Tentokrát měřenou veličinou byl pouze čas potřebný k načtení dat ze souboru. Měření jsem opět provedl 12-krát pro obě metody. Z každého měření jsem vytvořil aritmetické průměry, ze kterých jsem odebral nejmenší a největší hodnotu a zbývající hodnoty následně znovu zprůměroval a zaznamenal do grafu 6.13.



Obr. 6.13: Vliv zvoleného protokolu na rychlost načítání dat

Z něj plyne závěr, že i přestože byla metoda pro načítání z *GPB* implementována pouze experimentálně bez jakékoliv optimalizace, překonala původní načítání z XML souboru v rychlosti načítání. To znamená, že by vývoj nástroje i behaviorálního štítu tímhle směrem mohl v budoucnu přinést podstatně lepší efektivitu nástroje (přibližně 30 a více %).

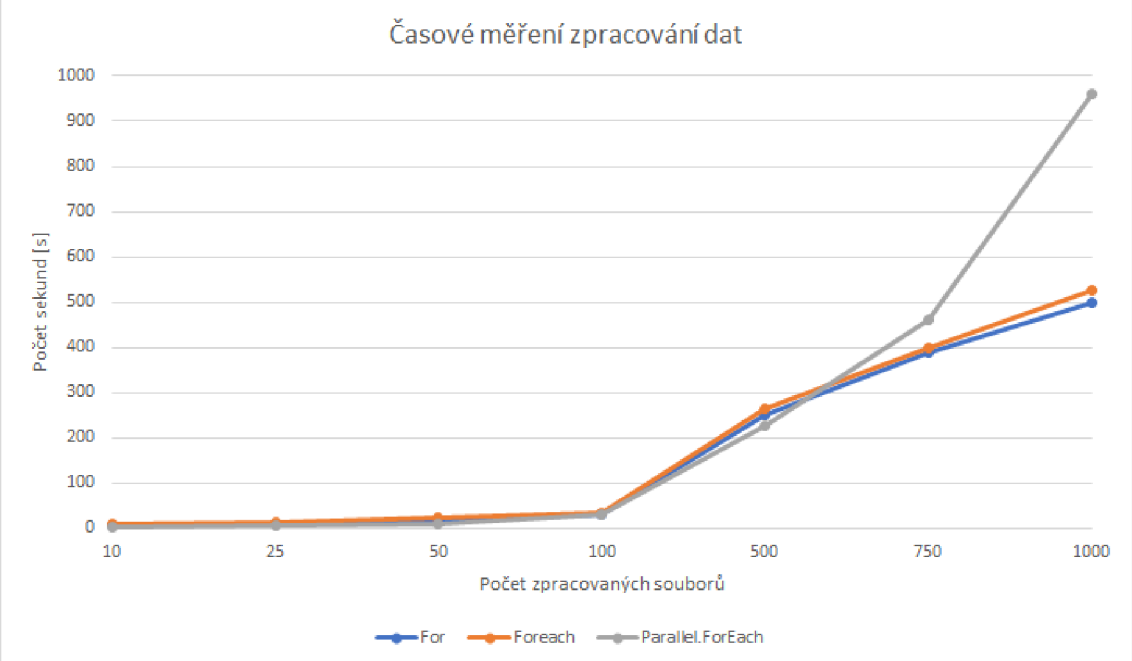
7 Problémy vzniklé v průběhu implementace

Během práce na praktické části bylo objeveno několik zásadních problémů, které bránily v jejím pokračování. V této kapitole jsou tyto problémy popsány a uvedeny jejich řešení.

Zpracování všech souborů (první fáze nástroje)

Samotný cyklus procházení všemi XML soubory byl postupně upravován a měřen. Jako první myšlenka byla provádět tento cyklus paralelně. Nejméně náročný způsob, jak tohoto docílit, je použít nativní knihovnu *Parallel*, konkrétně ***Parallel.ForEach***. Nicméně výsledky nebyly ani trochu uspokojivé (všech 17 961 souborů trvalo zpracovat asi 11 hodin). Důvodem bylo, že se vzrůstajícím počtem zpracovaných souborů bylo pro systém stále složitější paralelně přerozdělovat prostředky mezi procesory.

Paralelní zpracování dat může být při velkém množství kontraproduktivní, proto byly vyzkoušeny také neparalelní metody procházení. Nejdříve byla změřena metoda **foreach**, která přinesla příznivější výsledky. Po prvních 1000 souborech byl uplynulý čas téměř poloviční. Klasická metoda **for** a přístup ke kolekci přes indexy byl o trochu rychlejší. Důvodem je potřeba získat procházející iterátor u metody **foreach**. V grafu 7.1 je toto měření naznačeno. Měření času probíhalo v době, kdy počet zpracovaných souborů přesáhl hodnoty 10, 25, 50, 100, 500, 750 a 1000.



Obr. 7.1: Vztah mezi metodou zpracování a uplynulým časem

Nakonec se podoba programu vrací zpátky k paralelnímu zpracování, avšak namísto **Parallel.ForEach** je nyní použita série tzv. **ManualResetEvent** a **ThreadPool.QueueUserWorkItem**, u kterých je vlastnoručně nakonfigurována mechanika čekání na volné vlákno procesoru. Měření v tomhle případě již neproběhlo, ale celkový čas pro zpracování všech 17 961 zabere očekávané cca 3,5 hodiny, což představuje optimální výsledek.

Načítání *NodeGraph* z XML

Původní nástroj pro čtení XML dat ze souboru byl **XmlDocument**, který pomocí **XmlDocument.Load()** dokázal načíst všechna data, která se následně postupně naplňovala do atributů objektů. Tento nástroj byl následně změněn na **XmlReader**, protože vykazoval lepší průměrné výsledky (okolo 45 % rychlejší) při zpracování jednoho souboru. V prvním případě byl průměrný čas 296,3 sekund, v druhém případě 165,268 sekund (měřeno pětkrát pro každou metodu, měření zastaveno po načtení 50 souborů). Důvodem je fakt, že objekt **XmlReader** poskytuje pouze jednocestné čtení souboru, představuje tak určitý „stream dat“.

Transport nástroje na linuxovou distribuci

Na poskytnutém kontejneru byla nainstalována aplikace *Mono* a pomocí aplikace *Git* naklonován celý projekt. Nicméně při snaze spustit program v *Mono* nefungovalo načítání referenčních kompilačních souborů. Problém byl v tom, že *Mono* je hledal u sebe ve vlastní knihovně, nicméně tyto soubory se nacházejí ve složce projektu. Proto bylo třeba tuto cestu přidat před názvy referenčních souborů přímo v programu. Následně chybělo několik „.dll“ souborů, které bylo třeba doinstalovat pomocí příkazu `nuget restore Nebula.sln`.

Problém se systémovými limity (první fáze nástroje)

Jelikož se všechny soubory nejdříve mapují do paměti a posléze se s nimi pracuje, nástroj si v systému vyhrazuje nadměrné množství paměti. Program nejdříve v průběhu padal na chybovou hlášku `mono_gdb_render_native_backtraces not supported on this platform, unable to find gdb or lldb`. Po doinstalování knihoven *gdb* a *lldb* program stále padal s chybovými hláškami, že se nepodařilo alokovat místo v paměti i přestože systém volnou paměť ještě měl. Toto bylo vyřešeno po navýšení limitu na maximální počet mapovaných souborů jedním procesem pomocí příkazu `sysctl -w vm.max_map_count=1000000`.

Konfigurace generování CSV souborů (první fáze nástroje)

Tento proces potřebný k importu dat do různých databází zabral nejvíce času z důvodu velkého množství dat. Nativní nástroj pro import do *Neo4j* databáze vždy vyhodil drobnou chybovou hlášku, např. že na nějaké pozici je špatně strukturovaný řádek, který nesouhlasí s hlavičkou. Většina těchto problémů souvisela s tím, že v samotném textovém řetězci nějakého atributu byl obsažen znak pro oddělení sloupce v CSV nebo znak uvozovek. Bylo vždy třeba upravit tuto generaci v programu, exportovat data a zkusit je znovu importovat do databáze. Import do ostatních databází probíhal relativně v pořádku, protože všechny problémy byly průběžně opraveny i v generování CSV souborů pro tuto databázi.

Zisk *TaggerResponse* je příliš pomalý

Proces „tagování“ spustitelného souboru probíhá pomocí HTTP_GET požadavku na interní server od společnosti Avast Software s.r.o. Poskytnutý kontejner obsahující linuxovou distribuci však nemá povolen přístup do sítě, kde se tento server nachází. Proto je třeba pro správné odeslání požadavku a zisk odpovědi nastavit SSH tunel pro port 80 na IP adresu serveru. Nicméně toto tunelování zabírá příliš mnoho času. Přes SSH tunel přijetí odpovědi trvá průměrně 5 sekund zatímco při puštění programu ve virtuální soukromé síti zabere tento požadavek asi 20 milisekund. Proto je momentálně tato operace vypnutá a bude se v budoucnu pracovat na přidání kontejneru do stejné sítě, aby si požadavky mohl vyřizovat sám.

Chybějící vztahy ve vstupních datech

Po načtení vstupních souborů do paměti se občas mohlo stát, že nástroj padal na chybě, kdy např. cílové ID nějakého vztahu není obsaženo v kolekci všech uzlů. V praxi to vypadá tak, že uzel v jednom souboru ukazuje hranou na uzel, který ve stejném souboru není. Tento stav je anomálie a měl by se odstranit. Dalším možným nežádoucím stavem je, když existují nesrovnalosti mezi odchozími a příchozími vztahy. Např. uzel A obsahuje odchozí vztah k uzlu B, ale uzel B neobsahuje stejný příchozí vztah od uzlu A.

Proto jsem ihned po fázi načtení dat implementoval metodu *FixRelationships()*. Tato metoda iteračně projede všechny uzly a jejich kolekce vztahů. U každého vztahu zkontroluje, jestli se dané *CustomID* nachází v kolekci všech uzlů. Jestli ne, vztahy přidružené k tomu uzlu jednoduše odstraní. V opačném případě se však ještě musí zkontrolovat příchozí a odchozí vztahy. K tomuto se využívá LINQ notace *.Except()*, která provede rozdíl dvou množin. Tento rozdíl se následně doplní do správné kolekce a tím se nesrovnalost opraví.

Díky využívání slovníku jako kolekce se vztahy, je tato metoda extrémně efektivní a téměř všechny její operace mají konstantní časovou složitost.

Neplatné znaky ve vstupních datech

Ve vstupních souborech se občas mohou objevit znaky, které *XmlReader* nedokáže přečíst. Jedná se o kontrolní znaky, které se v souborech vyskytnou omylem (pravděpodobně v rámci generování telemetrických dat). Takové soubory nástroj zatím ignoruje a neimportuje je, neboť se zatím i samotná společnost Avast musí rozhodnout, v jaké fázi se tyto chyby budou opravovat (jestli až při zpracování nebo už při generování, apod.).

Druhým typem chybových vstupních souborů jsou takové, které jsou kompletně prázdné. Tyto soubory nástroj automaticky maže.

Závěr

Během bakalářské práce jsem se úspěšně seznámil s problematikou zadání a nastudoval relevantní teoretická témata. Vybral jsem a implementoval pět databázových systémů, zvolil vhodné metriky pro měření jejich výkonnosti a vytvořil nástroj pro zpracování, filtraci a import vstupních dat.

Na začátku práce je čtenář obeznámen s tématy relační a grafové databáze. Dále s problematikou behaviorální detekce a vícevláknové aplikace. Následně je v práci popsán návrh struktury grafové databáze na základně vstupních dat a implementace vhodných zástupců databázových systémů. V dalších kapitolách se práce věnuje návrhu a implementaci nástroje pro zpracování dat, která jsou generována behaviorálním štítem. Jeho úkolem je především efektivní načtení, filtrace a import telemetrických dat. Následně je v práci také popsán proces optimalizace tohoto nástroje pro zvýšení celkové rychlosti zpracování dat. V poslední kapitole jsou vypsány také problémy, které se během implementace vyskytly a jejich řešení.

Výsledkem práce je funkční nástroj pro zpracování a filtraci dat, nainstalované databázové řešení od pěti různých výrobců a jejich měření výkonnosti, využití místa na disku a v paměti. Na základě tohoto měření jsem včetně dalších faktorů (např. kvalita dokumentace, podpora vývojářů, komunita) zvolil databázový systém Neo4j. Následně jsem nástroj pro zpracování dat plně optimalizoval s ohledem na přímé spojení s databází, pomocí kterého nyní probíhá „real-time“ import. Dále jsem také úspěšně implementoval uživatelský filtr pro klasifikaci podgrafů vycházejících z nedůvěryhodných procesů. Tento filtr zároveň pročišťuje graf od osamocených či nedosažitelných uzlů.

Během fáze optimalizace jsem jednotlivými změnami postupně zvyšoval efektivitu nástroje až o desítky procent. Finální verze aplikace tak zpracovává data o 465,2 % rychleji, než je konzultantem stanovené kritérium. Existuje tedy prostor pro další, časově náročnější filtrace. Nad rámec zadání jsem také provedl základní „proof of concept“ ukazující, že by bylo vhodné v budoucnu implementovat *Google Protocol Buffers* namísto XML jako základní formát telemetrických dat.

Závěrem lze konstatovat, že výsledky práce plně splnily zadání a výsledné řešení bude nadále vyvíjeno a využíváno společností Avast Software s.r.o.

Literatura

[1] The IBM Punched Card. *IBM* [online]. USA: IBM100, 2012 [cit. 2019-10-27]. Dostupné z: https://www.ibm.com/ibm/history/ibm100/us/en/icons/punchcard/

[2] CODD, E. F. A relational model of data for large shared data banks .*Communications of the ACM* [online]. 13(6), 377-387 [cit. 2019-10-27]. DOI: 10.1145/362384.362685. ISSN 00010782. Dostupné z: http://portal.acm.org/citation.cfm?doid=362384.362685

[3] SUMATHI, S. a S. ESAKKIRAJAN. *Fundamentals of relational database management systems*. London: Springer, 2007. ISBN 3540483977.

[4] The executive's guide to oracle applications. *Profit Magazine* [online]. 2007, 12(2), 26–29 [cit. 2019-10-27]. Dostupné z: http://www.oracle.com/us/corporate/profit/p27anniv-timeline-151918.pdf

[5] KOŠÁREK, Lukáš. *Výkonnostní srovnání relačních databází* [online]. Brno, 2011 [cit. 2019-10-27]. Dostupné z: https://is.muni.cz/th/wox38/. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Vlastislav Dohnal.

[6] BÍNA, Josef a Karel ŠOTEK. Relační databáze letních táborových her. Univerzita Pardubice, 2012. Dostupné z: https://dk.upce.cz//handle/10195/45948

[7] Relační databáze. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001– [cit. 2019-10-27]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Rela%C4%8Dn%C3%AD_datab%C3%A1ze&oldid=17636548

[8] GRAY, Jim, et al. The transaction concept: Virtues and limitations. In: *VLDB*. 1981. p. 144–154. Dostupné z: https://www.hpl.hp.com/techreports/tandem/TR-81.3.pdf

[9] HAERDER, Theo a Andreas REUTER. Principles of transaction-oriented database recovery. *ACM Computing Surveys* [online]. 15(4), 287–317 [cit. 2019-10-27]. DOI: 10.1145/289.291. ISSN 03600300. Dostupné z: http://portal.acm.org/citation.cfm?doid=289.291

[10] YAOWEN, Chen. *Comparison of Graph Databases and Relational Databases When Handling Large-Scale Social Data*. Saskatoon, Saskatchewan Canada, 2016. Dostupné také z: https://pdfs.semanticscholar.org/a1a1/

78917f93c0eed6f444463da58da7bcbf43d3.pdf. A Thesis. University of Saskatchewan Saskatoo.

[11] AMSTERDAM, Jonathan. Atomic File Transactions, Part 1. In: *OnJava.com* [online]. Sebastopol, CA, USA: O'Reilly Media, 2001 [cit. 2019-10-27]. Dostupné z: https://web.archive.org/web/20160303090517/http://archive.oreilly.com/pub/a/onjava/2001/11/07/atomic.html

[12] Databázová transakce. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001– [cit. 2019-10-27]. Dostupné z: https://cs.wikipedia.org/wiki/Datab%C3%A1zov%C3%A1_transakce

[13] Database Indexes Explained. In: EssentialSQL [online]. Essential SQL Learning Group [cit. 2019-12-18]. Dostupné z: https://www.essentialsql.com/what-is-a-database-index/

[14] Index (databáze). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001– [cit. 2019-12-18]. Dostupné z: https://cs.wikipedia.org/wiki/Index_(datab%C3%A1ze)

[15] Index (databáze). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-12-18]. Dostupné z: https://en.wikipedia.org/wiki/Database_index

[16] P. VAGNOZZI, Paul. Database indexing method and apparatus. USA. US 2003/0135495 A1. Uděleno 17. červenec 2003. Zapsáno 21. červenec 2002.

[17] LITH, ADAM a JAKOB MATTSSON. Investigating storage solutions for large data: A comparison of well performing and scalable data storagesolutions for real time extraction and batch insertion of data [online]. Goteborg, Sweden, 2010 [cit. 2019-12-19]. Dostupné z: https://pdfs.semanticscholar.org/b8f6/b9d79c75e66c3b2f5034fe8172fd24cc0d13.pdf. Master of Science Thesis. Department of Computer Science and Engineering, CHALMERS UNIVERSITY OF TECHNOLOGY.

[18] ANGLES, Renzo. A Comparison of Current Graph Database Models. Camino Los Niches, Km. 1, Curic ó, Chile, 2012. Department of Computer Science, Engineering Faculty, Universidad de Talca.

[19] PALÍŠEK, Luboš. Možnosti využití grafových databázív informačním systému internetovéhoobchodu [online]. Praha, 2015 [cit. 2019-12-19]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/63008/F8-DP-2015-Palisek-Lubos-thesis.pdf?sequence=1&isAllowed=y. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

[20] XML RDF. W3schools [online]. [cit. 2019-12-19]. Dostupné z: https://www.w3schools.com/xml/xml_rdf.asp

[21] GILBERT, Seth a Nancy LYNCH. Brewer's Conjecture and the Feasibilty of Consistent, Available, Partition-Tolerant Web Services [online]. [cit. 2019-12-19]. Dostupné z: https://users.ece.cmu.edu/ãdrian/731-sp04/readings/GL-cap.pdf

[22] The Neo4j Cypher Manual v3.5. Neo4j [online]. Neo4j Team, 2019 [cit. 2019-12-19]. Dostupné z: https://neo4j.com/docs/cypher-manual/current/

[23] Dgraph Documentation. Dgraph [online]. San Francisco, 2019 [cit. 2019-12-19]. Dostupné z: https://docs.dgraph.io/

[24] SPARQL. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001– [cit. 2019-12-19]. Dostupné z: https://en.wikipedia.org/wiki/SPARQL

[25] JING HAN, HAIHONG E, GUAN LE a JIAN DU. Survey on NoSQL database. In: 2011 6th International Conference on Pervasive Computing and Applications [online]. IEEE, 2011, 2011, s. 363–366 [cit. 2019-12-19]. DOI: 10.1109/ICPCA.2011.6106531. ISBN 978-1-4577-0208-2. Dostupné z: http://ieeexplore.ieee.org/document/6106531/

[26] The Database Model Showdown: An RDBMS vs. Graph Comparison. In: Https://neo4j.com/ [online]. Neo4j Staff, 2015 [cit. 2019-12-19]. Dostupné z: https://neo4j.com/blog/database-model-comparison/

[27] XU, Yu, Victor LEE, Mingxi WU, Gaurav DESHPANDE a Alin DEUTCH. Native Parallel Graphs [online]. TigerGraph, 2018 [cit. 2019-12-19]. Dostupné z: https://info.tigergraph.com/ebook

[28] VLČEK, Ondřej. Behaviorální štít: nová ochranná vrstva v antiviru Avast 2017. In: Avast blog [online]. 2017 [cit. 2019-12-19]. Dostupné z: https://blog.avast.com/cs/behavioralni-stit-nova-ochranna-vrstva-v-antiviru-avast-2017

[29] MARGOLD, Tomáš. KLASIFIKACE PŘÍSPĚVKŮ VE WEBOVÝCH DISKUSÍCH [online]. Brno, 2008 [cit. 2020-05-15]. Dostupné z: https://core.ac.uk/download/pdf/44386915.pdf. Diplomová práce. Vysoké učení technické v Brně.

[30] WILLIAMSON, Matthew a Vladimir GORELIK. Method and apparatus for detecting harmful software. USA. US 8719924 B1. Uděleno 6. květen 2014.

[31] WILLIAMSON, Matthew a Vladimir GORELIK. Method and Apparatus for Removing Harmful Software. USA. US 20090049552 A1. Uděleno 19. únor 2009.

[32] WILLIAMSON, Matthew a Vladimir GORELIK. Method and apparatus for removing harmful software. Worldwide. WO 2007035417 A3. Uděleno 29. březen 2007.

[33] System Deadlocks. Computing Surveys [online]. 1971, 3(2), 12 [cit. 2020-05-26]. Dostupné z: http://www.ccs.neu.edu/home/pjd/cs7600-s10/Tuesday_January_26_01/p67-coffman.pdf

[34] Method of calculating the scores of the DB-Engines Ranking. In: DB-Engines [online]. [cit. 2019-12-19]. Dostupné z: https://db-engines.com/en/ranking_definition

[35] GNU General Public License v3 (GPL-3). In: TLDRLegal [online]. [cit. 2019-12-19]. Dostupné z: https://tldrlegal.com/license/gnu-general-public-license-v3-(gpl-3)

[36] Apache License 2.0 (Apache-2.0). In: TLDRLegal [online]. [cit. 2019-12-19]. Dostupné z: https://www.tldrlegal.com/l/apache2

[37] 2.9 Patterns. The Neo4j Operations Manual v4.0 [online]. Neo4j [online]. Neo4j, 2020 [cit. 2020-05-16]. Dostupné z: https://neo4j.com/docs/cypher-manual/current/syntax/patterns/

[38] 15.5 Memory recommendations. The Neo4j Operations Manual v4.0 [online]. Neo4j [online]. Neo4j, 2020 [cit. 2020-06-01]. Dostupné z: https://neo4j.com/docs/operations-manual/4.0/tools/neo4j-admin-memrec/

[39] Readify / Neo4jClient. GitHub, Inc. [online]. 2020 [cit. 2020-05-19]. Dostupné z: https://github.com/Readify/Neo4jClient

[40] Neo4j / neo4j-dotnet-driver. GitHub, Inc. [online]. 2020 [cit. 2020-05-19]. Dostupné z: https://github.com/Readify/Neo4jClient

[41] Dependency inversion principle. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-05-19]. Dostupné z: https://en.wikipedia.org/wiki/Dependency_inversion_principle

[42] The Neo4j Drivers Manual v4.0. Neo4j [online]. Neo4j, 2020 [cit. 2020-05-25]. Dostupné z: https://neo4j.com/docs/driver-manual/current/

[43] Cyan4973 / xxHash. GitHub, Inc. [online]. 2020 [cit. 2020-05-20]. Dostupné z: https://github.com/Cyan4973/xxHash

[44] XxHash [online]. [cit. 2020-05-20]. Dostupné z: https://cyan4973.github.io/xxHash/

Seznam symbolů, veličin a zkratek

ACID *Atomicity, Consistency, Isolation, Durability*

DB báze dat – *database*

DBMS *database management system* – systém řízení báze dat

Guid datový typ sloužící jako jednoznačný identifikátor

Integer velmi rozšířený datový typ uchovávající celé číslo

malware *malicious software* – škodlivý software

Seznam příloh

A Ukázka vstupních dat

A.1 Hlavní struktura vstupních dat ve formátu XML

Výpis A.1: Hlavička a definice *IDPAgentData*

```
<?xml version="1.0" encoding="utf-8"?>
<IDPAgentData version="1" osVersion="6.1.7601"
    platform="64b"
    agentVersion="19.8.3108"
    generateTime="2019-10-29T20:24:02+07:00"
    agentId="b7217a04-dbc3-4438-8674-9d7b8d949a1a"
    vpsVersion="19102800">
    <Nodes>
        ...
    </Nodes>
</IDPAgentData>
```

A.2 Uzel definovaný ve formátu XML

Výpis A.2: Hlavička a definice *Node*

```
<Node>
 <NodeId>3</NodeId>
 <NodeType>EXECUTABLE</NodeType>
 <NodeName>C:\PROGRAMFILES\COMMONFILES\MICROSOFTSHARED
     \OFFICE12\MSOXMLMF.DLL
    </NodeName>
 <SHA256/>
 <Characteristics>
  <Characteristic>
      IS_PLUGGABLE_PROTOCOL_HANDLER
     </Characteristic>
  <Characteristic>FIRST_PROCESS</Characteristic>
  <Characteristic>FIRST_PROCESS_EXE</Characteristic>
 </Characteristics>
 <UnknownCharacteristics>
  <Characteristic>SMALL_IMAGE_SIZE</Characteristic>
  <Characteristic>
```

```
          HAS_SHORTCUT_IN_START_MENU
         </Characteristic>
  <Characteristic>SIGNED_EXECUTABLE</Characteristic>
  <Characteristic>IS_SERVICE</Characteristic>
 </UnknownCharacteristics>
 <OutEdgeRelationships>
  <Relationship>
   <RelationshipName>0xedfeb9bc</RelationshipName>
   <RelationshipConnection>3</RelationshipConnection>
  </Relationship>
 </OutEdgeRelationships>
 <InEdgeRelationships/>
 <RegistryContext>
  <RegistryOperations>
   <RegistryOperationInfo>
    <TraceableId>2147483679</TraceableId>
    <OperationType>8194</OperationType>
    <Flags>2</Flags>
    <KeyName>\REGISTRY\MACHINE\SOFTWARE
        \CLASSES\PROTOCOLS\FILTER\TEXT/XML
       </KeyName>
    <ValueName/>
    <ValueData/>
    <ValueType>0</ValueType>
   </RegistryOperationInfo>
  </RegistryOperations>
 </RegistryContext>
</Node>
```

B Příprava systému

Výpis B.1: Aktualizace a příprava systému

```
# Aktualizace
root@ct5104:~$ apt-get update and apt upgrade

# Instalace užitečných balíčků
root@ct5104:~$ apt-get install git, curl, mc, ncdu

# Zkopírování adresářů, kam se budou ukládat větší data,
   na připojený disk: /home, /root, /tmp, /var
root@ct5104:~$ cp -r /root /mnt/storage/root
# obdobně pro další

# Smazání původních adresářů
root@ct5104:~$ rm -rf /home /root /tmp /var

# Vytvoření odkazů ve smazaném adresáři
root@ct5104:~$$ ln -s /mnt/storage/root /root
# obdobně pro další
```

Výpis B.2: Ukázka příkazů pro práci s Mono

```
# Instalace
root@ct5104:~$ apt-get install gnupg ca-certificates
root@ct5104:~$ apt-key adv --keyserver hkp://keyserver.
   ubuntu.com:80 --recv-keys 3
   FA7E0328081BFF6A14DA29AA6A19B38D3D831EF
root@ct5104:~$ echo "deb https://download.mono-project.
   com/repo/ubuntu stable-bionic main" | sudo tee /etc/
   apt/sources.list.d/mono-official-stable.list
root@ct5104:~$ apt-get update
root@ct5104:~$ apt-get install mono-devel
root@ct5104:~$ apt-get install mono-complete
root@ct5104:~$ apt-get install mono-dbg
root@ct5104:~$ apt-get install ca-certificates-mono
root@ct5104:~$ apt-get install referenceassemblies-pcl

# Spuštění nástroje pomocí Mono na pozadí s~logováním
root@ct5104:~/XmlLoader$ mono bin/Debug/XmlLoader.exe &>
   outputLog &
root@ct5104:~/XmlLoader$ disown
root@ct5104:~/XmlLoader$ tail -f outputLog
```

C Grafové databáze Neo4j

C.1 Instalace

Výpis C.1: Ukázka příkazů pro práci s Neo4j

```
# Instalace a spuštění
root@ct5104:~$ apt install java
root@ct5104:~$ wget -O - https://debian.neo4j.org/
   neotechnology.gpg.key | apt-key add -
root@ct5104:~$ echo "deb https://debian.neo4j.com stable
   latest" | sudo tee -a /etc/apt/sources.list.d/neo4j.
   list
root@ct5104:~$ apt-get update
root@ct5104:~$ apt-get install neo4j=1:4.0.4
root@ct5104:~$ service neo4j start
```

C.2 Konfigurace databázového serveru

Ve výpise C.2 je uvedena použitá konfigurace pro správný běh serveru. Většina hodnot jsou nastavené jako výchozí. Změna proběhla pouze u nastavení týkající se paměti podle výstupu nástroje „memrec“. Z důvodu úspory místa jsou z výpisu smazány všechna zakomentovaná nastavení.

Výpis C.2: Použitá konfigurace databázového systému Neo4j

```
dbms.directories.data=/var/lib/neo4j/data
dbms.directories.plugins=/var/lib/neo4j/plugins
dbms.directories.logs=/var/log/neo4j
dbms.directories.lib=/usr/share/neo4j/lib
dbms.directories.run=/var/run/neo4j
dbms.directories.import=/var/lib/neo4j/import
dbms.memory.heap.initial_size=31g
dbms.memory.heap.max_size=31g
dbms.memory.pagecache.size=364600m
dbms.tx_state.max_off_heap_memory=8g
dbms.jvm.additional=-XX:+ExitOnOutOfMemoryError
dbms.connector.bolt.enabled=true
dbms.connector.http.enabled=true
dbms.connector.https.enabled=false
dbms.tx_log.rotation.retention_policy=1 days
dbms.jvm.additional=-XX:+UseG1GC
dbms.jvm.additional=-XX:-OmitStackTraceInFastThrow
dbms.jvm.additional=-XX:+AlwaysPreTouch
dbms.jvm.additional=-XX:+UnlockExperimentalVMOptions
dbms.jvm.additional=-XX:+TrustFinalNonStaticFields
dbms.jvm.additional=-XX:+DisableExplicitGC
dbms.jvm.additional=-Djdk.nio.maxCachedBufferSize=262144
dbms.jvm.additional=-Dio.netty.tryReflectionSetAccessible
   =true
dbms.jvm.additional=-Djdk.tls.ephemeralDHKeySize=2048
dbms.jvm.additional=-Djdk.tls.
   rejectClientInitiatedRenegotiation=true
dbms.windows_service_name=neo4j
```

D Sestavané dotazy volané nad databázemi

D.1 Neo4j

Ve výpise D.1 jsou vypsány konkrétní dotazy pro testování měření. V prvním dotazu se nejprve nalezne uzel typu spustitelného souboru a označí se jako *e* `(e:Executable)`. Následně se specifikuje konkrétní cesta, jakou má databáze najít. Jde tedy o cestu, kde daný spustitelný soubor vyvolal nějaký proces, který vyvoval další proces, který následně smazal uzel *e*. Notací `*1..10` se specifikuje, že se hledá opakovaná hrana typu „SPAWN“. Nakonec se mají vrátit všechny zúčastněné uzly, aby byly mezi nimi vidět všechny hrany a celkové cesta.

V druhém dotaze je přidán dotaz `WHERE`, který umoňuje specifikovat libovolné podmínky na jakékoliv uzly z dotazu `MATCH`. Tyto podmínky jdou však také psát rovnou do specifikace uzlu do složených závorek.

Výpis D.1: Sestavené dotazy pro databázi Neo4j

```
// Dotaz č.1
MATCH
    (e:Executable)-[:INSTANCE_OF]->
    (p1:Process)-[:SPAWN*1..10]->(p2:Process)
    -[:DELETE]->(e)
RETURN e,p1,p2

// Dotaz č.2
MATCH
    (p:Process {NodeName: toUpper("C:\\Windows\\System32
       \\spoolsv.exe")})-[:SPAWN]->(x)
WHERE NOT "TRUSTED" IN x.Characteristics
return p,x
```

D.2 AgensGraph

Dotazovací dialekt do databáze AgensGraph je velmi podobný, jako do Neo4j. Zásadní rozdíl je v dotazování se na hrany, které je třeba provádět také ve složených závorkách na atribut hrany *relationshipType*.

Výpis D.2: Sestavené dotazy pro databázi AgensGraph

```
// Dotaz č.1
MATCH
    (p:Process {NodeName: 'C:\\WINDOWS\\SYSTEM32\\
       SPOOLSV.EXE'})
    -[:HAS_RELATIONSHIP {relationshipType: 'SPAWN'}]->(x)
WHERE NOT 'TRUSTED' IN x.charecteristics
RETURN p,x;

// Dotaz č.2
MATCH
    (e:Executable)
    -[:HAS_RELATIONSHIP {relationshipType:'INSTANCE_OF'
       }]->(p1:Process)
    -[:HAS_RELATIONSHIP*1..10 {relationshipType: 'SPAWN'
       }]->(p2:Process)
    -[:HAS_RELATIONSHIP {relationshipType: 'DELETE'}]
    ->(e)
RETURN e,p1,p2;
```

D.3 Dgraph

Ve výpise D.3 je vypsán Dotaz č.1 z kapitoly 5.3.1. Ve složených závorkách je nejprve libovolný název dotazu, po kterém v kulatých závorkách následuje dotaz `func:`. Zde se specifikuje, od kterých uzlu se má začít graf procházet. V tomhle konkrétním případě se vybírají ty uzly, jejichž *NodeName* se rovná zadanému řetězci. V dodatečném `@filter()` se dají specifikovat další podmínky.

Uvnitř složených závorek pro dotaz se specifikuje, které všechny atributy se mají s dotazem vrátit. Pokud dané atributy představují hrany na další uzly, vytváří se tak vlastní „poddotaz“, na který je možné dále aplikovat další filtry. Notace `@cascade` specifikuje, že by se v dotazu měly vrátit pouze ty uzly, které obsahují minimálně všechny požadované atributy. Druhý dotaz napsat nelze, jelikož zde není žádný mechanismus nespecifického počtu opakování hran.

Výpis D.3: Sestavené dotazy pro databázi Dgraph

```
// Dotaz č.1
{
    query1(func: eq(NodeName, "C:\\WINDOWS\\SYSTEM32\\
       SPOOLSV.EXE")) @filter(type(Process)) @cascade{
        NodeName
        spawn @filter(not eq(Characteristics, "TRUSTED"))
           {
            uid
            NodeName
            Characteristics
        }
    }
}
```

D.4 ArangoDB

Ve výpise D.4 jsou vypsány oba dotazy. Druhý dotaz jsem se pokusil vytvořil i přestože v ArangoDB nelze vytvářet „pattern matching“[37], ale dotaz se nedokázal zpracovat v reálném čase, proto jsem výsledky zahodil.

Výpis D.4: Sestavené dotazy pro databázi ArangoDB

```
// Dotaz č.1
FOR p IN mainNodes
FILTER p.nodeType == 2 AND p.nodeName == UPPER("C:\\
   Windows\\System32\\spoolsv.exe")
    FOR xv, xe, xp IN outbound p does_
    FILTER xe.relationshipType == "SPAWN" AND NOT
       position(xv.characteristics, "TRUSTED")
        RETURN xp

// Dotaz č.2
FOR s IN mainNodes
FILTER s.nodeType == 1
FOR v, e, p IN 3..6 OUTBOUND s does_
    PRUNE (v._key == s._key AND
        e.relationshipType == "DELETE") OR
```

```
        NOT p.edges[0].relationshipType == "INSTANCE_OF"
           OR
        NOT SHIFT(POP(p.edges[*].relationshipType)) ALL
           == "SPAWN" OR
        NOT SHIFT(POP(p.vertices[*].nodeType)) ALL == 2
    FILTER v._key == s._key AND
        e.relationshipType == "DELETE" AND
        p.edges[0].relationshipType == "INSTANCE_OF" AND
        SHIFT(POP(p.edges[*].relationshipType)) ALL == "
           SPAWN" AND
        SHIFT(POP(p.vertices[*].nodeType)) ALL == 2
    RETURN p
```

D.5 PostgreSQL

Ve výpise D.5 jsou vypsány oba dotazy v jazyce SQL.

Výpis D.5: Sestavené dotazy pro databázi PostgreSQL

```
// Dotaz č.1
SELECT
    V1."mainNodeID" as "fromID",
    V1."nodeType" as "fromNodeType",
    V1."nodeName" as "fromNodeName",
    V1."hash" as "fromHash",
    V1."fileSize" as "fromFileSize",
    V1."commandLine" as "fromCommandLine",
    V1."pid" as "fromPID",
    V1."creationTime" as "fromCreationTime",
    V1."content" as "fromContent",
    V1."characteristics" as "fromCharacteristics",
    R."type" as "relationshipType",
    V2."mainNodeID" as "toID",
    V2."nodeType" as "toNodeType",
    V2."nodeName" as "toNodeName",
    V2."hash" as "toHash",
    V2."fileSize" as "toFileSize",
    V2."commandLine" as "toCommandLine",
    V2."pid" as "toPID",
    V2."creationTime" as "toCreationTime",
```

```
    V2."content" as "toContent",
    V2."characteristics" as "toCharacteristics"
FROM "MainNodes" as V1
    INNER JOIN "Relationships" as R ON R."startID" = V1."
   mainNodeID"
    INNER JOIN "MainNodes" as V2 ON R."endID" = V2."
   mainNodeID"
WHERE V1."nodeName" = UPPER('C:\Windows\System32\spoolsv.
   exe') AND
R."type" = 'spawn' AND
'TRUSTED' != ALL(V2."characteristics");

// Dotaz č.2
SELECT
    V1."mainNodeID" as "ID",
    V1."nodeType" as "NodeType",
    V1."nodeName" as "NodeName",
    '-------------',
    R1."type" as "relationshipType",
    '------------>',
    V2."mainNodeID" as "ID",
    V2."nodeType" as "NodeType",
    V2."nodeName" as "NodeName",
    '-------------',
    R2."type" as "relationshipType",
    '------------>',
    V3."mainNodeID" as "ID",
    V3."nodeType" as "NodeType",
    V3."nodeName" as "NodeName",
    '-------------',
    R3."type" as "relationshipType",
    R3."endID" as "endID",
    '------------>',
    V4."mainNodeID" as "ID",
    V4."nodeType" as "NodeType",
    V4."nodeName" as "NodeName",
    '-------------',
    R4."type" as "relationshipType",
    R4."endID" as "endID",
```

```
    '------------>',
    V5."mainNodeID" as "ID",
    V5."nodeType" as "NodeType",
    V5."nodeName" as "NodeName",
    '-------------',
    R5."type" as "relationshipType",
    R5."endID" as "endID",
    '------------>',
    V6."mainNodeID" as "ID",
    V6."nodeType" as "NodeType",
    V6."nodeName" as "NodeName",
    '-------------',
    R6."type" as "relationshipType",
    R6."endID" as "endID",
    '------------>',
    V7."mainNodeID" as "ID",
    V7."nodeType" as "NodeType",
    V7."nodeName" as "NodeName",
    '-------------',
    R7."type" as "relationshipType",
    R7."endID" as "endID",
    '------------>',
    V8."mainNodeID" as "ID",
    V8."nodeType" as "NodeType",
    V8."nodeName" as "NodeName",
    '-------------',
    R8."type" as "relationshipType",
    R8."endID" as "endID"
FROM "MainNodes" as V1
    INNER JOIN "Relationships" as R1 ON R1."startID" = V1
   ."mainNodeID"
    INNER JOIN "MainNodes" as V2 ON R1."endID" = V2."
   mainNodeID"
    INNER JOIN "Relationships" as R2 ON R2."startID" = V2
   ."mainNodeID"
    INNER JOIN "MainNodes" as V3 ON R2."endID" = V3."
   mainNodeID"
    INNER JOIN "Relationships" as R3 ON R3."startID" = V3
   ."mainNodeID"
```

```
    LEFT JOIN "MainNodes" as V4 ON R3."endID" = V4."
   mainNodeID" AND V4."mainNodeID" <> V1."mainNodeID"
    LEFT JOIN "Relationships" as R4 ON R4."startID" = V4.
   "mainNodeID"
    LEFT JOIN "MainNodes" as V5 ON R4."endID" = V5."
   mainNodeID" AND V5."mainNodeID" <> V1."mainNodeID"
    LEFT JOIN "Relationships" as R5 ON R5."startID" = V5.
   "mainNodeID"
    LEFT JOIN "MainNodes" as V6 ON R5."endID" = V6."
   mainNodeID" AND V6."mainNodeID" <> V1."mainNodeID"
    LEFT JOIN "Relationships" as R6 ON R6."startID" = V6.
   "mainNodeID"
    LEFT JOIN "MainNodes" as V7 ON R6."endID" = V7."
   mainNodeID" AND V7."mainNodeID" <> V1."mainNodeID"
    LEFT JOIN "Relationships" as R7 ON R7."startID" = V7.
   "mainNodeID"
    LEFT JOIN "MainNodes" as V8 ON R7."endID" = V8."
   mainNodeID" AND V8."mainNodeID" <> V1."mainNodeID"
    LEFT JOIN "Relationships" as R8 ON R8."startID" = V8.
   "mainNodeID"
WHERE
    V1."nodeType" = 'executable' AND
    R1."type" = 'instance_of' AND
    V2."nodeType" = 'process' AND
    R2."type" = 'spawn' AND
    ((R3."type" = 'delete' AND R3."endID" = V1."
   mainNodeID") OR R3."type" = 'spawn') AND
    (R4."type" IS NULL OR (R4."type" = 'delete' AND R4."
   endID" = V1."mainNodeID") OR R4."type" = 'spawn') AND
    (R5."type" IS NULL OR (R5."type" = 'delete' AND R5."
   endID" = V1."mainNodeID") OR R5."type" = 'spawn') AND
    (R6."type" IS NULL OR (R6."type" = 'delete' AND R6."
   endID" = V1."mainNodeID") OR R6."type" = 'spawn') AND
    (R7."type" IS NULL OR (R7."type" = 'delete' AND R7."
   endID" = V1."mainNodeID") OR R7."type" = 'spawn') AND
    (R8."type" IS NULL OR (R8."type" = 'delete' AND R8."
   endID" = V1."mainNodeID"));
```

E Nástroj pro zpracování dat

E.1 Základní třídní struktura

- `Nebula` — základní adresář projektu
  - `ExtensionHelpers` — třídy pro práci s filtry
    - `ExtensionBase.cs` — abstraktní třída, podle které uživatel definuje filtr
    - `ExtensionCompiler.cs` — třída provádějící kompilaci filtrů
    - `ExtensionPipeline.cs` — třída, která pomáhá aplikovat filtry na grafy
    - `ExtensionPriority.cs` — atributní třída zavádějící prioritu na filtry
  - `Helpers` — pomocné třídy
    - `ConfigurationHelper.cs` — třída pro kontrolu a načtení konfigurace
    - `FileHelper.cs` — třída pracující s jednotlivými soubory
    - `NodeGraphHelper.cs` — pomocná třída pro vytváření podgrafů
    - `XmlFilesHelper.cs` — třída pro načtení informací o vstupních souborech
  - `Neo4j` — třídy pro import dat
    - `Neo4jDriver.cs` — třída pro vytvoření spojení a import dat do databáze
  - `MainNodes` — primární uzly
    - `ExecutableNode.cs`
    - `INode.cs`
    - `ProcessNode.cs`
    - `VirtualNode.cs`
  - `SecondaryNodes` — sekundární uzly, třídy
    - `Connection.cs`
    - `FilterInfo.cs`
    - `NodeState.cs`
    - `NamedObject.cs`
    - `RegistryOperationInfo.cs`
    - `Relationship.cs`
    - `RootNode.cs`
    - `TaggerResponse.cs`
  - `App.config` — soubor obsahující uživatelskou konfiguraci
  - `NodeGraph` — základní stavební třída
  - `packages.config` — automaticky vygenerovaný soubor
  - `Program.cs` — hlavní třída zajišťující správný postup a logování
  - `ThreadPoolLogic.cs` — třída obsahující logiku pro práci jednoho vlákna

E.2 Načítání dat ze vstupních XML

Výpis E.1: Vnořené načítání *Connection* pomocí předaného *XmlReader*

```
public static List<Connection> LoadConnections(XmlReader
   xmlConnections)
{
  List<Connection> connections = new List<Connection>();
  while (xmlConnections.Read())
  {
    if (xmlConnections.NodeType == XmlNodeType.Element &&
       xmlConnections.Name == "Connection")
    {
      Connection connection = new Connection();

      xmlConnections.ReadToFollowing("Domain");
      connection.Domain = xmlConnections.
         ReadElementContentAsString();

      xmlConnections.ReadToFollowing("URL");
      connection.Url = xmlConnections.
         ReadElementContentAsString();

      xmlConnections.ReadToFollowing("Address");
      connection.Address = xmlConnections.
         ReadElementContentAsString();

      xmlConnections.ReadToFollowing("Port");
      connection.Port = xmlConnections.
         ReadElementContentAsInt();

      xmlConnections.ReadToFollowing("Protocol");
      connection.Protocol = xmlConnections.
         ReadElementContentAsString();

      connections.Add(connection);
    }
  }
  xmlConnections.Dispose();
  return connections;
}
```

E.3 Import dat do databáze

```
if (!executable.NodeState.Imported)
{
    mainNodeID = tx.Run(
        "MATCH (r) WHERE id(r) = $rootNodeID " +
        "WITH $executable as executable, r " +
        "MERGE (n:Node {customID: executable.parameters.
           customID}) " +
            "SET n:Executable, n += executable.parameters " +
        "CREATE (n)-[:HAS_ROOT_NODE]->(r) " +
        "FOREACH (temp IN CASE WHEN executable.detection IS
           NULL THEN [] ELSE [1] END | " +
            "CREATE (d:Detection)<-[:HAS_DETECTION]-(n) SET d
                = executable.detection) " +
        "FOREACH (registryOperationInfo IN executable.
           registryContext | " +
            "CREATE (rgi:RegistryOperationInfo)<-[:
               HAS_REGISTRY_OPERATION_INFO]-(n) SET rgi =
               registryOperationInfo) " +
        "FOREACH (connection IN executable.connections | " +
            "CREATE (c:Connection)<-[:HAS_CONNECTION]-(n) SET
                c = connection) " +
        "RETURN id(n) AS id",
        new { rootNodeID = rootNodeID, executable =
           executable.CypherData }).First()["id"].As<long>();

    foreach (FilterInfo filterInfo in executable.
       AppliedFilters)
    {
        tx.Run(
            "MERGE (f:FilterInfo {fileHash: $fileHash, name:
               $name, value: $value}) " +
            "   ON CREATE SET f += {version: $version} " +
            "WITH f MATCH (e) WHERE id(e) = $mainNodeID
               CREATE (f)<-[:HAS_FILTER_INFO]-(e)",
            new
            {
                fileHash = filterInfo.FileHash,
                name = filterInfo.Name,
                value = filterInfo.Value,
```

```
                version = filterInfo.Version,
                mainNodeID = mainNodeID
            });
    }

    foreach (Relationship relationship in executable.
       OutgoingRelationships.SelectMany(x => x.Value))
    {
        string relationshipQuery =
            "MERGE (t:Node {customID: $customID}) WITH t " +
            "MATCH (e) WHERE id(e) = $mainNodeID " +
            $"CREATE (e)-[:{relationship.Name.ToUpper()}]->(t
               )";

        tx.Run(relationshipQuery, new { mainNodeID =
           mainNodeID, customID = relationship.LinkedCustomID
            });
    }

    executable.NodeState.Imported = true;
}
```

E.4 Optimalizace

Výpis E.2: Vytvoření profilace projektu

```
# Vytvoření .mlpd souboru
root@ct5104:~/graph-importer/bin/Release$ mono --profile=
   log:calls,alloc,output.mlpd,maxframes=8,calldepth=100
   Nebula.exe

# Zpracování .mlpd souboru
root@ct5104:~/graph-importer/bin/Release$ mprof-report
   output.mlpd > profileOutput.txt
```