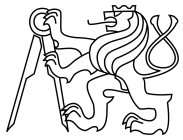
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Nástroj pro zátěžové testování grafových databází
Student:	Bc. Marek Červák
Vedoucí:	Ing. Adam Šenk
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Zátěžové testování zjišťuje, jak je systém schopný zpracovávat velké množství požadavků. Jedním z populárních nástrojů, které jsou k tomuto způsobu testování používány, je framework Gatling. Ten dokáže simulovat složité uživatelské chování i velký počet uživatelů přistupujících k systému zároveň. Je však vytvořen pro testování HTTP rozhraní. Cílem této práce je rozšířit framework Gatling tak, aby ho bylo možno použít pro zátěžové testování grafových databází.

- Seznamte se s nástroji pro zátěžové testování informačních systémů, zaměřte se na Gatling.
- Seznamte se s grafovými databázemi a dotazovacími jazyky.
- Navrhněte rozšíření nástroje Gatling o rozhraní, pomocí něhož bude možno definovat zátěžové testy nad grafovými databázemi.
- Navržené rozšíření implementujte a otestujte jeho funkčnost.
- Navrhněte ukázkové zátěžové testovací scénáře.
- Scénáře použijte k porovnání několika grafových databází.
- Vyhodnoťte výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 17. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Informačních technologií

Diplomová práce

Nástroj pro zátěžové testování grafových databází

Bc. Marek Červák

Vedoucí práce: Ing. Adam Šenk

9. ledna 2018

Poděkování

V první řadě bych rád poděkoval za spolupráci mému vedoucímu panu Šenkovi. Téma práce bylo jeho nápadem a během vypracování jsem se dostal k velmi zajímavé problematice. Dále bych rád poděkoval členům nejbližší rodiny za podporu během studia a psaní diplomové práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2018 .....

Odkaz na tuto práci

Červák, Marek. *Nástroj pro zátěžové testování grafových databází.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem této diplomové práce je rozšířit testovací nástroj Gatling, aby ho bylo možné použít pro zátěžové testování grafových databází. Práce se nejprve zabývá principy zátěžového testování a pak zkoumá blíže funkcionalitu nástroje Gatling. Dalším tématem je úvod do grafových databází a grafové dotazovací jazyky. Na základě těchto poznatků je navrženo a implementováno rozšíření nástroje Gatling. Následně je provedeno ukázkové zátěžové testovaní trojice grafových databází Neo4j, OrientDB a JanusGraph.

Klíčová slova Gatling, Gremlin, Zátěžové testování, Grafová databáze

Abstract

The aim of this master thesis is to extend the testing tool Gatling to use it for performance testing of graph databases. The work discusses principles of performance testing and functionality of Gatling. Afterwards, it deals with graph databases and graph query languages. Based on these findings, an extension of Gatling is designed and implemented. This extension is then showcased on a set of performance tests of graph databases Neo4J, OrientDB and JanusGraph.

Keywords Gatling, Gremlin, Performance Testing, Graph database

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Úvod

Cíl práce

Tato práce má za cíl navrhnout a implementovat rozšíření frameworku Gatling tak, aby ho bylo možné použít pro zátěžové testování grafových databází. Dalším cílem je navrhnout ukázkové testovací scénáře a ty použít pro porovnání několika grafových databází.

Úvod

Grafové databáze jsou relativně nový trendem ve světě databází. Existuje mnoho případů užití, kdy se grafová databáze zdá být přirozeně lepší volbou než standardní relační databáze. Hlavní doménou grafových databází je rychlost některých operací. Proto jsem vytvořil nástroj, který by pomohl právě rychlost a obecně výkon grafových databází testovat.

V současnosti existují studie porovnávající výkon databází pro určitý scénář, na základě toho si můžeme grafovou databázi vybrat. Podle mě ale chybí nástroj, který by mi dovolil definovat scénář vhodný pro daný případ užití a ten pak využít během vývojového cyklu. K vytvoření takového nástroje jsem použil existující testovací nástroj Gatling, který je určen primárně pro webové aplikace.

Členění práce

Práce je rozdělena do tří kapitol. Úvodní teoretická kapitola je zaměřená na principy zátěžového testování, poté se zabývá testovacími nástroji s důrazem na nástroj Gatling. Závěr kapitoly je věnován představení grafových databází. V druhé kapitole je obecně popsáno jakým způsobem vytvořit rozšíření Gatlingu a následně je popsána implementace pro jazyk Gremlin. V poslední kapitole je implementované rozšíření použito při ukázkovém testování grafo-

vých databází. Postupně je popsán návrh a implementace scénářů, příprava databází a testovacích dat. V závěru je provedena analýza výsledků měření.

Kapitola 1

Teoretická část

1.1 Zátěžové testování

Zátěžové testování *(Performance testing)* je obecný název pro testy zkoumající chování a výkon systému pod určitou zátěží. Zátěžové testy zkoumají reakce, stabilitu a dostupnost, rychlost a využití zdrojů. Snaží se identifikovat úzká hrdla systému, která omezují výkon.

Tento druh testování se řadí do skupiny nefunkčních testů. Cílem zátěžového testování tak není ověření správnosti systému, k tomu jsou určeny testy ze skupiny funkčního testování. [2]

V zátěžovém testování používáme pojmy:

- **Virtuální uživatel** Vygenerovaná simulace reálného uživatele, která interaguje se systémem.
- **Scénář** Posloupnost kroků, které virtuální uživatel vykonává. Reprezentace očekávaného chování uživatele.
- **Zátěžové testy** Simulace mnoha virtuálních uživatelů, kteří paralelně vykonávají scénář.

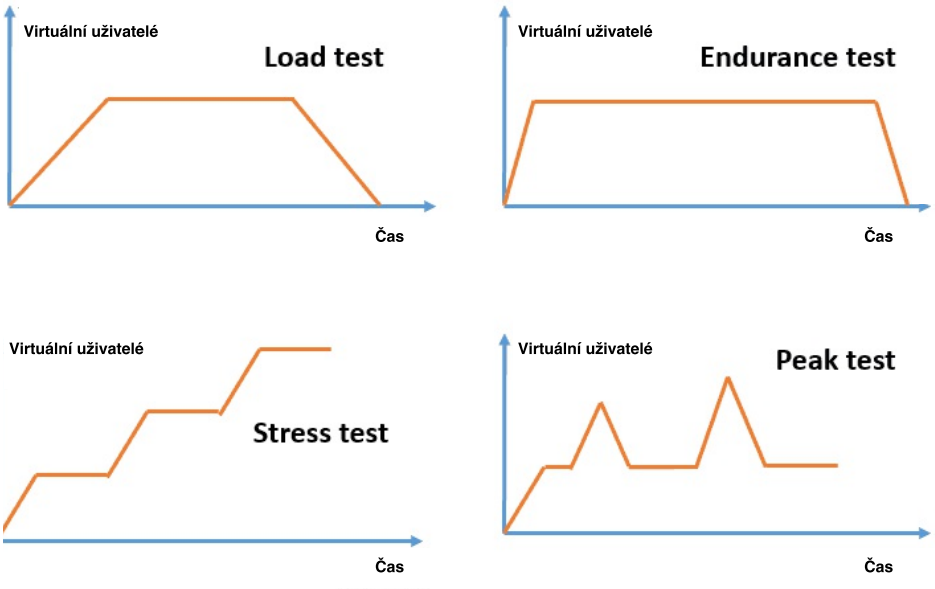
Zátěžové testy můžeme dělit na konkrétní typy testů podle jejich cílů:[3]

1.1.1 Výkonnostní test

Výkonnostní test je základním testem z této skupiny testů. S minimální zátěží ověřujeme, zdali systém odpovídá v čase akceptovatelném z pohledu uživatele.[4]

1.1.2 Load Test

Zátěžový test ověřuje výkonnost systému s očekávaným provozním množstvím uživatelů. Cílem tohoto testu je ověřit, že i v reálné zátěži se výkon blíží výsledkům výkonnostního testu.[4]



Obrázek 1.1: Rozložení virtuálních uživatelů v čase

1.1.3 Stress Test

Test hraniční zátěže ověřuje horní limity daného systému během extrémní zátěže. Tento test může být vykonán jako Load test s abnormálním počtem uživatelů. Pokud test zahrnuje náhlé zvýšení virtuálních uživatelů, nazýváme tento test Spike testem.

Tento druh testů je vhodné použít před důležitými událostmi spojenými s výrazným nárůstem provozu. U internetových aplikací například Black Friday, prodej lístků na koncert nebo před volbami.[4]

1.1.4 Soak / Endurance Test

Testování probíhá s normální zátěží, která se očekává v produkčním prostředí, avšak testuje se po delší dobu. Při tomto testu je možné odhalit chyby při práci se zdroji jako například neuvolňovaní paměti.[4]

1.1.5 Failover Test

Cílem tohoto testu je zjistit hranici, při které se systém zhroutí, a jak se u toho zachová. Případně jak proběhne následná obnova systému.[5]

1.1.6 Proces zátěžového testování

Metodika pro zátěžové testování se může lišit, cíl je ovšem vždy stejný - ověření výkonu daného systému. Proces zátěžového testování může vypadat ná-

sledovně: [5]

1. **Definice testovacího prostředí.** Testovací prostředí by mělo být co nejvíce podobné produkčnímu prostředí z pohledu softwaru, hardwaru a sítové kapacity.

2. **Definice akceptačních kritérií.** V tomto kroku se definují cíle a požadavky, které by systém měl splňovat. To zahrnuje dobu odezvy, množství alokovaných zdrojů.

3. **Návrh testů.** Při návrhu testů by měli pokryty scénáři klíčové případy užití s ohledem na míru užití skutečnými uživateli.

4. **Nastavení testovacího prostředí.** Příprava testovacího prostředí, testovacích nástrojů a potřebných zdrojů k exekuci.

5. **Implementace testů.** Implementace testů podle návrhů.

6. **Spuštění testů.** Spuštění a monitorování testů.

7. **Analýza výsledku** Nejprve jsou shromážděny výsledku testů a provedena jejich analýza s důrazem na splnění akceptačních kritérií. V případě neúspěchu by mě být problém opraven a testování navraceno k bodu 6.

1.1.7 Výhody zátěžových testů

Hlavní přínos zátěžových testů je ověření, zda aplikace zvládne svůj reálný nebo očekávaný provoz. Pokud je test znovupoužitelný, tak můžeme snadno ověřit chování při změně v aplikaci či změně konfigurace nebo hardwaru.

Jako u každého druhu testů zde platí pravidlo, že čím dříve tuto oblast začneme smysluplně testovat, tím dříve odhalíme případné nedostatky a snadněji a levněji je budeme moci odstranit. Na druhou stranu tvorba testů přidává práci na vývoji, proto je vhodné co nejvíce přepoužít existující testy zaměřené na funkčnost. Pro zátěžové testování je velmi zásadní, aby se prostředí blížilo co nejvíce prostředí produkčnímu.

1.1.8 Proč testovat výkon

V roce 2011 během prezidentské kampaně v USA proběhlo testování portálu pro dary, kde byly uživatelé rozděleni do 2 skupin s různou dobou odpovědi portálu. Tento druh testu nazýváme A/B testování. Snížení doby odezvy z 5 sekund na 2 znamenal zvýšení darů o 14 %, absolutní nárůst byl o 34.000.000 $. Pozorujeme tak, že doba odezvy může mít vážný dopad na rozhodování uživatele.[6]

1.2 Existující nástroje pro zátěžové testování

Při hledání existujících nástrojů podporujících zátěžové testování grafových databází jsem narazil pouze na hotové srovnávací testy, které byly pro konkrétní data a dotazy. Takové testy se složitě rozšiřují, protože dotazy pro jednotlivé databáze jsou psány v jejich nativním jazyce, tak aby databáze v testu dosahovaly nejlepších výsledků.

Cílem této práce je nabídnout univerzální nástroj pro testování grafových databází, proto jsem se zaměřil na existující nekomerční zátěžové nástroje a prozkoumal jejich rozšiřitelnost o nový protokol.

1.2.1 Apache JMeter™

Apache JMeter je aktuálně nejpopulárnější open source nástroj pro zátěžové testování. Původně byl navržen pro testování webových a FTP aplikací. JMeter je napsán v Jave a díky modulární architektuře je možné ho rozšiřovat o pluginy, avšak oficiální dokumentace pro vytvoření vlastního pluginu pochází z roku 2005.[7]

1.2.2 Gatling

Gatling je open source nástroj vyvíjený v jazyce Scala a postaven nad frameworkem Netty a nástrojem Akka. Přestože jsou i testy psané v jazyce Scala, jsou i bez znalosti tohoto jazyka velmi dobře čitelné a pochopitelné.

První verze nástroje Gatling byla vydána až v roce 2013, ale od té doby vzniklo hodně neoficiálních rozšíření, která doplňují jinak poměrně malý počet podporovaných protokolů.[8]

1.2.3 The Grinder

The Grinder je další z řady aplikací postavených nad JVM. Framework je napsán v jazyce Jython, speciální Java implementaci jazyka Python.[8]

1.2.4 TSUNG

Tsung je open source testovací nástroj původně navržen pro účely testování komunikačního programu Jabber (XMPP). Tsung je postavený nad Erlangem, tím pádem je spustitelný pouze na Unix systému. Erlang je funkcionální jazyk určen pro vývoj paralelních distribuovaných systémů.

Tsung je velmi výkonný nástroj, ale bohužel postrádá jakékoliv GUI a vše je tak nutno vykonávat v příkazové řádce.[8]

Funkce	The Grinder	Gatling	Tsung	JMeter
OS	Libovolný	Libovolný	Linux/Unix	Libovolný
GUI	Konzole	Pouze nahrávaní scénářů	Žádné	Nahrávání i spouštení
Nahrávání testů	TCP, HTTP	HTTP	HTTP, Postgres	HTTP
Jazyk testů	Python, Clojure	Scala	XML	XML
Jazyk rozšíření	Python, Clojure	Scala	Erlang	Java Jexl Beanshell Javascript
Výsledné reporty	Console	HTML	HTML	CSV, XML, Embedded Tables, Graphs, Plugins
Oficiálně podporované protokoly	HTTP SOAP JDBC POP3 SMTP LDAP JMS	HTTP JDBC JMS	HTTP XMPP WebDAV Postgres MySQL WebSocket AMQP MQTT LDAP	HTTP FTP JDBC SOAP LDAP TCP JMS SMTP POP3 IMAP
Omezení	Nutná znalost jazyka Python pro vytváření testů. Reporty jsou velmi stručné.	Neškáluje.	Omezení pouze na Linux/Unix. Jazyk Erlang.	Defaultní reporty nejsou uživatelsky přívětivé.

1.2.5 Shrnutí

Po zvážení všech pro a proti jsem se rozhodl pro Gatling jako nástroj, který se pokusím rozšířit o protokol určený k testování grafových databází. Pro Gatling hovořilo především větší množství dohledatelných článků a projektů implementujících vlastní protokol. Hlavní nevýhodou Gatlingu je podle mě nemožnost škálovat horizontálně, tedy běžet na více strojích paralelně. V kontextu grafových databází si myslím, že ale nebude problém vytvořit dostatečný počet dotazů.

1.3 Koncepty nástroje Gatling

V této kapitole představím základní koncepty, na kterých je Gatling postaven s odkazem na HTTP protokol. Velká část funkcionality ovšem s protokolem svázaná není, a tak je možné ji využívat v libovolném rozšíření. Všechny ukázky kódu jsou v jazyce Scala.

1.3.1 Architektura

Tradiční testovací nástroje jako JMeter stojí na konceptu Jeden uživatel = Jedno vlákno. To znamená, že během simulace je vygenerováno za každého uživatele jedno vlákno. Čekání během exekuce je prováděné pomocí Thread.sleep. Uživatel čeká na odpověď serveru a takové čekání nazýváme blokující, protože vlákno nedělá nic jiného.

Gatling využívá pokročilý výpočetní model postavený nad knihovnou Akka. Akka je distribuovaný framework implementující Aktorový model. Aktoři jsou entity komunikující s ostatními Aktory pomocí zpráv. Tento model je asynchronní – dovoluje tak simulovat mnoho virtuálních uživatelů pomocí malého počtu vláken s nízkou paměťovou náročností.

Aktor model byl poprvé představen v roce 1973 v publikaci pánů Hewitta, Bishopa a Steigera. Do povědomí se dostal díky implementaci v jazyce Erlang. Model je atraktivní především tím, že odpadá synchronizační část vývoje paralelní aplikace. Zprávy posílané mezi aktory jsou neměnné (immutable), tak aby například příjemce nemohl měnit obsah zprávy i jiným příjemcům. Aktoři by měli měnit pouze svůj vlastní stav.[9]

I přes rozdílné výpočetní modely nástrojů Gatling a Jmeter, nebylo prokázáno, že by jeden převyšoval výkonnostně druhého. To platí v případě, že jsou nasazeny na jednom stroji.[10]

Gatling využívá sílu asynchronní komunikace, proto by i klient, který komunikuje přes daný protokol, měl dotazy zpracovávat asynchronně. Pro HTTP protokol je využíván Apache Async HTTP Client.

1.3.2 Test

Testy pro Gatling jsou skripty napsané v jazyce Scala, konkrétně třídy obohacující třídu io.gatling.core.scenario.Simulation. API pro psaní testů je silně založeno na návrhovém vzoru Builder. Testy mají typovou kontrolu a dovolí nám pouze platné konstrukce.

Každý test můžeme rozdělit na 3 základní části:[11]

1. Konfigurace protokolu
2. Scénář
3. Simulace

```
scenario("Běžný uživatel")
  .exec(http("Otevři stránku vyhledávače")
    .get("https://www.google.cz"))
  .pause(3)
  .exec(http("Najdi restaurace na Praze 6")
    .get("https://www.google.cz/#q=restaurace+praha+6"))
  .pause(2)
```

Zdrojový kód 1.1: Jednoduchý scénář Bežný uživatel

To, že testy vlastně programujeme, namísto vytváření např. XML skriptu jako v případě JMeter, má ohromnou výhodu v znuvu použitelnosti. Jednotlivé komponenty napříč všemi testy mohou být sdíleny. Nevýhodou naopak je nutná kompilace při každé změně testu, aby se změna projevila. Pokud chceme test pouštět pouze s rozdílnými parametry, je možné využít JAVA_OPTS při spuštění testu a ty v testech získat standardně přes System.getProperty("$PARAMETR").

1.3.3 Protokol

Konfigurace protokolu je společná pro všechny kroky scénáře. V případě HTTP protokolu můžeme nastavit například základní URL a všechny dotazy následně definovat relativně vůči této adrese. Dalším nastavením může být základní autentizace, hlavička dotazů nebo Cache.

1.3.4 Scénář

Scénář používá silný a dobře čitelný doménový jazyk. Scénáře je možné vytvářet ručně nebo vygenerováním z reálného průchodu uživatele. Nahrávání probíhá přes HTTP proxy mezi prohlížečem a HTTP serverem.

Ze scénáře je, myslím, patrné i bez znalosti Gatlingu, že scénář se jmenuje "Běžný uživatel", provede 2 HTTP dotazy následované dvěma pauzami. Pauzy simulují dobu, kterou se vykonává dotaz, načítá prohlížeč a dobu, kterou uživatel stráví čtením, přemýšlením a navigací na stránce před dalším krokem.

Scénář je složen z akcí, které za sebe řetězíme s ohledem na pořadí, v jakém akce uživatele chceme vykonávat. Nejdůležitější metoda scénáře je metoda exec, která vykonává akce daného protokolu.

Ostatní metody jsou obecné a slouží k modelování scénáře z různých pohledů:

- **Čas:** pause, pace
- **Smyčky:** repeat, foreach
- **Podmínky:** doIf, doIfOrElse, doSwitch

```
val stdUser = scenario("Běžný uživatel") {...}
val advUser = scenario("Pokročilý uživatel") {...}

setUp(stdUser.inject(atOnceUsers(100)),
  advUser.inject(rampUsers(50) over (20 seconds))
)
```

Zdrojový kód 1.2: Nastavení simulace

Pokud rozšíříme Gatling o nový protokol, budeme moci využít všechny existující metody kromě akcí, jež jsou vázány na protokol a jsou volány metodou exec.[12]

1.3.5 Simulace

Simulace definují, jak velkou zátěž chceme vykonat a jak bude rozložená v čase. Jednotlivým scénářům jsou přiřazeni virtuální uživatelé a časový rámec toho, jak budou uživatelé vykonávat daný scénář viz 1.2. Pokud vše spojíme dohromady, vznikne jednoduchý spustitelný test, zdrojový kód lze nalézt v příloze A.1.

Model toho, kdy a kolik uživatelů interaguje s testovaným scénářem, musí být přizpůsoben případu užití. Model vhodný pro call centra se spíše konstantní zátěží, kde je omezený počet uživatelů počtem operátorů, se nehodí pro e-shop, který by měl ideálně obsloužit všechny zákazníky i ve špičce.[12]

1.3.6 Relace

Každý virtuální uživatel má k dispozici relaci (Session). Jedná se o úložiště, kam je možné vkládat či odkud je možné data získat. Interní reprezentace je Map [String, Any]. Jedná se o velmi důležitý koncept, protože nám dovoluje používat dynamická testovací data. Pokud používáme statická data, do hry silně vstupuje cache.[12]

Jak bylo řečeno v kapitole o architektuře, každý krok simulace je na pozadí reprezentován aktorem. Relace je zpráva, jež si jednotliví aktoři mezi sebou posílají, a protože jsou zprávy neměnné, tak jakákoliv operace měnící data relace vytváří novou instanci.

Vkládat data do relací lze třemi způsoby:

- Generátory
- Extrakcí dat z odpovědí a uložení, například HTTP saveAs
- Manuálně operacemi nad Instancí Session

Nejzajímavější variantou jsou generátory vstupů *Feeders*, které umožňují vkládat do testů data z externího zdroje dat. Podporuje soubory ve formáty

```
val feeder = Iterator.continually(Map("id" -> r.nextInt(100)))

.feed(feeder)
.doIf(session => session("id").as[Int] > 20) {
    exec(http("Zobraz předmět")
        .get("https://items.com/${id}/"))
}
```

Zdrojový kód 1.3: Generování dat a jejich použití v testu

```
http("My Request").get("myUrl").check(status.is(200))
```

Zdrojový kód 1.4: Kontrola odpovědi

CSV, JSON, datová úložiště s JDBC driverem a REDIS. Můžeme také jednoduše definovat náš vlastní zdroj jako Iterator[Map[String, T]].

Důležitou vlastností generátoru je, že je sdílen všemi virtuálními uživateli. Generátor musí nabídnout nejméně tolik hodnot, kolikrát je zavolán všemi uživateli dohromady. Při zavolání metody *feed* vrátí generátor pár klíč-hodnota, která je následně vložena jako hodnota do relace.

Získání dat z relace probíhá voláním instance.("klíč").as[T], kde T je očekávaný datový typ. Většina metod, jež Gatling poskytuje, podporuje vlastní výrazový jazyk, což je v podstatě String obohacený o jisté konstrukce. Například zápisem ${klíč} dovolí do Stringu dávat proměnné, které jsou poté při vyhodnocení nahrazeny hodnotou z relace, jak je patrné z ukázky 1.3. Během práce s relacemi je potřeba dát si pozor na existenci hodnoty pro daný klíč a také na chyby spojené s přetypováním, obě operace mohou skončit výjimkou.

1.3.7 Ověření

Rozhraní pro ověření (Checks) umožňuje zpracovávat odpověď, kterou vrátí akce zpět Gatlingu. U HTTP protokolu můžeme kontrolovat návratové kódy, hlavičky či těla odpovědi. Tyto hodnoty je možné ukládat do relací a využít je v další části testu. Jednotlivé podmínky se vždy vážou na akci, je možné jich definovat více a platí mezi nimi logicky operátor & při vyhodnocování.[12]

1.3.8 Assertions

Assertions API je navrženo pro ověření globálních statistik jako například doba odezvy nebo množství neúspěšných dotazů. Assert je definován rozsahem, druhem statistiky, metrikou a číselnou podmínkou. Vyhodnocování probíhá jako u klasických Unit testů – porovnáním očekávané a skutečné hodnoty dostaneme boolean hodnotu.[12]

```
setUp(scn).assertions(
  global.responseTime.max.lessThan(100),
  details("Zobraz předmět").successfulRequests.percent.greaterThan
      (95)
)
```

Zdrojový kód 1.5: Nastavení globálních metrik testu

1.3.9 Reporty

Gatling na závěr každého testu automaticky vytvoří report ve formě HTML souboru. Report obsahuje základní grafy s daty o čase a správnosti jednotlivých testů. Assertions se exportují do zvláštních souborů assertions.json a assertions.xml. Tyto soubory mají strukturu vhodnou pro integraci s nástroji typu Jenkins, pro které existují pluginy schopné zobrazovat reporty. Můžeme tak zapojit zátěžové testování v průběžné integraci a obohatit regresi i o výkonnost. Přílohy od obrázku A.1 až po soubor A.3 obsahují ukázky reportu vzorové http simulace.

1.3.10 Spouštění testů

Ke spouštění testů můžeme využít klasickou instalaci Gatlingu nebo Mavenu plugin. Vzhledem k tomu, že se jedná o aplikaci napsanou ve Scale, je prerekvizitou JDK8.[12]

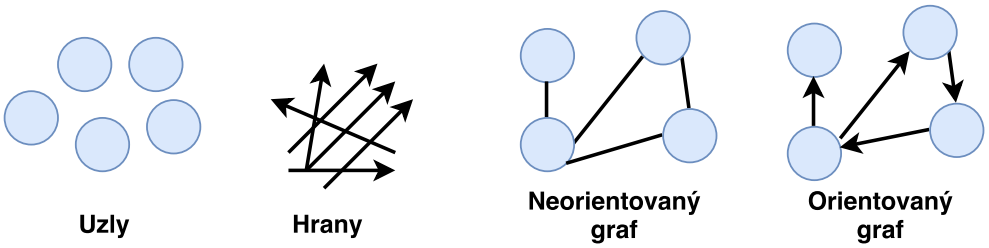
1.3.11 Distribuované testování

I přes pokročilý výpočetní model, který Gatling využívá, může být exekuce na jediném stroji nedostatečná. Problémem může být paměť či kapacita sítě. Gatling bohužel neumožnuje možnost být spuštěn jako distribuovaný na rozdíl od Jmeter. Tato možnost je pouze u komerčního projektu – cloud aplikace Gatling FrontLine.

Dokumentace nabízí návod, jak toho docílit manuálně v duchu *master-slave* architektury.[13]

1. Gatling nasadíme na několik strojů. Všechny instance musí obsahovat třídy pro simulace a všechny potřebné zdrojů (data, těla dotazů atd.)
2. Spustíme Gatling vzdáleně s přepínačem -nr (žádný report)
3. Shromáždíme simulation.log soubory
4. Přejmenujeme je unikátně tak, aby se předešlo kolizi
5. Vložíme je do složky ve výsledkové složce *master* instance Gatlingu
6. Vygenerujte reporty pomocí Gatlingu s parametry -ro $jméno-složky-simulace, Gatling použije všechny soubory s maskou .*log

Toto řešení tak v podstatě není distribuované v pravém smyslu, protože pokud použijeme stejnou simulaci ve všech N instancích, tak celkový počet uživatelů bude N*počet uživatelů definovaný v simulaci. Dalším problémem mohou být zdroje, pokud chceme unikátní vstupní data pro každého uživatele napříč všemi instancemi.



Obrázek 1.2: Základní struktura grafu

1.4 Grafové databáze

V úvodu této kapitoly si nejprve popíšeme základní pojmy, které se týkají grafových databází. Poté si popíšeme vlastnosti grafových databází.

1.4.1 Graf

Graf je uspořádaná dvojice G = (V, E), kde V je množina vrcholů (uzlů) a E je množina hran – množina vybraných dvouprvkových podmnožin množiny vrcholů 1.2. Z výše uvedené definice vyplývá, že hrana musí existovat vždy mezi existující dvojicí uzlů a pokud mažeme uzel, musíme smazat nejprve všechny hrany vedoucí z/do uzlu. [14] Rozlišujeme mezi orientovaným a neorientovaným grafem, podle toho, jestli jsou hrany orientované nebo ne.

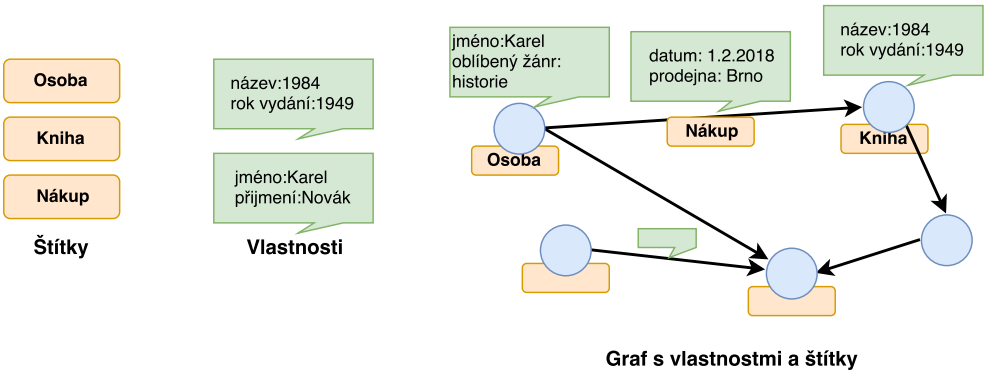
Graf může reprezentovat entity jako vrcholy a vztahy jako hrany mezi nimi. Pomocí grafu můžeme snadno a intuitivně modelovat mnoho oblastí reálného světa.

1.4.2 Graf s vlastnostmi a štítky

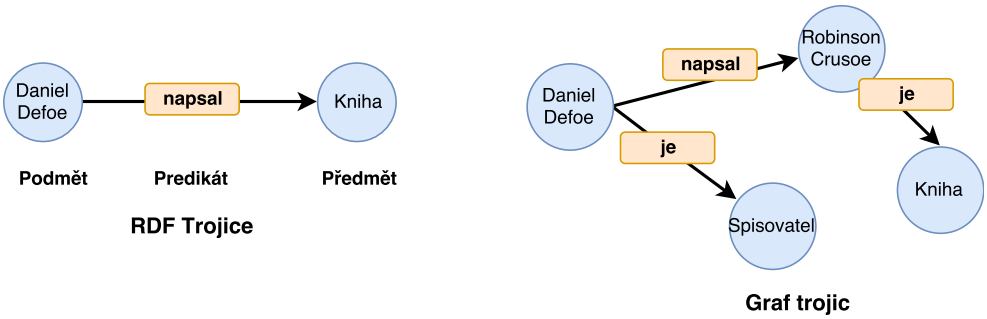
Graf s vlastnostmi a štítky (Labeled Property Graph) je nejpopulárnějším modelem grafu v grafových databázích. Získáme ho tak, že standardní graf obohatíme:

1. Uzly mohou mít vlastnosti (páry klíč-hodnota)
2. Uzly mohou mít jeden či více štítků
3. Hrany mají typ a směr
4. Hrany mohou mít také vlastnosti

Štítky můžeme chápat jako roli v dané doméně. Vlastnosti pak jako běžnou mapu, umožňující uložit do ní libovolný objekt.[15]



Obrázek 1.3: Příklad grafu s vlastnostmi a štítky



Obrázek 1.4: Příklad grafu trojic

1.4.3 RDF

RDF je založen na trojici podmět-predikát-předmět. Podmět je uzel, predikát hrana a předmět opět uzel. RDF model oproti grafu s vlastnostmi a štítky postrádá interní strukturu uzlů a hran 1.4.

RDF je normou pro sémantický web.[15]

1.4.4 Grafová databáze

Grafová databáze je označení pro online databázový systém užívající graf jako datový model, který umožnuje základní operace CRUD pro práci nad daty. Primárně jsou grafové databáze určeny pro práci s transakčními systémy (OLTP).

Nový typ dotazu oproti tradičním databázovým systémům je Traverzování neboli průchod grafem.

Grafové databáze můžeme dělit dle úložiště používaného pro uložení grafu a podle toho, jak dochází k vyhodnocování dotazů.

1.4.4.1 Úložiště

Některé databáze používají nativní grafovou strukturu pro uložení grafu, zatímco zbylé databáze mapují model na běžnou relační databázi, objektově-orientovanou databázi anebo jiné generické uložiště dat.

1.4.4.2 Vyhodnocení dotazu

Jestliže na sebe uzly v databázi fyzicky ukazují, tedy mají odkaz na sousední hrany nebo uzly, pak taková databáze podporuje index-free adjacency. Tato vlastnost přináší velmi zásadní rozdíl ve výkonu. Můžeme se setkat s označením "opravdová" grafová databáze, myslíme tím databázi, která používá nativní grafové uložiště a nativně zpracovává graf.

1.4.5 Výhody grafových databází

1.4.5.1 Výkon

Hlavní výhoda grafových databází je při práci se souvislými daty oproti relačním a NOSQL databázím. V porovnání s relačními databázemi, kde s rostoucí velikostí dat roste náročnost dotazů s častým výskytem operace JOIN, výkon grafových databázi zůstává víceméně konstantní. Dotazy jsou vyhodnoceny lokálně nad relevantní částí grafu, nikoliv nad grafem celým.[16]

1.4.5.2 Flexibilita

Grafová databáze nemá pevné schéma, tudíž je velmi lehce rozšiřitelná. Můžeme přidávat uzly, hrany nebo nový typ vazeb/štítků, aniž bychom narušili existující podgrafy a dotazy k nim vztažené. Díky těmto vlastnostem nemusíme dopředu detailně modelovat entity tak jako například v relační databázi a můžeme se snadno adaptovat novým požadavkům. Další výhodou je to, že odpadá nutnost migrace databáze kvůli změně schématu, tak jako tomu je u relačních databází. Na druhou stranu, to, že grafová databáze nemá schéma, přináší riziko v tom, že například podle štítku 'osoba' očekáváme, že vrchol reprezentuje danou entitu. To ovšem není zaručeno, a tak musíme být opatrní na aplikační vrstvě, kde typicky s entitami pracujeme.[16]

1.4.5.3 Případy užití grafových databází

V enterprise světě jsou hlavní případy užití grafových databází:

- **Doporučovací systémy** S tímto využitím se nejčastěji setkáme ve světě prodeje. Na základě informací o produktech, uživatelích a jejich chování v minulosti (předchozí nákupy, hodnocení, prohlížení) se systém snaží doporučit zákazníkovi relevantní zboží.[17]

Relační databáze jsou pro tento případ nevhodné, protože dotaz kombinuje různé entity, což vede na velké množství dotazů SELECT do různých tabulek, také množství možných JOIN konstruktů narůstá s počtem vazeb velmi rychle. Grafová databáze dokáže nalézt všechny relevantní entity/informace jedním dotazem, a relace ji výkonnostně tolik neomezují při zpracovaní.

- **Detekování podvodů** Detekce podvodů v reálném čase je velmi náročná a vyžaduje komplexní algoritmy pro vyvarování se falešným nálezům. Například praní špinavých peněz je možné odhalit nalezením cyklu v grafu.[18]
- **Geolokace** V telekomunikacích, logistice či cestování potřebujeme najít všechny relevantní objekty do určité vzdálenosti nebo hledáme nejkratší cesty mezi dvěma body.[17]
- **Sociální sítě** Sociální sítě jsou především o vazbách. Grafové databáze jsou vhodné na rychlý průchod sociální sítě pro nalezení doporučení přátel a známých, či nabídnutí relevantního obsahu.[18]

1.5 Grafové jazyky

V současné době se svět grafových databází ocitá ve stejné situaci jako byly relační databáze kdysi. S tím, jak se stávají grafové databáze populárnější a do jejich světa vstupují velcí hráči jako IBM či Oracle, roste tlak po jednotném standardu podobně jako je jazyk SQL.[19] Nyní existuje několik jazyků, které si popíšeme v této kapitole. Pro relevantní jazyky uvedu i příklad dotazu: Získej všechny uzly se štítkem Osoba, kteří si koupili knihu "Na západní frontě klid".

1.5.1 Cypher

Cypher je deklarativní dotazovací jazyk, původně vyvinutý pouze pro databázi Neo4j. V jistých ohledech jsou konstrukce podobné jako v jazyce SQL. Cypher v sobě reflektuje i grafickou reprezentaci grafu:
Uzly jsou v závorkách, to by mělo evokovat kruh.
Hrany jsou vyjádřeny pomocí šipek : "–>" "<–" [20]

```
MATCH (uzel1:Osoba)-->(uzel2:Kniha)
WHERE uzel2.nazev = {"Na zapadni fronte klid"}
RETURN uzel1
```

V roce 2015 vznikl openCypher projekt, který dává jazyk Cypher k dispozici ve snaze stát se standardem. Projekt obsahuje dokumentaci, specifikaci, referenční implementaci a sérii testů ověřující, že je daná verze jazyka Cypher podporována.[21]

1.5.2 SPARQL

SPARQL je deklarativní dotazovací jazyk RDF grafů podobný jako SQL. Od roku 2008 patří SPARQL mezi doporučení W3C, je jednou z technologií používaných pro sémantický web.[22]

```
PREFIX uri: <http:// ... >
SELECT ?osoba
WHERE
  {
    ?osoba uri:koupila ?kniha .
    ?kniha uri:nazev "Na zapadni fronte klid" .
  }
```

1.5.3 GraphQL

GraphQL na rozdíl od ostatních jazyků není určen pro práci s určitým typem databáze, ale slouží jako rozhraní pro komunikaci mezi klientem a serverem typicky skrze RESTful metody. Původní projekt z roku 2012 vyvinutý firmou Facebook je od roku 2015 open source.

Rozhraní sloužící ke komunikaci nazýváme GraphQL API schéma. Schéma je popis dat, která dává serverová část k dispozici. Schéma se skládá z objektů a typovaných atributů. Objekty mohou obsahovat jiné objekty, vznikají tak vazby, jež mají podobu grafu.

Klienti pomocí GraphQL dotazů mohou poptávat libovolná data definována schématem. GraphQL je deklarativní, vždy jasně říkáme, jaká data chceme dostat a jak mají vypadat. To dává jasnou výhodu oproti REST APi, kde musíme leckdy volat několik dotazů po sobě a výsledky skládat do sebe. Další výhodou je úspora dat v komunikaci, protože dostáváme jen ty objekty a atributy, které nás zajímají. To je kritické především při vývoji mobilních aplikací.

Na rozdíl od REST, kde má typicky každý zdroj svůj endpoint, využívá GraphQL jediný endpoint(/graphql).[23]

GraphQL je vhodné použít pro práci s daty, které mají strukturu podobnou grafu, či konkrétně stromu. GraphQL je zajímavou alternativou REST API a je možné, že ho v budoucnu nahradí.[22]

1.5.4 Gremlin

Gremlin je funkcionální dotazovací jazyk pro čtení a modifikaci dat v grafu. Gremlin umožnuje vyjádřit komplexní průchody grafem. Každý průchod grafem je složen ze sekvence kroků (steps). Kroky mohou být i vnořené. Každý krok vykonává atomickou operaci nad proudem dat. Kroky dělíme do tří základních skupin:[24]

- **Map-step** Objekty ve streamu dat se mění na jiné objekty.
- **Filter-step** Odstraňuje objekty ze streamu.
- **SideEffect-step** Výpočet statistik nad streamem dat.

Dotaz napsaný v jazyce Gremlin lze vyjádřit jak imperativně, tak deklarativně. Imperativní přístup říká prohledávači grafu v každém kroku, co vykonat. Deklarativní je podobný jako Cypher.

Gremlin je vyvíjený od roku 2009 jako součást projektu Apache TinkerPop™.[22]

```
1 g.V().hasLabel("Osoba").as('x').outE().hasLabel("nákup").inV().
    hasLabel("kniha").has("name", "Na západní frontě klid").select
    ('x')
```

Zdrojový kód 1.6: Dotaz v jazyce Gremlin

1.5.5 Shrnutí

OpenCypher a Gremlin jsou vážní kandidáti na standard grafových databází. Gremlin má momentálně větší podporu mezi poskytovateli, na druhou stranu

Neo4j má zdaleka největší podporu na trhu. I vzhledem k tomu, že velcí hráči jako Google, IBM a Microsoft se v roce 2017 přiklonili ke Gremlinu, rozhodl jsem se pro tento jazyk při implementaci.[18]

1.6 Apache TinkerPop™

Z kapitoly o grafových jazycích vyplynulo, že v implementaci použiju grafový jazyk Gremlin. V této kapitole si rozebereme podrobně celý framework Apache TinkerPop™, jehož je Gremlin součástí.

1.6.1 Nástroje frameworku TinkerPop

Cílem frameworku TinkerPop je obohatit datový systém o možnost chovat se k němu jako ke grafu nebo vybudování grafového systému od základu a mít okamžitě k dispozici dotazovací jazyk, server s infrastrukturou, metriky, reporting atd. To zajištují nástroje, jež jsou součástí frameworku.[25]

Pokud se datový systém stane podporovaným, může využít libovolný nástroj z níže uvedených.

1.6.1.1 Gremlin jazyk

Hlavní přidanou hodnotou frameworku je bez pochyby jazyk Gremlin. Tento jazyk byl navržen pro analýzu a manipulaci s grafem.

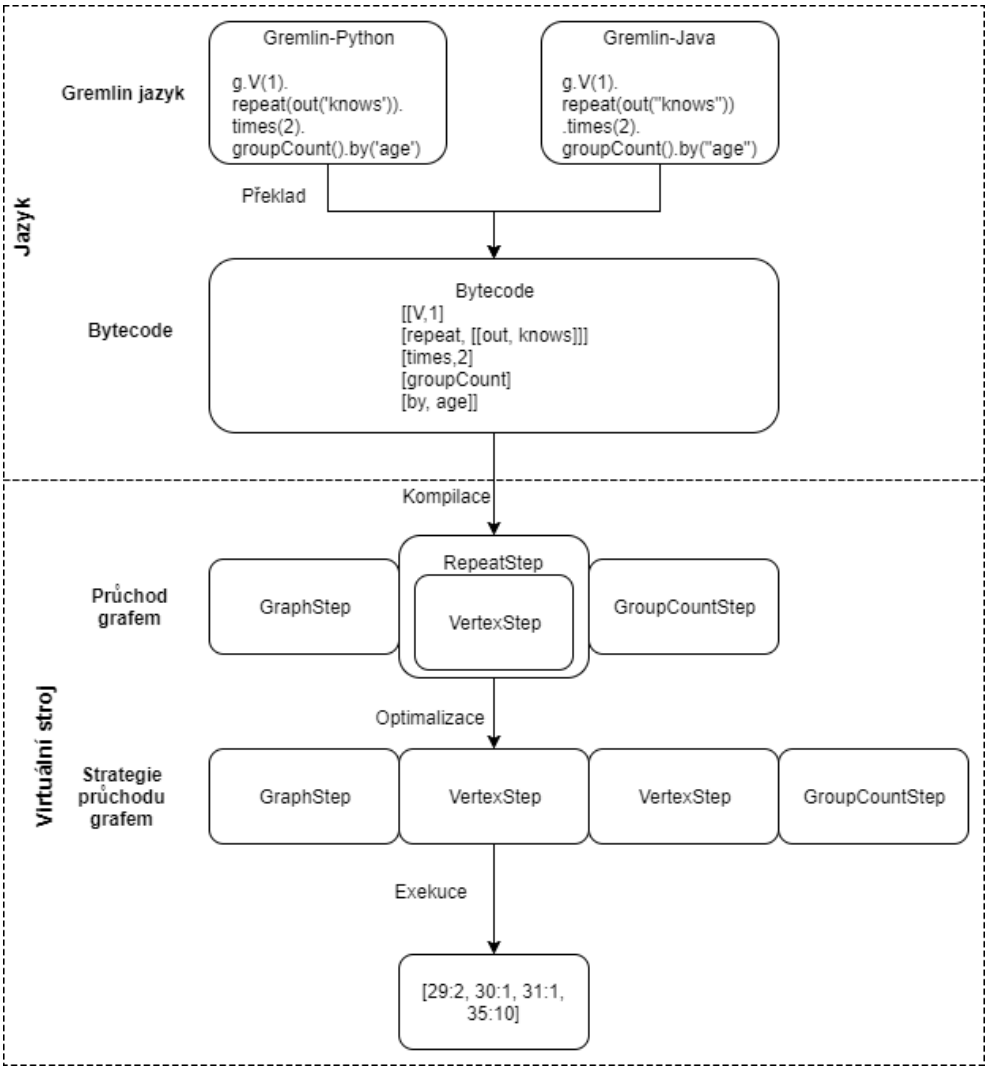
1.6.1.2 Gremlin Graph Traversal Machine

Gremlin je paralelou k jazyku Java, který je virtuální stroj a zároveň programovací jazyk. Výhodou JVM je, že Java program může být spuštěn na libovolném operačním systému podporující JVM bez nutnosti přepisování/-překompilování programu. To symbolizuje hlavní heslo Javy "napiš jednou, spusť kdekoliv". Program napsaný v jazyce Java je zkompilován pomocí javac do sekvence instrukcí JVM nazývaných Java bytecode.

Stejně tak Gremlin je grafový jazyk a grafovým virtuálním strojem. A právě paralela s Javou, ukazuje univerzálnost Gremlinu. Gremlin traversal machine obstarává knihovnu instrukcí (steps).

Koncept oddělení virtuálního stroje a jazyka má velmi praktické důsledky. JVM nevyžaduje, aby program byl napsaný v jazyce Java, protože vstupem je sada instrukcí. JVM tak podporuje libovolný programovací jazyk, pro který existuje kompilátor do Java bytecode. Díky tomu existují například jazyky Groovy, Scala a Clojure. Analogicky Gremlin traversal machine nevyžaduje, aby Gremlin traversal byl napsaný v Gremlin-Java8, a proto vzniklo velké množství jazykových mutací např. Gremlin-Groovy a Gremlin-Scala Když je dotaz kompilován, v závislosti na vyhodnocovacím systému na pozadí (OLTP graph database nebo OLAP grafový procesor), je uživatelův dotaz přepsán do série strategií průchodu. Při kompilaci je zohledněn cena přístupu k datům a vlastnosti daného systému.

Série kroků *...out().count().is(gt(10))...* je během kompilace přepsána do efektivnější formy *...outE().limit(11).count().is(gt(10))...*



Obrázek 1.5: Gremlin Traversal Machine

Gremlin je tedy navržen tak, aby uživatelé měli volnost při zápisu dotazu, a zároveň poskytovatelé grafových systémů mají možnost zvolit optimální strategii pro vyhodnocení dotazu na jejich TinkerPop-podporujícím datovém systému. Pokud přidáme krok explain() na závěr dotazu, dostaneme místo výsledku dotazu detailní rozbor, jak byl originální dotaz zkompilován a zoptimalizován na optimální sadu kroků.[26]

1.6.1.3 TinkerGraph

TinkerGraph je jednoduchá netransakční grafová databáze v paměti, vhodná pro průzkum grafů, které se vejdou do paměti, pro vytváření návodů a pro ladění dotazů bez nadbytečné režie.[25]

1.6.2 Gremlin Server

Gremlin Server je způsob, jak vyhodnocovat Gremlin dotazy vzdáleně nad jednou či více instancemi grafu. Komunikace se serverem probíhá pomocí klienta komunikujícího se serverem přes protokol HTTP nebo WebSocket. Server momentálně neumožnuje možnost komunikace přes oba protokoly zároveň.

Použitím serveru získáme možnost komunikovat s grafy i pomocí klientů napsaných v ne-JVM kompatibilních jazycích jako například Python. Další výhodou je zabezpečení přístupu ke grafu pomocí basic autentizace pro HTTP a SASL pro websockety.

Gremlin Server používá vlákna, jejich množství závisí na proměnné threadPoolWorker. Když dorazí dotaz na server, deserializuje se a zařadí se do fronty, která je odbavována pracovními vlákny. Jakmile nejsou k dispozici žádná vlákna pro práci, Server dále přijímá požadavky a fronta roste.

Vyhodnocování skriptů je blokující operace, pokud se tedy sejde větší množství "pomalých"dotazů, zablokuje se celá fronta. S tím, jak roste fronta, rostou její paměťové nároky, a to ještě více zpomaluje server. Této situaci se dá částečně bránit nastavením scriptEvaluationTimeout na straně serveru, takže "příliš pomalé"dotazy jsou přerušeny, pokud překročí limit.[27]

1.6.3 Gremlin Console

Pro práci v příkazové řádce existuje REPL konzole. Konzole je jednoduchý způsob, jak interaktivně komunikovat s instancí grafu jak lokální, tak vzdálenou. Konzole je vhodná jak pro interaktivní vytváření a ladění dotazů, tak pro administrátory systému pro získávání statistik či manuální průzkum dat.[25]

1.6.4 Podporované datové systémy

Datový systém se stane použitelným v Tinkerpop ekosystému, pokud implementuje následující rozhraní:

- **Graf (povinný)** Zásadní rozhraní definující sémantiku operací nad grafem, uzly, hranami a štítky. S touto minimální implementací se můžeme systému dotazovat jako grafové databáze pomocí Gremlin OLTP. Tuto minimální implementaci je vhodné obohatit o kolekci optimalizací strategií průchodu grafem specifických pro daný datový systém. Tyto optimalizace jsou použity při kompilaci dotazu a týkají se specifických oblastí, které bývají unikátní pro každý datový systém (např.: indexy, řazení)[25]

- **Grafový procesor (volitelný)** Všechny grafové procesory z kategorie OLAP musí implementovat jak rozhraní Graf, tak sadu rozhraní pro paralelní výpočet založený na předávání zpráv. Nicméně je možné implementovat pouze primární rozhraní grafu a podporovat OLAP integrací již existující implementace grafového procesoru např. SparkGraphComputer nebo GiraphGraphComputer.[25]

V současnosti existuje třináct podporovaných datových systémů, které se velmi liší v množství využitých nástrojů z Tinkerpop ekosystému.[25] KeyLines^{TM} ,Tom Sawyer Perspectives^{TM} a Linkurious^{TM} jsou nástroje pro zobrazování grafů, proto jsem je nezařadil do porovnání 1.1.

1.6.5 Podporované jazyky

Podporovaný jazyk je libovolný jazyk, pro který existuje kompilátor do Gremlin Traversal. První skupinou těchto jazyků jsou variace jazyka Gremlin. Gremlin-Java je považována za referenční implementaci Gremlinu. Existuje několik jazykových variací jako např. Gremlin-Python, které používají stejné jmenné konvence a styl pro zápis dotazů jako Gremlin-Java, ale využívají vlastní syntaxi a výhody daného jazyka. Druhou skupinou jazyků jsou zcela odlišné a na první pohled nesouvisející dotazovací jazyky SQL a SPARQL. Jazyky jsou natolik rozdílné, že mají svůj vlastní kompilátor. Kompilace nemusí být optimální, ale musí být sémanticky správná. Gremlin traversal machine se následně postará o optimalizace na úrovni kompilace a vyhodnocení dotazu. Oficiální jazykové variace Gremlinu jsou: Gremlin-Groovy, Gremlin-Java, Gremlin-Python a Gremlin Scala, Ogre(Closure), SQL a SPARQL.[25]

1.6.6 Komunikace s instancí grafu

Pro graf implementující TinkerPop3 API platí, že s ním lze komunikovat z Java aplikace dvěma způsoby:

1. Přímo: Vnořený graf běžící ve stejném virtuálním stroji jako naše aplikace.

2. Nepřímo: Graf běžící jako komponenta Gremlin Serveru.

Název	Vyvinuto v jazyce	Dotazovací jazyk	Licence	Databázový Model	Prostředí
Blazegraph™	Java	SPARQL Gremlin	GPLv2 komerční	graf RDF	Libovolné
Azure Cosmos DB™		Gremlin	komerční	graf document key/value wide column	Cloud
DataStax Enterprise Graph™	Java	Gremlin DseGraphFrames	komerční	graf	Cloud
GRAKN.AI™	Java	Gremlin Graql	GPLv3 komerční	graf relační	Libovolné
IBM Graph™	Java	Gremlin	komerční	graf	Cloud
JanusGraph	Java	Gremlin	Apache 2	graf	Libovolné
Neo4j™	Java, Scala	Cypher Gremlin	GPL 3 komerční	graf	Libovolné
OrientDB™	Java	Gremlin GraphQL SparkQL SQL	Apache 2 komerční	graf document key/value	Libovolné
Stardog™	Java	SPARQL Gremlin	Apache 2 komerční	graf RDF	Libovolné
Titan™	Java	Gremlin	Apache 2	graf	Libovolné

Tabulka 1.1: [Přehled podporovaných databází][1]

Postupně si představíme jednotlivé varianty a ukážeme jednoduché volání: Získej uzel s identifikátorem 1.

1.6.6.1 Vnořený graf

Vnořený graf je vhodná varianta, pokud je úložiště grafu lokální nebo pouze v paměti. Veškerá komunikace se odehrává v rámci stejného virtuálního stroje, tudíž nevzniká zbytečná režie. Na druhou stranu, pokud je úložiště vzdálené, vzniká velká režie při komunikaci a přenosu dat.

Nevýhodou tohoto řešení je, že jde o vždy o blokující volání a samozřejmě nefunguje pro aplikace neběžící v JVM.

Graf je reprezentován instancí implementující rozhraní org.apache.tinkerpop.-

gremlin.structure.Graph. Nad grafem pak můžeme volat metody přímo anebo využít skripty.

```
TinkerGraph graph = TinkerFactory.createModern();
GraphTraversal<Vertex, Vertex> result = graph.traversal().V(1)
```

```
TinkerGraph g = TinkerFactory.createModern();
String query = "g.V(1)";
GremlinGroovyScriptEngine engine = new GremlinGroovyScriptEngine()
    ;
Bindings bindings = engine.createBindings();
bindings.put("g", graph.traversal());
Object result = engine.eval(query, bindings);
```

Přímé volání je z pohledu vývoje elegantnější a bezpečnější vzhledem k statické typové kontrole jak při volání, tak při zpracování výsledku. Pokud bychom ovšem toto řešení chtěli použít, museli bychom obalit veškerá volání nad grafem, tak aby uživatel mohl nadefinovat testovací dotazy a my je mohli později volat.

Oproti tomu skripty vypadají složitěji, kompilují se až v době spuštění a návratovou hodnotou je Object, avšak podporují libovolný validní dotaz v jazyce Groovy. Jelikož není cílem této práce vytvořit nový jazyk, respektive překladač, skripty nabízejí elegantní možnost, jak vytvořit univerzální rozšíření.

Gremlin-Groovy je jazyk standardně podporován Gremlin konzolí. Konzole je interaktivní a bez nutnosti kompilace poskytuje prostředí pro rychlou exekuci dotazů nad grafem. V ideálním případě tak vytvoříme dotaz v konzoli, ověříme jeho funkčnost a poté ho jako String vložíme do testovacího scénáře.

1.6.6.2 Gremlin Server

Klient posílá dotazy ve formě skriptu podobně jako u přímého volání, avšak typicky má přiřazenou instanci grafu a prohledávač grafu na proměnné. Nespornou výhodou klienta je, že podporuje jak synchronní, tak asynchronní komunikaci se Serverem. Nevýhodou je opět chybějící typová kontrola.

```
Cluster cluster = Cluster.open();
Client client = cluster.connect();
ResultSet result = client.submit("g.V(1)");
CompletableFuture<ResultSet> asyncResult = client.submitAsync("g.V
    (1)");
```

V příkladu komunikace se serverem předpokládáme existenci proměnné g typu GraphTraversalSource, nad kterou se provede dotaz.

Je pochopitelné, že pokud chceme testovat výkonnost, neměl by Gatling sdílet výpočetní výkon stroje s testovanou grafovou databází. Tudíž varianta vnořeného grafu nedává smysl. Pro grafové databáze je typické, že běží na

jiném stroji než aplikační kód zvláště v kontextu cloud prostředí. Z tohoto důvodu jsem se rozhodl pro komunikaci s grafem využít Gremlin server, který je nejčastěji využíván v produkčním prostředí.

Kapitola 2

Realizace

V této kapitole nejprve předvedeme, jak obecně vytvořit libovolné rozšíření pro nástroj Gatling a poté si postupně ukážeme jak implementovat rozšíření pro grafové databáze s použitím jazyka Gremlin. Ukázky kódu jsou mixem Java a Scala kódu, vzhledem k jistým specifikům Scaly případné konstrukce vysvětlíme v kontextu Javy.

2.1 Jak vytvořit rozšíření Gatlingu

Gatling je open source projekt a podporuje rozšiřování a používání jeho kódu, avšak oficiální dokumentace, jak toho docílit, existovala naposledy k verzi 1.5.6. Interní API se často mění, a proto většina existujících rozšíření již není kompatibilní s aktuální verzi 2.3.0.[28]

Vytvoření vlastního Gatling protokolu spočívá ve vytvoření interního DSL, který je použitelný ve scénářích. DSL protokol dělíme na dvě části: konfigurace protokolu a akce, které nad ním můžeme vykonávat. Nepovinná je pak možnost kontrolovat výsledky akcí.

Prakticky pak pro vytvoření musíme implementovat konfigurační builder třídy, které umožnuji použití DSL v definici simulace. Dále pak třídy implementující logiku, které vzniknou zavoláním metody build() při překladu konfiguračních tříd.[29]

Následující návod je pro verzi Gatlingu 2.3.0.

2.1.1 Protocol

Protokol by měl definovat naše komunikační rozhraní a poskytovat nezbytné prostředky pro vykonávání akcí. Pro vytvoření protokolu musíme implementovat rozhraní Protocol, které nedefinuje žádnou metodu. Scala nemá *interface*, ale používá *trait* který bychom mohli přirovnat k rozhraním z Java 8 po přidání default metod.

```
1 trait Protocol
```

Zdrojový kód 2.1: Rozhraní ProtocolComponents

```
1 trait ProtocolComponents {
2   def onStart: Option[Session => Session]
3   def onExit: Option[Session => Unit]
4 }
```

Zdrojový kód 2.2: Rozhraní ProtocolComponents

```
1 trait ProtocolKey {
2   type Protocol
3   type Components
4   def protocolClass: Class[io.gatling.core.protocol.Protocol]
5
6   def defaultProtocolValue(configuration: GatlingConfiguration):
        Protocol
7   def newComponents(system: ActorSystem, coreComponents:
        CoreComponents): Protocol => Components
8 }
```

Zdrojový kód 2.3: Rozhraní ProtocolKey

2.1.2 ProtocolComponents

Jelikož protokol a akce nejsou typově svázané, musíme kolem naší implementace třídy Protocol udělat obálku jako implementaci rozhraní ProtocolComponents. Naše komponenta je při překladu DSL na akce snadno přístupná v protocolComponentsRegistry.

Rozhraní ProtocolComponents definuje dvě metody onStart a onExit, které jsou zavolány pro každého virtuálního uživatele před startem a po skončení scénáře. HTTP implementace nejprve nastaví cache a na konci ji uvolní. Obecně je to tak způsob, jak pracovat se zdroji na úrovni jednotlivých virtuálních uživatelů.

2.1.3 ProtocolKey

ProtocolKey je rozhraní, které slouží ke dvěma akcím. Prvně jako klíč k protokolu, každá třída ActionBuilder dostává kontext jako parametr. Kontext obsahuje registr protokolů, ze kterého pomocí klíče získáme Protocol.

Za druhé je ProtocolKey zodpovědný za inicializaci konkrétního protokolu voláním metody newComponents. Pokud registry neobsahuje instanci protokolu vázaného na klíč, provolá se metoda defaultProtocolValue, která vytvoří defaultní implementaci, pokud má smysl, jinak vyhodí výjimku.[29]

```
trait ActionBuilder {
  def build(ctx: ScenarioContext, next: Action): Action
}
```

Zdrojový kód 2.4: Rozhraní ActionBuilder

```
def logResponse(session: Session, requestName: String, timings:
    ResponseTimings,
    status: Status, responseCode: Option[String],
    message: Option[String], extraInfo: List[Any] = Nil): Unit
```

Zdrojový kód 2.5: Metoda pro logování výsledků.

2.1.4 ActionBuilder

Rozhraní ActionBuilder je určené k vytvoření instance akce v době spuštění simulace. ActionBuilder je parametrem funkce exec užívané při tvorbě scénáře. Jedinou metodou rozhraní je metoda build, která produkuje akci. Prvním parametrem funkce je kontext scénáře ScenarioContext obsahující všechny statické informace, které můžeme využít k vytvoření Akce. Mimo jiné v sobě uchovává protocolComponentsRegistry, z kterého lze vytáhnout protokol pomocí ProtocolKey a předat ho akci, tak aby s ním mohla pracovat. Druhým parametrem je reference na následující akci Action. Koncept řetězení akcí je implementován tak, že každá akce je zodpovědná za spuštění následující akce.[28]

2.1.5 Stats Engine

StatsEngine je třída potřebná k zaznamenání doby odezvy jednotlivých akcí. Její instance by měla být vždy předána z ActionBuilder do Akce. Defaultní implementace této třídy je DataWritersStatsEngine, která data zaznamenává do souboru, který je poté použit pro tvorbu reportu.[30]

Nejdůležitější poskytovanou metodou je logResponse. Signatura této metody je v ukázce 2.5. Jméno dotazu je použito pro shromažďování výsledku v reportu, timings je doba vykonání dotazu a status je konečný výsledek. Gatling rozlišuje pouze na statusy OK a KO.

2.1.6 Action

Rozhraní Action reprezentuje unikátní instanci uživatele, kde se v těle metody execute skutečně vykoná daný krok simulace. Jediným argumentem metody execute je relace Session, reprezentující stav simulace daného uživatele. Při implementování této metody musíme zajistit, že v každém případě skončí, to je nebezpečné především pro asynchronní komunikaci, kde není použito blokující

```
trait Action extends StrictLogging {
  def name: String
  def execute(session: Session): Unit
}
```

Zdrojový kód 2.6: Rozhraní Action

```
trait Check[R] {
  def check(response: R, session: Session)(implicit cache: mutable
      .Map[Any, Any]): Validation[CheckResult]
}
```

Zdrojový kód 2.7: Rozhraní Check

volání. Dále pak musíme na konci zavolat následující akci opět pomocí metody execute na referenci akce získané při vytváření akce v ActionBuilder.

2.1.7 Check

Kontrola odpovědi je nepovinná nadstavba při tvorbě rozšíření. Check je generické rozhraní, které pro daný datový typ R definuje metodu check, která vrací objekt typu Validation[R], což je obálka původní odpovědi implementována jako Success nebo Failure.

```
class GremlinSimulation  extends Simulation {
  val gremlinProtocol = new GremlinProtocol("src/main/resources/
      remote.yaml")
 def scn = scenario("test").repeat(1){
   exec(gremlin("získej vrchol").query("g.V(1)"))
 }
 setUp(
   scn.inject(atOnceUsers(1))
 ).protocols(gremlinProtocol)
}
```

Zdrojový kód 2.8: Návrh scénáře využívající protokol Gremlin

2.2 Implementace

V této kapitole přiblížím implementaci rozšíření pomocí postupů z předchozí kapitoly. Objevují se zde i vlastní pomocné třídy, které nejsou nutné pro rozšíření rozhraní Gatlingu.

2.2.1 Prvotní návrh

V kapitole věnovanému Gatlingu jsme si představili funkcionalitu na HTTP protokolu a jeho DSL. Takové testy jsou velmi dobře čitelné a udržovatelné, a to samé by mělo splnit rozšíření pro jazyk Gremlin. Z kapitol o komunikaci s grafem skrz Tinkerpop API vyplynulo, jak by měl vypadat protokol. Jednoduchý scénář by tak měl vypadat jako v ukázce 2.8.

Protokol inicializujeme s klientem, který se postará o komunikaci se serverem a DSL by mělo být co nejjednodušší a univerzální zároveň, tedy podporovat základní sadu předdefinovaných dotazů, tak libovolné uživatelské dotazy.

2.2.2 Protokol

Protokol přijímá v konstruktoru instanci typu GremlinServerClient, která se stará o komunikaci se serverem. Protokol mimo to poskytuje ještě dva konstruktory pro vytvoření klienta ze souboru, nebo vytvoření klienta s defaultní konfigurací. Protokol dává akcím k dispozici jednu sdílenou instanci klienta. Při testování klienta jsem byl schopen dosáhnout propustnosti cca 500 dotazů za vteřinu, potom docházelo k timeoutu už při snaze poslat dotaz.

2.2.3 Klient

Ke komunikaci je použita knihovní implementace Gremlin klienta. Pro připojení k serveru potřebujeme znát jeho adresu a port. Dále pak můžeme konfigurovat, jakým způsobem dochází k serializaci a deserializaci během komunikace.[31]

Pokud nespecifikujeme cestu ke konfiguračnímu souboru, je použita defaultní konfigurace 2.10. Třída použitá pro serializaci musí být definována

```
case class GremlinProtocol (serverClient : GremlinServerClient)
    extends Protocol {
      def this(path: String) {
      this(GremlinServerClient.createClient(path))
  }
  def this() {
    this(GremlinServerClient.createDefaultClient())
  }
}
```

Zdrojový kód 2.9: Protokol

```
hosts: [localhost]
port: 8182
serializer:{className:org.apache.tinkerpop.gremlin.driver
.ser.GryoMessageSerializerV1d0, config: { serializeResultToString:
    true }}
```

Zdrojový kód 2.10: Defaultní konfigurace

také na straně serveru, pokud jej chceme využít z klienta. Volba serializeResultToString je velmi důležitá pro rychlost exekuce dotazu, protože typicky nechceme vracet celý objekty uzlu či hrany, ale pouze určité hodnoty se kterými dále pracujeme, proto se doporučuje pracovat s serializeResultToString=true. V opačném případě může nastat úskalí při zpracování odpovědi, protože musíme mít na classpath knihovny dané grafové databáze.[27]

2.2.4 Protocol Key

Implementace klíče v podstatě kopíruje implementaci oficiálního protokolu JmsProtocol. Defaultní implementace protokolu jsem neimplementoval, i když by to mohl být protokol s klientem používající defaultní konfiguraci. Působí to ovšem jako anti-pattern, protože bychom měli pracovat v simulacích jen s protokoly, které jsme explicitně nadefinovali. Kód je v ukázce 2.11.

Při implementaci mě nenapadlo vhodné využití metod onStart a onExit. V implementaci nepoužívám žádné zdroje vázané na virtuálního uživatele, všichni uživatele sdílí jednoho klienta v současné implementaci.

```
case class GremlinComponents(gremlinProtocol: GremlinProtocol)
    extends ProtocolComponents {
  override def onStart: Option[(Session) => Session] = None
  override def onExit: Option[(Session) => Unit] = None
}
```

2.2.5 Predef

Predef je object, který musíme importovat v simulaci pro použití Gremlin

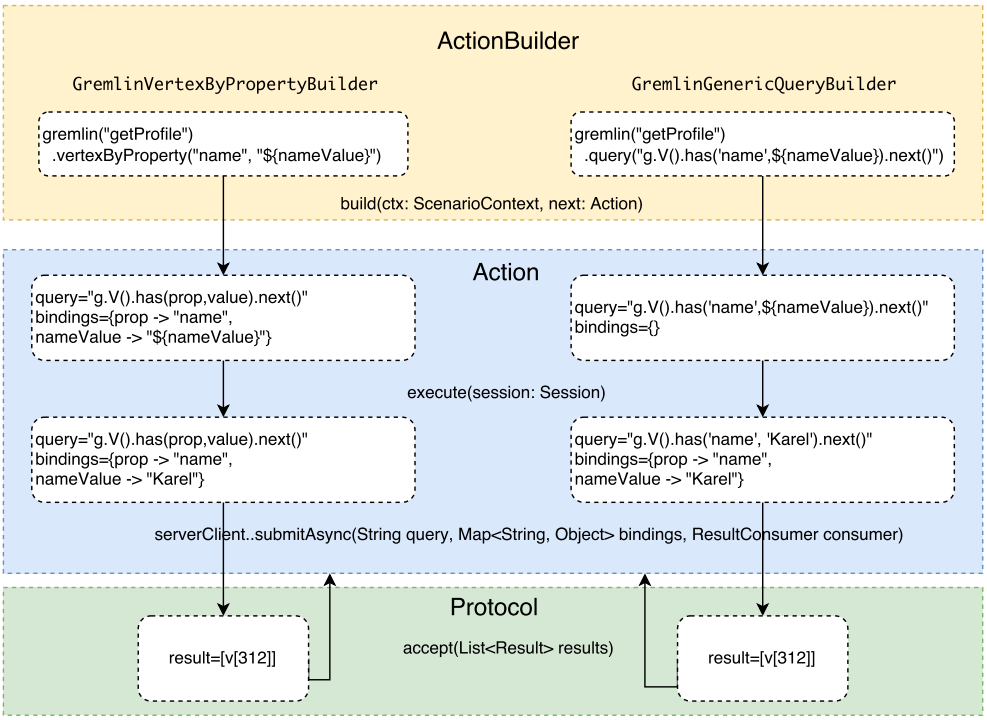
```
object GremlinProtocol {
  val GremlinProtocolKey = new ProtocolKey {
  type Protocol = GremlinProtocol
  type Components = GremlinComponents

  def protocolClass: Class[io.gatling.core.protocol.Protocol] =
      classOf[GremlinProtocol].asInstanceOf[Class[io.gatling.
    core.protocol.Protocol]]
  def defaultProtocolValue(configuration: GatlingConfiguration):
    GremlinProtocol = throw new IllegalStateException("Can't
    provide a default value for GremlinProtocol")

  def newComponents(system: ActorSystem, coreComponents:
    CoreComponents): GremlinProtocol => GremlinComponents {
     gremlinProtocol: GremlinProtocol => GremlinComponents(
        gremlinProtocol)
    }
  }
}
```

Zdrojový kód 2.11: Tento kód je nezbytný, aby byl vytvořený protokol dostupný při vytváření akce.

```
object GremlinPredef {
  def gremlin(requestName: Expression[String]) = Gremlin(
    requestName)
}
case class Gremlin(requestName: Expression[String]) {
  def query(query: Expression[String]) =
    GremlinGenericQueryBuilder(requestName, query)
  def vertex(id: Expression[String]) = GremlinVertexBuilder(
    requestName, id)
  def vertexByProperty(property: Expression[String], value:
    Expression[String]) = {..}
  ..
}
```

Zdrojový kód 2.12: Tvorba DSL.

protokolu. Scala nemá klíčové slovo *static*, místo toho využívá koncept *companion object*, což je singleton ke stejnojmenné třídě. Všechny statické metody a proměnné patří sem. Predef obsahuje jedinou metodu Gremlin, která má parametr jméno dotazu a vrací Třídu Gremlin. Třída Gremlin dává na výběr, jaký dotaz chceme použít. Tímto způsobem dosáhneme vytvoření zřetězeného DSL podobně jako u HTTP protokolu. Zřetězení dosáhneme jednoduše jako v ukázce 2.12.



Obrázek 2.1: Reprezentace dotazu v simulaci.

2.2.6 Action Builder

Pro každý druh dotazu, který jsme definovali v Predef objektu existuje třída rozšiřující abstraktní třídu GremlinBuilder. V implementaci rozlišuji mezi dvěma druhy dotazů tzv. obecný reprezentován třídou GremlinGenericQueryBuilder a sadou dotazů, které existují jako jaká si šablona. Jak funguje překlad uživatelem zadaného dotazu, až po jeho vykonání ilustruje obrázek 2.1.

2.2.7 Action

Metoda execute je naprosto zásadní při vykonávání simulace. Musíme zajistit, že vždy dojde k zavolání další akce. Proto jsem v implementaci metodou execute pouze obalil metodu doAction a odchytávám výjimky, jež by mohly nastat.

Asynchronní přístup s sebou přináší i mnohé problémy. Při bližším prozkoumání kódu v akci je patrné, že pokud klient nezavolá jednu z metod accept, zůstane uživatel zaseknutý v tomto kroku a simulace nikdy neskončí.

Implementace metody a její popis je v ukázce 2.13.

```
override def execute(session: Session): Unit = {
  val startTime: Long = now() //1 Začátek měření času
  val tried = Try(doAction(session, startTime))
  if (tried.isFailure) {
    log(startTime, now(), tried, requestName, session, statsEngine
        )
    next ! session.markAsFailed}}

  def doAction(session: Session, startTime: Long): Unit = {
  val resolvedQuery = resolveQuery(session)  //2 Překlad dotazu a
      vytvoření bindings.
  protocol.serverClient.submitAsync(resolvedQuery.query,
      resolvedQuery.bindings, //3 Asynchronní odeslání požadavku
      na server
  new ResultConsumer {
  override def accept(result: util.List[Result]): Unit = { //4 Př
      ijetí odpovědi
  log(start, now(), scala.util.Success(""), requestName, session,
      statsEngine) //6 Zalogování výsledku
  next ! newSession //7 Volání následující akce v~simulaci.}

  override def acceptError(t: Throwable): Unit = { //5 Přijetí
      chyby během exekuce
    log(startTime, now(), Try(t), requestName, session,
        statsEngine) //6 Zalogování výsledku
  next ! session.markAsFailed} //7 Volání následující akce v~
      simulaci.
})
}
```

Zdrojový kód 2.13: Exekuce akce

2.2.8 GremlinServerClient

GremlinServerClient je obálka nad oficiálním klientem. Gremlin klient je sice asynchronní, ale v mezních případech může selhat. Akce při volání klienta musí předat implementaci rozhraní ResultConsumer, což je rozšíření standardního rozhraní Consumer o metodu pro zpracování chyb. Pro akce je naprosto kritické, aby klient zavolal jednu z metod accept jako callback a akce mohla zpracovat výsledek a skončit.

```
public interface ResultConsumer extends Consumer<List<Result>> {
    void acceptError(Throwable t);
}
```

Jedním z případů, kdy může nastat problém, je přetížení serveru, kdy požadavky začnou „umírat“ na serveru a klientovi nepřijde žádný výsledek. Tento problém asynchronního volání jsem vyřešil s maximální dobou čekání. Metoda acceptEither přijímá výsledek z dvou volání a čeká na první z nich. Pokud tak klient neobdrží včas odpověď, je zpracována prázdná odpověď vytvořená me-

```
1     public void submitAsync(String gremlinQuery, Map<String,
          Object> variables,
2     ResultConsumer consumer) {
3         CompletableFuture<ResultSet> response = client.submitAsync
              (gremlinQuery, variables);
4         response.acceptEither(timeoutAfter(10, TimeUnit.SECONDS),
5                 results -> results.all()
6                         .acceptEither(timeoutAfter(2, TimeUnit.
                              SECONDS), consumer)
7                         .exceptionally(getThrowableVoidFunction(
                              consumer)))
8                 .exceptionally(getThrowableVoidFunction(consumer))
                      ;
9     }
10
11    private Function<Throwable, Void> getThrowableVoidFunction(
          ResultConsumer consumer) {
12        return throwable -> {
13            consumer.acceptError(throwable);
14            return null;
15        };
16    }
```

Zdrojový kód 2.14: Asynchronní volání klienta.

todou timeoutAfter, akce simulace skončí statusem KO a simulace pokračuje dál. Metoda timeoutAfter využívá knihovní třídu ScheduledExecutorService, která umožnuje exekuci instancí třídy Runnable v zadaný čas.

Dalším případem je, když provedeme dotaz, který nemá žádný výsledek, klient pak vyhodí výjimku. Pro řešení výjimek při zpracovávání CompletableFuture je vhodná metoda exceptionally, která při chybě zavolá přiřazenou Void funkci.

2.2.9 Check

Do této chvíle pro nás byla odpověď klienta vrácena metodou accept(List[Result]) správná. Přidání kontroly znamená další kroky ve zpracování v metodě Action. Odpověď ze serveru je vždy typu List[Result] a zároveň víme, že Check musí mít návratovou hodnotu typu boolean. Proto by Gremlin akce měly podporovat rozhraní ResultCheck = Check[List[Result]].

V tomto případě jsem se rozhodl pro nejjednodušší a uživatelsky nejpřívětivější řešení, tedy že uživatel bude moci nadefinovat vlastní vyhodnocovací funkci List[Result] => Boolean a tu vložit do akce. K tomu slouží třída SimpleResultCheck 2.15. Nyní musíme rozšířit GremlinActionBuilder o rozhraní GremlinCheckResultActionBuilder, odteď je možné použití jako v ukázce 2.17. K vyhodnocení dochází v akci pomocí knihovní metody Check:check[R].

```
case class SimpleResultCheck(func: List[Result] => Boolean)
    extends ResultCheck {
  override def check(response: List[Result], session: Session)(
      implicit cache: mutable.Map[Any, Any]): Validation[
      CheckResult] = {
    if (func(response)) CheckResult.NoopCheckResultSuccess
    else Failure("Gremlin result check failed")
}
```

Zdrojový kód 2.15: Jednoduchá implementaci rozhraní Check

```
trait GremlinCheckResultActionBuilder extends ActionBuilder {
    protected val checks: ArrayBuffer[ResultCheck] = ArrayBuffer.
        empty
    def check(check: ResultCheck): GremlinCheckResultActionBuilder
        = {
      checks += check
      this}
```

Zdrojový kód 2.16: Přidání podpory kontroly výsledků

```
.exec(gremlin("Najdi uzel podle jména Petr")
    .query("g.V().has('name','Petr').next()")
    .check(simpleCheck(res => res.size == 1)) // očekáváme právě
        jeden výsledek
```

Zdrojový kód 2.17: Kontrola výsledku

2.2.10 Ukládání výsledku

Podobně jako při kontrole výsledku, při ukládání výsledku je argumentem funkce, která je poté aplikováná při vykonání akce. Druhým parametrem metody je klíč, pod kterým se výsledek uloží do relace. Pro ukládání výsledku už nám stačí pouze rozšířit rozhraní GremlinCheckResultActionBuilder o další metodu, zpropagovat funkci do akce a při zpracovávání výsledku ji zavolat. Použití je vidět na ukázce 2.18

```
def extractResultAndSaveAs(extractionMethod: List[Result] =>
    Object, key: String)

.exec(gremlin("Najdi počet přátel")
    .query("g.V(1).out().count()")
    .extractResultAndSaveAs(result => result.head.getString, "
        numberOfFriends"))
.exec{session =>
    println(session("numberOfFriends").as[String])
    session}
```

Zdrojový kód 2.18: Uložení výsledku do relace

2.3 Instalace a spuštění

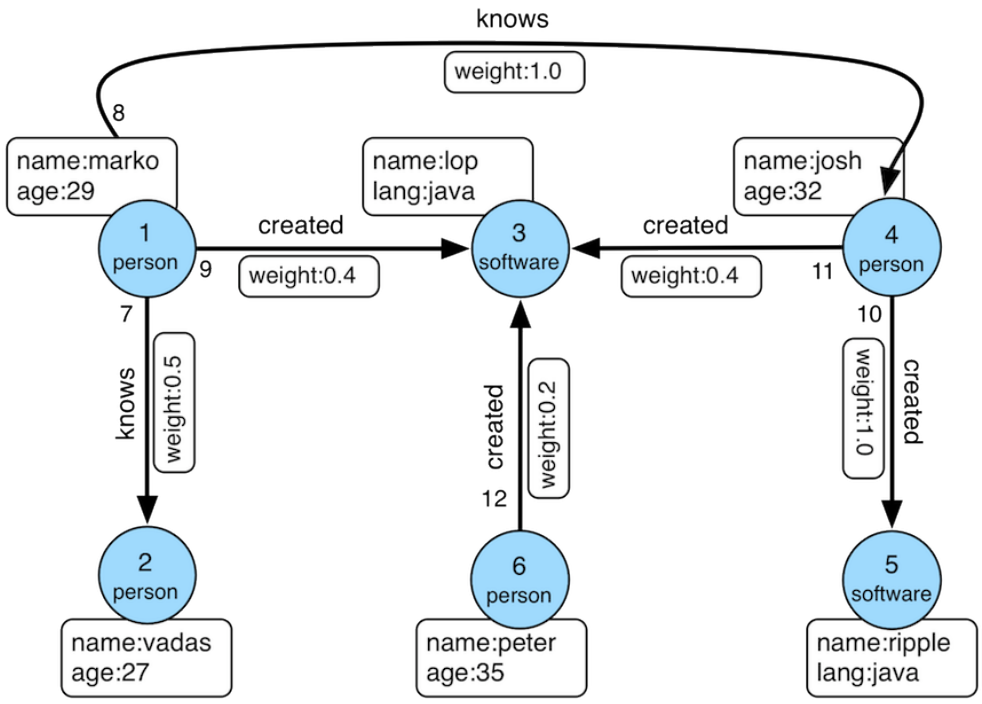
Implementace rozšíření je projekt typu Maven. Pro exekuci simulace postupujte následovně:

- **mvn compile** Kompilace zdrojových souborů.
- **mvn scala:testCompile** Kompilace testových souborů. Doporučuji před prvním spuštěním použít mvn test, který spustí zároveň testy.
- **mvn gatling:test** Spustění simulace pomocí pluginu Gatling. Defaultní simulace je definována v konfiguraci v souboru pom.xml

Defaultní simulace je třída GremlinModernSimulation, která je určená k testu schéma databáze Tinkerpop Modern 2.2. Spuštěním hlavní metody ve třídě EmbeddedServer dojde ke spuštění Gremlin serveru s touto databází v paměti.

Celý projekt je volně k dispozici v repositáři GitHub[1]

[1] `https://github.com/cervamar/gremlin-gatling`.



Obrázek 2.2: Tinkerpop Modern graf

Kapitola 3

Testování

V této kapitole demonstruji použití nástroje Gatling pro zátěžové testování. Rozhodl jsem se postupovat podle metodiky z kapitoly 1.1.6. a na hypotetickém příkladu demonstrovat celý proces testování.

Při vývoji hypotetické sociální sítě padlo rozhodnutí použít grafovou databázi pro persistenci dat. Nyní je potřeba ověřit, že databáze splňuje požadavky na výkon, které vyplývají z očekávané zátěže.

Na sociálních sítích se nabízí mnoho případů užití, my budeme uvažovat ty nejjednodušší:

1. Zobraz profil uživatele
2. Zobraz všechny profily, které má v oblibě uživatel
3. Zobraz všechny profily, které jsou v oblibě obou uživatelů

Složitější dotazy určeny například pro doporučování přátel bývají často předpočítány dopředu a obecně je to spíše směr OLAP systémů.

3.1 Testované databáze

Na základě žebříčku nejpopulárnějších databází[1] jsem se rozhodl pro databáze Neo4j, OrientDb a JanusGraph. Další kritérium byla bezplatná licence a možnost lokální instalace. Nejnovější verzí serveru, pro kterou existuje kompatibilní Gremlin driver pro všechny databáze je server verze 3.2.3.

3.1.1 OrientDb

OrientDB je první open source NoSQL databáze, která kombinuje model grafu s dokumenty v jedné databázi. Vývoj databáze začal v roce 2010. Patří stejně jako Neo4j do skupiny "opravdových" databází. Zajímavá je podpora jazyka OrientDB SQL, což je variace tradičního jazyka SQL. [32]

3.1.2 Neo4j

Neo4j je open source NoSQL grafová databáze vyvíjena v jazycích Java a Scala. Vývoj začal v roce 2003 a první veřejná verze byla k dispozici v roce 2007. Zdrojový kód je k dispozici v repositáři GitHub. Nativním dotazovacím jazykem je Cypher. Neo4j exceluje především v dotazech zaměřených na průchod grafu, díky efektivní reprezentaci uzlů a hran. [33]

Distribuovaná varianta Neo4 je postavená na *master-slave* architektuře. Všechna data jsou replikována a synchronizována mezi všemi instancemi. Díky tomu dokáže velmi dobře škálovat především čtecí operace. Neo4j je vhodná pro případy, kdy se graf vejde na jeden stroj. [34]

3.1.3 JanusGraph

Společnost Aurelius vyvinula open source grafovou databázi Titan. V roce 2015 byla společnost odkoupena firmou Datastax (Apache Cassandra). V důsledku této akvizice byl vývoj Titanu přerušen (poslední verze 1.0.0 je z roku 2015.) a postupně vznikly dva projekty: DSEGraph a JanusGraph. DSEGraph je komerční řešení firmy Datastax. Dá se říci, že původní tým přepsal Titan do specializovaného řešení pouze s databází Cassandra.

V lednu 2017 se několik firem, mimo jiné Google a IBM, rozhodlo založit projekt JanusGraph pod licencí Linux Foundation project s cílem rozvíjet původní Titan. [35] IBM následně ukončila svůj projekt IBM Graph a nahradila ho databází JanusGraph.[36] JanusGraph podporuje několik druhů databáze pro persistenci dat: Apache Cassandra®, Apache HBase®, Google Cloud Bigtable a Oracle BerkeleyDB. JanusGraph má nativní integraci s Apache TinkerPop™, Gremlin je tak nativním dotazovacím jazykem. JanusGraph je velmi zajímavý svou modulární architekturou, umožňující výběr implementace persistentního uložiště a volitelně také indexu.

Při výběru uložiště musíme vycházet z CAP teorému, tedy obětovat jeden z vrcholů trojúhelníku: konzistence, dostupnost a odolnost vůči přerušení.

Databáze	Verze databáze	Gremlin plugin
Neo4j	2.2.3	org.apache.tinkerpop neo4j-gremlin 3.2.3
OrientDb	2.2.12	com.michaelpollmeier:orientdb-gremlin:3.2.3.0
JanusGraph	0.1.1	org.janusgraph:janusgraph-all:3.2.3

Tabulka 3.1: Přehled testovaných databází

Volitelné je použití indexovacích nástrojů ElasticSearch či Apache Lucene pro rychlejší a komplexnější dotazy. [37]

3.1.4 Testovací prostředí

K testování byl použit běžný přenosný počítač, na kterém běžel Gremlin server s vnořenou testovanou databází. Simulace pak byla spuštěna pomocí pluginu Maven. Rozhodně se nejedná o ideální provedení, protože testovací nástroj a server běží na jednom stroji, avšak cílem této simulace je především ukázat, že můžeme jednoduše otestovat různé databáze bez nutných změn na straně testovacího nástroje.

Konfigurace testovacího prostředí se nalézá v příloze A.1.

Jediná věc, která by se mohla měnit, je konfigurační soubor potřebný pro připojení k serveru, avšak v našem případě poběží server vždy lokálně na stejném portu. Před jednotlivými testy je server vždy restartován s konfigurací pro danou grafovou databázi.

3.1.5 Vytvoření databází

Pro vytvoření databáze jsem postupoval podle následujícího postupu:

1. Stáhněte apache-tinkerpop-gremlin-server-$VERZE-bin.zip
2. Rozbalte archiv a přejděte do složky apache-tinkerpop-gremlin-server-$VERZE
3. Spusťte bin/gremlin-server.bat s parametrem -i $groupId $artifactId $VERZE
4. Vytvořte konfiguraci serveru conf/gremlin-server-$databaze.yaml
5. Vytvořte konfiguraci databáze conf/$databaze.properties
6. Spusťte bin/gremlin-server.bat conf/gremlin-server-$databaze.yaml

Ve třetím kroku server nainstaluje databázi jako plugin. Na pozadí Groovy Grape stáhne knihovnu pro danou databázi se všemi jejími závislostmi a uloží je do složky plugins. V konfiguraci ze čtvrtého kroku musíme specifikovat, které pluginy mají být použity, tzn., které knihovny se přidají do Java classpath. Konfigurace z pátého kroku má společnou jednu proměnnou, a to gremlin.graph, která odkazuje na jméno třídy grafu, který se má vytvořit.[31]

Všechny databáze v testu byly vytvořeny s persistencí na disk a nastaveny v netransakčním módu. Jejich konfigurace můžete najít v přílohách A.4, A.5 a A.6. Všechny konfigurace pochází z referenčních dokumentací, stejně tak konfigurace Gremlin serveru.

	Pokec	Podgraf určený k testování
Počet uzlů	1 632 803	10 000
Počet hran	30 622 564	121 716

Tabulka 3.2: Statistika dat ze sítě Pokec

3.2 Testovací data

Testovací data pochází ze slovenské sociální sítě Pokec. Model je velmi jednoduchý, jedná se o profily uživatelů, kteří si mohou přidávat ostatní uživatele jako přátele. Pokud model převedeme do grafu, uživatelé reprezentují uzly a přátelství orientované hrany mezi nimi.

Jména byla nahrazena číselnými identifikátory a data jsou tak anonymní. Profily obsahují mnoho informaci, avšak pro účely testu jsem vybral pouze identifikátor, věk, region a pohlaví. Data jsou součástí projektu Stanford Network Analysis Project, který mimo jiné poskytuje knihovnu dat určených pro analýzu a vytěžování dat. Množství dat bylo zbytečně velké pro náš scénář, omezil jsem testovací data pouze na profily s id 1 až 10000 a jím korespondující hrany.[38]

3.2.1 Import dat

Testovací data jsou k dispozici ke stažení ve formátu dvou textových souborů - jeden pro profily a druhý pro vazby. K importování jsem použil třídu PokecImporter, která čte vstupní soubor řádek po řádku a jednotlivě posílá dotazy na server. Tento postup není příliš efektivní oproti nativnímu importu jednotlivých databází, ale je univerzální a funguje pro všechny podporované databáze. Velikost testovacích dat odpovídá hodnotám v tabulce 3.2.

Během importu se ukázalo jako naprosto zásadní využití parametrických skriptů, protože kompilace je považována za drahou operaci. Import byl při použití neparametrických skriptů velmi pomalý a dokonce docházelo k zaseknutí serveru z důvodu nedostatku paměti. Pravděpodobně se snažil ukládat všechny dotazy do cache, avšak bez použití parametrů byly unikátní a zabíraly příliš mnoho místa. Když se použijí parametrické skripty, přeloží se pouze první z dotazů a ostatní už využívají zkompilované verze v mezipaměti serveru. Rozdíl mezi dotazy je vidět na ukázce 3.2. I na malém vzorku dat 1000 uzlů byl parametrický import téměř 3x rychlejší, konkrétně 3006 ms oproti 8877 ms.

Operace přidání hrany mezi existujícími uzly pomocí jejich id je velmi drahá, protože rozhraní vyžaduje jako argumenty instance uzlů. Proto musíme nejdřív uzly vyhledat a potom je spojit. Použitý databázový dotaz je v ukázce 3.1. V čem se jednotlivé databáze liší, je to, zda umožňují zvolit si identifikátor, respektive jestli naši volbu respektují. Například OrientDb si identifikátory přiřazuje sama, proto nelze přes původní id ze vstupu přistupovat přímo, ale

```
g.V().has("pokecId","1").as('v1')"
  .V().has("pokecId","2").as('v2')"
  .addE('likes').from('v1').to('v2')
```

Zdrojový kód 3.1: Vytvoření hrany mezi uzly s identifikátory 1 a 2.

```
//Neparametrické dotazy, které se vždy musí kompilovat
client.submit(graph.addVertex('pokecId','1'), Collections.
    emptyMap());
client.submit(graph.addVertex('pokecId','2'), Collections.
    emptyMap());
//Parametrické dotazy, pouze první se kompiluje
client.submit(graph.addVertex('pokecId','param'), Collections.
    singletonMap("param", "3"));
client.submit(graph.addVertex('pokecId','param'), Collections.
    singletonMap("param", "4"));
```

Zdrojový kód 3.2: Parametrické vs. neparametrické vytvoření uzlu.

přes přiřazenou vlastnost pokecId, do které jsme původní identifikátor uložili během importu.

Vybrané grafové databáze jsou perzistentní, proto stačí udělat import jednou. Za předpokladu, že test je omezen pouze na čtecí operace, je zajištěna konzistence dat před a po každé exekuci testu, a tak můžeme test opakovat na stejném vzorku dat.

```
def scn = scenario("Load test")
  .feed(idFeeder)
  .exec(gremlin("getProfile")
    .vertexByProperty("pokecId", "${pokecId}")
    .check(simpleCheck(res => res.size == 1))
    .extractResultAndSaveAs(parseVertexId, "profileId"))
    .pause(5)
    .exec(gremlin("getUserFriends")
    .neighbours("${profileId}", 1)
    .extractResultAndSaveAs(parseVertexId, "neighbourId"))
    .pause(5)
  .doIf("${neighbourId.exists()}") {
    exec(gremlin("getFriendProfile")
      .vertex("${neighbourId}")
      .check(simpleCheck(res => res.size == 1)))
    .exec(gremlin("getMutualFriends")
      .mutualNeigbours("${profileId}", "${neighbourId}"))
  }
```

Zdrojový kód 3.3: Ukázkový testovací scénář.

3.3 Ukázkový testovací scénář

Slovní popis scénáře:

1. Otevři profil uživatele.
2. Pauza 5 sekund.
3. Zobraz jeho přátele.
4. Pauza 5 sekund.
5. Pokud má uživatel nějaké přátele, zobraz profil prvního z nich, jinak ukonči test.
6. Zobraz společné přátele.

Implemetace v Gatlingu je v ukázce 3.3.

3.3.1 Ukázkové měření

Měření jsem rozdělil podle typů testů z úvodní kapitoly. Ve všech testech byl využit stejný scénář, ale lišilo se množství a strategie s jakou byli tvořeni virtuální uživatelé.

Pro každou databází jsem před měřením spustil test s pouhým jedním uživatelem a ověřoval správnost odpovědí. Před každým měřením jsem restartoval databázi a zahřál jí několika dotazy.

```
setUp(scn.inject(rampUsersPerSec 1 to 5 over 20,
    constantUsersPerSec 5 during 40 sec))
  .protocols(gremlinProtocol)
  .assertions(global.failedRequests.count.is(0),
     global.responseTime.percentile2.lt(200),
     global.responseTime.max.lt(1000))
```

Zdrojový kód 3.4: Nastavení load testu.

ASSERTIONS

Assertion	Status
Global: count of failed requests is 0.0	OK
Global: 75th percentile of response time is less than 200.0	OK
Global: max of response time is less than 1000.0	OK

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	936	936	0	0%	13.371	1	4	10	31	38	64	8	10
getProfile	240	240	0	0%	3.429	2	23	30	37	43	64	22	10
getUserFriends	240	240	0	0%	3.429	1	3	5	9	14	21	4	3
getFriendProfile	228	228	0	0%	3.257	1	3	4	7	8	11	3	2
getMutualFriends	228	228	0	0%	3.257	1	3	5	10	12	17	4	3

Obrázek 3.1: Výsledky zátěžového testu databáze Neo4j.

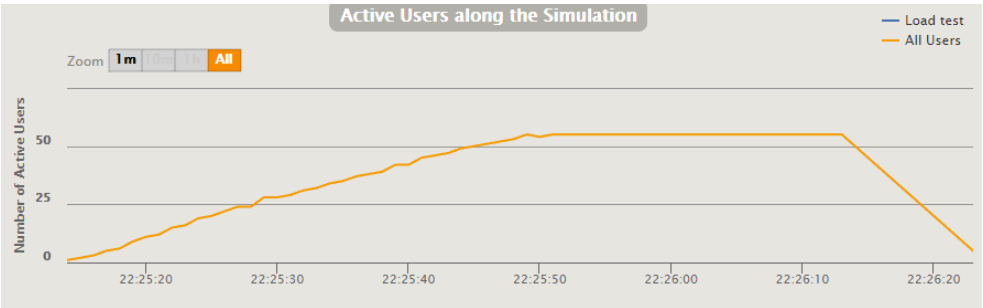
3.3.1.1 Zátěžový test

Akceptační kritéria zátěžového testu jsou:

- Do systému vstupuje 5 uživatelů za vteřinu.
- Žádný dotaz neskončí chybou.
- Doba odpovědi dotazu bude s percentilem 75 menší než 200 ms.
- Žádný dotaz nepřekročí dobu odpovědi 1000 ms.

Ověření akceptačních kritérií vykoná Gatling. Jejich implementace je v ukázce 3.4. Neo4j prošlá testem bez problémů a splnila akceptační kritéria 3.1. Rychlost odbavení požadavků byla natolik nízká, že nedocházelo k hromadění uživatelů a křivka aktivních uživatelů přesně odpovídá modelu toho, jak uživatele vstupují3.2.

JanusGraph neprošel testem zátěže kvůli vyšší době odpovědi na dotaz "getProfile". Tento dotaz hledá uzel na základě hodnoty pokecId a JanusGraph nevytvořil automaticky index na tuto vlastnost uzlu. Zbylé dotazy přistupují přes primární id záznamů a jsou vyhodnoceny rychleji. Příčinou neúspěchu byl tedy chybějící index, Tinkerpop API bohužel nenabízí API pro vytváření indexů.



Obrázek 3.2: Rozložení virtuálních uživatelů v čase během zátěžového testu databáze Neo4j.

ASSERTIONS

Assertion	Status
Global: count of failed requests is 0.0	OK
Global: 75th percentile of response time is less than 200.0	KO
Global: max of response time is less than 1000.0	OK

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	934	934	0	0%	13.155	1	4	233	297	362	492	75	123
getProfile	240	240	0	0%	3.38	226	273	291	347	437	492	281	39
getUserFriends	240	240	0	0%	3.38	1	3	4	5	10	17	3	2
getFriendProfile	227	227	0	0%	3.197	1	2	4	5	5	7	3	1
getMutualFriends	227	227	0	0%	3.197	1	5	6	10	15	19	5	3

Obrázek 3.3: Výsledky zátěžového testu databáze JanusGraph.

OrientDb prošla akceptačními kritérii, přestože jí také chyběl index na vlastnosti pokecId. Dosažené výsledky jsou o něco horší než u Neo4j. Výsledky testu jsou na obrázku 3.4.

Graf vývoje uživatelů v čase mají všechny databáze téměř totožný jako 3.2, křivka uživatelů je podobná vzorové z úvodní kapitoly.

Výsledky databází se liší v celkovém počtu dotazů, to je způsobeno tím, že z některých vrcholů nevedou žádné odchozí hrany a vrcholy jsou vybírány vždy náhodně. U všech databází docházelo shodně k nárůstu operační paměti, jak si ukládaly data do cache, stejně tak server si ukládal do cache zkompilované Gremlin dotazy. V testu byly použity parametrické dotazy, aby se minimalizovala režie serveru. I z toho důvodu bylo nutné provést restart serveru před každým testem, protože testovací vzorek by se vešel do paměti a opakované testy by dosahovaly naprosto rozdílných výsledků.

ASSERTIONS

Assertion	Status
Global: count of failed requests is 0.0	OK
Global: 75th percentile of response time is less than 200.0	OK
Global: max of response time is less than 1000.0	OK

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	942	942	0	0%	13.457	1	11	50	97	125	155	27	31
getProfile	240	240	0	0%	3.429	48	68	83	122	130	155	75	23
getUserFriends	240	240	0	0%	3.429	1	8	11	22	26	41	10	5
getFriendProfile	231	231	0	0%	3.3	1	5	8	18	22	29	6	5
getMutualFriends	231	231	0	0%	3.3	3	13	19	41	62	111	17	13

Obrázek 3.4: Výsledky zátěžového testu databáze OrientDB.

```
1  setUp(scn.inject(rampUsersPerSec(1) to 10 during 10,
       constantUsersPerSec(10) during 10,
2    rampUsersPerSec(10) to 20 during 10, constantUsersPerSec(20)
       during 10,
3    rampUsersPerSec(20) to 30 during 10, constantUsersPerSec(30)
       during 10,
4    rampUsersPerSec(30) to 40 during 10, constantUsersPerSec(40)
       during 10,
5    rampUsersPerSec(40) to 50 during 10, constantUsersPerSec(50)
       during 10))
6    .protocols(gremlinProtocol)
7    .assertions(
8      global.failedRequests.count.is(0))
```

Zdrojový kód 3.5: Nastavení stress testu.

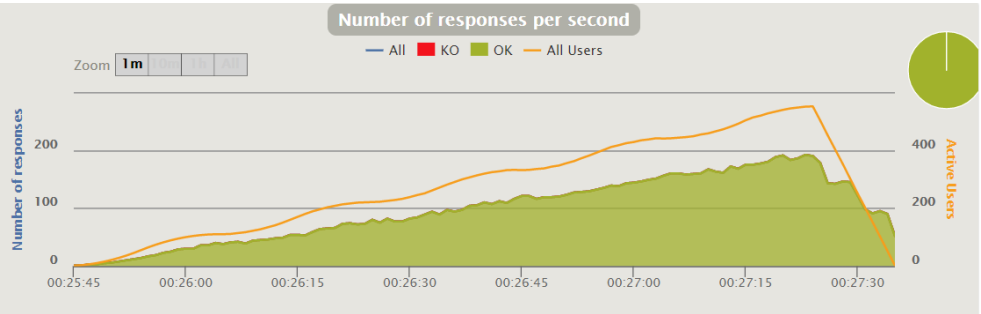
3.3.1.2 Stress Test

Předpokládejme, že hraniční zátěž našeho systému je 10x větší než v případě běžného provozu. Očekáváme, že systém bude stále odpovídat, dlouhá doba zpracování požadavku je tolerována.

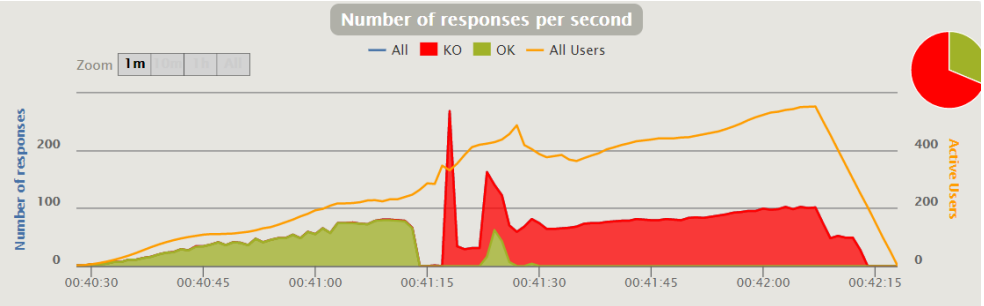
Akceptační kritéria zátěžového testu jsou:

- Do systému vstupuje postupně až 50 uživatelů za vteřinu.
- Žádný dotaz neskončí chybou.

Test použil totožný testovací scénář, lišila se implementace modelu uživatelů 3.5. Graf počtu uživatelů v systému by měl připomínat stoupající schody, jak se střídá konstantní zátěž s postupným zvyšováním. Neo4j prošel i tímto testem bez problému. Na obrázku 3.5 je vidět, že počet úspěšných dotazů koresponduje v grafu s počtem uživatelů. Požadavky byly stále odbavovány



Obrázek 3.5: Množství odpovědí databáze Neo4j v čase proložena křivkou množství virtuálních uživatelů.



Obrázek 3.6: Množství odpovědí databáze OrientDB v čase proložena křivkou množství virtuálních uživatelů.

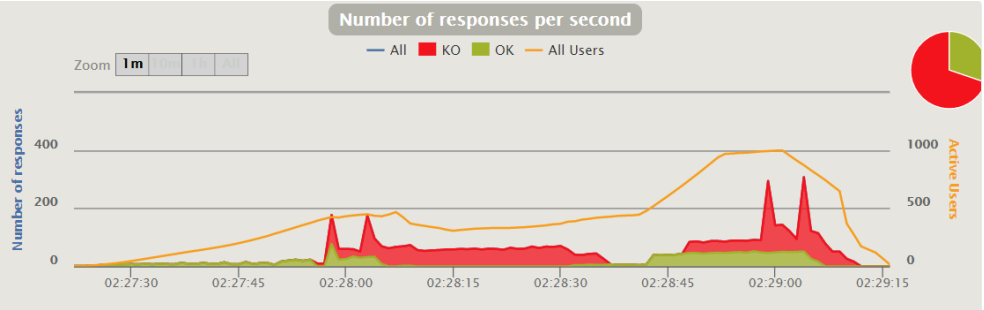
velmi rychle a nehromadily se ve frontě. Test ukázal, že zátěž stále pod limitem a mohli bychom jí ještě zvyšovat.

OrientDB testem neprošel, a dokonce úplně přestal odpovídat, jak můžeme pozorovat na obrázku 3.6. Když počet požadavků na databázi dosáhl hranice 80 dotazů za vteřinu, přestala databáze přijímat dotazy a pomalu odbavila část do té doby přijatých požadavků. Poté přestala pracovat úplně, procesor zůstal využíván na maximum a server musel být ručně zastaven, bohužel zanechal velké množství neuvolněné paměti.

Vzhledem k tomu, že JanusGraph neprošel ani testem zátěže, nemohl projít ani Stress testem. Přesto bylo zajímavé pozorovat, jak se vypořádal s enormní zátěží. Jak ukazuje obrázek 3.6, databáze dokázala úspěšně odbavit pouze zlomek dotazů, ale narozdíl od OrientDB nedošlo k zahlcení a po skončení testu byla databáze zpět v normálním stavu.

3.3.2 Vyhodnocení měření.

V kapitole měření jsem na hypotetickém scénáři prošel celým testovacím cyklem zátěžového testování. Měření bylo omezeno na jeden testovací cyklus,



Obrázek 3.7: Množství odpovědí databáze JanusGraph v čase proložena křivkou množství virtuálních uživatelů.

v dalších cyklech bychom se mohli zaměřit na ladění prostředí a optimalizaci databází a hotové testy by sloužily k porovnání změn.

Smyslem této kapitoly nebylo rozhodnout, která z testovaných databází je lepší, proto jsem se vyhnul jakýmkoliv optimalizacím jednotlivých databází. Přesto mohu prohlásit, že databáze Neo4j fungovala nejlépe v defaultní konfiguraci.

Závěr

V úvodní kapitole byla blíže popsána skupina typů testů zátěžového testování a představeny existující nástroje k testování se zaměřením na nástroj Gatling.

V následující kapitole byly představeny grafové databáze a grafové dotazovací jazyky. Poté jsem prozkoumal možnost rozšíření nástroje Gatling o nový protokol, na základě informací z předchozích kapitol jsem implementoval rozšíření pro testování grafových databází s využitím grafového jazyka Gremlin.

V kapitole testování jsem na ukázkovém scénáři demonstroval, že mnou implementované rozšíření Gatlingu je univerzální, protože byly použity různé grafové databáze testované totožnou simulací. Zároveň byl nástroj schopen vygenerovat zátěž odpovídající testům z kategorie zátěžového testování.

Univerzálnost nástroje je dána frameworkem Apache TinkerPop™, jehož je Gremlin součástí. Gremlin se může stát v budoucnu standardem grafových databází a v tom případě by se zvětšila podpora grafových databází.

Cíle práce, které jsem si stanovil v úvodu práce, byly splněny. Jedním z přínosů této práce nad rámec, je návod jak vytvořit rozšíření Gatlingu o nový protokol.

Implementace rozšíření je volně dostupná v GitHub[2] repositáři. V budoucnu bych rád rozšíření udělal více konfigurovatelné a rozšířil jej o sadu integračních testů s databázemi podporující TinkerPop™ ekosystém.

Pokud by někdo chtěl na mojí práci navázat, nabízí se využití k testování výkonu databáze v reálné konfiguraci a ověřit limity nástroje. Zajímavé by určitě bylo testování logikou komplexnějších a operační složitostí náročnějších scénářů. Samostatnou skupinou je testování výkonu cloud řešení.

[2] `https://github.com/cervamar/gremlin-gatling`

Literatura

[1] DB-Engines: *DB-Engines Ranking of Graph DBMS*. [online]. [cit. 2017-12-28]. Dostupné z: https://db-engines.com/en/ranking/graph+dbms

[2] Performance Testing – Testing for Speed, Stability, and Scalability. *TestLodge [online]*, listopad 2016, [cit. 2017-06-17]. Dostupné z: https://blog.testlodge.com/performance-testing/

[3] ITBiz.cz: *Úvod do zátěžového testování webových aplikací*. [online]. [cit. 2017-12-01]. Dostupné z: http://www.itbiz.cz/clanky/uvod-do-zatezoveho-testovani-webovych-aplikaci

[4] Automation Rhapsody: *Performance, Load, Stress and Soak testing*. [online]. [cit. 2017-12-01]. Dostupné z: http://automationrhapsody.com/performance-load-stress-and-soak-testing/

[5] Guru99: *Performance Testing Tutorial: Types, Process & Important Metrics*. [online]. [cit. 2017-12-02]. Dostupné z: https://www.guru99.com/performance-testing.html

[6] Kyle Rush: *Meet the Obama campaign's $250 million fundraising platform*. [online]. [cit. 2017-12-02]. Dostupné z: http://kylerush.net/blog/meet-the-obama-campaigns-250-million-fundraising-platform/

[7] JMeter: *How to Write a plugin for JMeter*. [online]. [cit. 2017-12-03]. Dostupné z: https://jmeter.apache.org/extending/jmeter_tutorial.pdf

[8] BlazeMeter: *Open Source Load Testing Tools: Which One Should You Use?* [online]. [cit. 2017-12-03]. Dostupné z: https:

//www.blazemeter.com/blog/open-source-load-testing-tools-which-one-should-you-use

[9] Etnetera: *Akka: Actor model a use cases pro výkonné paralelní systémy*. [online]. [cit. 2017-12-10]. Dostupné z: http://dev.etnetera.cz/scala/akka_actor_model_a_use_cases_pro_vykonne.html

[10] OctoPerf: *JMETER VS GATLING TOOL*. [online]. [cit. 2017-12-10]. Dostupné z: https://octoperf.com/blog/2015/06/08/jmeter-vs-gatling/

[11] Codecentric: *Gatling Load Testing Part 1 – Using Gatling*. [online]. [cit. 2017-12-15]. Dostupné z: https://blog.codecentric.de/en/2017/06/gatling-load-testing-part-1-using-gatling/

[12] Gatling: *Gatling documentation*. [online]. [cit. 2017-12-16]. Dostupné z: https://gatling.io/docs/2.3/

[13] Gatling: *Scaling Out*. [online]. [cit. 2017-12-17]. Dostupné z: https://gatling.io/docs/2.3/cookbook/scaling_out/

[14] Doc. RNDr. Petr Hlíněný, P.: *Zaklady Teorie Grafů pro (nejen) informatiky*. https://www.fi.muni.cz/ hlineny/Vyuka/GT/Grafy-text10.pdf: The name of the publisher, 2010.

[15] Neo4j: *RDF Triple Stores vs. Labeled Property Graphs: What's the Difference?* [online]. [cit. 2017-12-17]. Dostupné z: https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/

[16] Robinson, I.; Webber, J.; Eifrem, E.: *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., druhé vydání, 2015, ISBN 1491930896, 9781491930892.

[17] Microsoft: *Introduction to Azure Cosmos DB: Graph API*. [online]. [cit. 2017-12-20]. Dostupné z: https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction

[18] Forrester: *Vendor Landscape: Graph Databases*. [online]. [cit. 2017-12-20]. Dostupné z: https://reprints.forrester.com/#/assets/2/364/RES121473/reports

[19] IBM: *No more joins: An overview of Graph database query languages*. [online]. [cit. 2017-12-21]. Dostupné z: https://developer.ibm.com/dwblog/2017/overview-graph-database-query-languages/

[20] Neo4j: *Intro to Cypher*. [online]. [cit. 2017-12-22]. Dostupné z: https://neo4j.com/developer/cypher-query-language/

[21] Neo4j: *Meet openCypher: The SQL for Graphs*. [online]. [cit. 2017-12-22]. Dostupné z: https://neo4j.com/blog/open-cypher-sql-for-graphs/

[22] React, etc. Tech Stack: *Graph Query Languages: GraphQL, OpenCypher, Gremlin and SPARQL*. [online]. [cit. 2017-12-22]. Dostupné z: https://react-etc.net/entry/graph-query-languages-graphql-opencypher-gremlin-and-sparql

[23] Apollo: *GraphQL vs. REST*. [online]. [cit. 2017-12-22]. Dostupné z: https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b

[24] JanusGraph: *Chapter 6. Gremlin Query Language*. [online]. [cit. 2017-11-20]. Dostupné z: http://docs.janusgraph.org/latest/gremlin.html

[25] Apache TinkerPop™: *Apache TinkerPop™ TinkerPop-Enabled Providers*. [online]. [cit. 2017-11-25]. Dostupné z: http://tinkerpop.apache.org/providers.html

[26] Datastax: *The Benefits of the Gremlin Graph Traversal Machine™ TinkerPop-Enabled Providers*. [online]. [cit. 2017-11-26]. Dostupné z: http://tinkerpop.apache.org/providers.html

[27] Apache TinkerPop™: *TinkerPop3 Documentation*. [online]. [cit. 2017-11-30]. Dostupné z: http://tinkerpop.apache.org/docs/current/reference/

[28] Trivento: *Write a Custom Protocol for Gatling*. [online]. [cit. 2017-10-05]. Dostupné z: https://www.trivento.io/write-custom-protocol-for-gatling/

[29] Callista: *Creating a custom Gatling prococol for AWS Lambda*. [online]. [cit. 2017-10-05]. Dostupné z: http://callistaenterprise.se/blogg/teknik/2016/11/26/gatling-custom-protocol/

[30] Codecentric: *Gatling Load Testing Part 2 – Extending Gatling*. [online]. [cit. 2017-10-10]. Dostupné z: https://blog.codecentric.de/en/2017/07/gatling-load-testing-part-2-extending-gatling/

[31] JanusGraph: *Chapter 7. JanusGraph Server*. [online]. [cit. 2018-01-02]. Dostupné z: http://docs.janusgraph.org/latest/server.html

[32] OrientDB: *OrientDB - The World's First Distributed Multi-Model NoSQL Database with a Graph Database Engine*. [online]. [cit. 2018-01-08]. Dostupné z: http://orientdb.com/orientdb/

[33] Neo4j: *What is a Graph Database?* [online]. [cit. 2018-01-08]. Dostupné z: https://neo4j.com/developer/graph-database/

[34] ETH Zurich: *Big Data - Graph Databases*. [online]. [cit. 2017-12-20]. Dostupné z: https://www.systems.ethz.ch/sites/default/files/file/COURSES/2017%20FALL%20COURSES/big_data/14%20Data%20in%20the%20Very%20Small%20-%20Graph%20Databases.pdf

[35] datanami: *JanusGraph Picks Up Where TitanDB Left Off*. [online]. [cit. 2017-01-03]. Dostupné z: https://www.datanami.com/2017/01/13/janusgraph-picks-titandb-left-off/

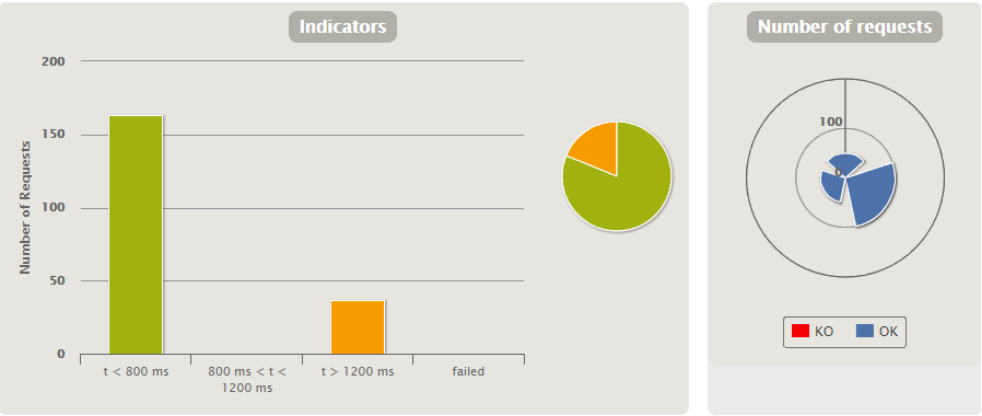
[36] IBM: *Announcing retirement of IBM Graph, availability of Compose for JanusGraph*. [online]. [cit. 2017-01-03]. Dostupné z: https://www.ibm.com/blogs/bluemix/2017/10/retirement-of-ibm-graph/

[37] JanusGraph: *JanusGraph Picks Up Where TitanDB Left Off*. [online]. [cit. 2017-01-03]. Dostupné z: http://docs.janusgraph.org/latest/arch-overview.html

[38] SNAP: *Pokec social network*. [online]. [cit. 2017-06-20]. Dostupné z: https://snap.stanford.edu/data/soc-pokec.html

Příloha A

Doplňkové materiály



Obrázek A.1: Globální statistiky

```
1  package cz.cvut.fit.cervamar.gatling.simulations
2
3  import io.gatling.core.Predef._
4  import io.gatling.http.Predef._
5  import io.gatling.http.request.builder.HttpRequestBuilder.
       toActionBuilder
6  import scala.concurrent.duration._
7
8  class ExampleHttpSimulation extends Simulation {
9
10   val httpProtocol = http
11   val stdUser = scenario("Bežný uživatel")
12     .exec(http("Otevři stránku vyhledávače")
13       .get("https://www.google.cz"))
14     .pause(3)
15     .exec(http("Najdi restaurace na Praze 6")
16       .get("https://www.google.cz/#q=restaurace+praha+6")
17       .check(status.is(200)).check())
18     .pause(2)
19
20   val advUser = scenario("Pokročilý uživatel")
21     .exec(http("Otevři stránku vyhledávače")
22       .get("https://www.google.cz"))
23     .pause(3)
24     .exec(http("Najdi restaurace na Praze 6, které nejsou
           vietnamské a mají .cz doménu")
25       .get("https://www.google.cz/#q=restaurace+praha+6+" +
26         "-vietnamská&as_sitesearch=.cz")
27       .check(status.is(200)).check())
28     .pause(2)
29
30   setUp(
31     stdUser.inject(atOnceUsers(100)),
32     advUser.inject(rampUsers(50) over (10 seconds))
33   ).protocols(httpProtocol).assertions(
34     global.responseTime.max.lt(100),
35     global.successfulRequests.percent.gt(95)
36   )
37 }
```

Zdrojový kód A.1: Http simulace z úvodní kapitoly o Gatlingu.

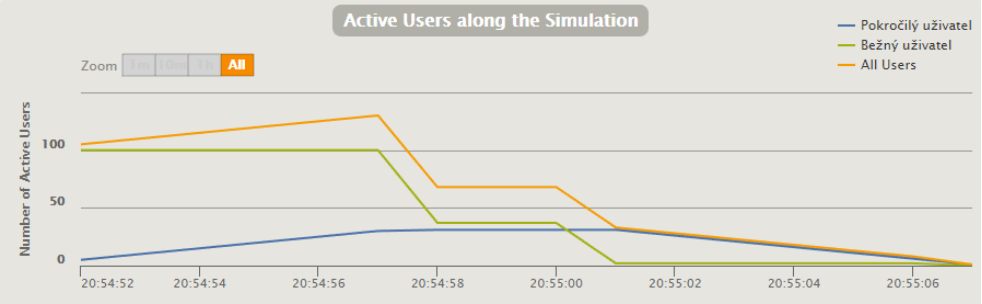
ASSERTIONS

Assertion	Status
Global: max of response time is less than 100	KO
Global: percentage of successful requests is greater than 95	OK

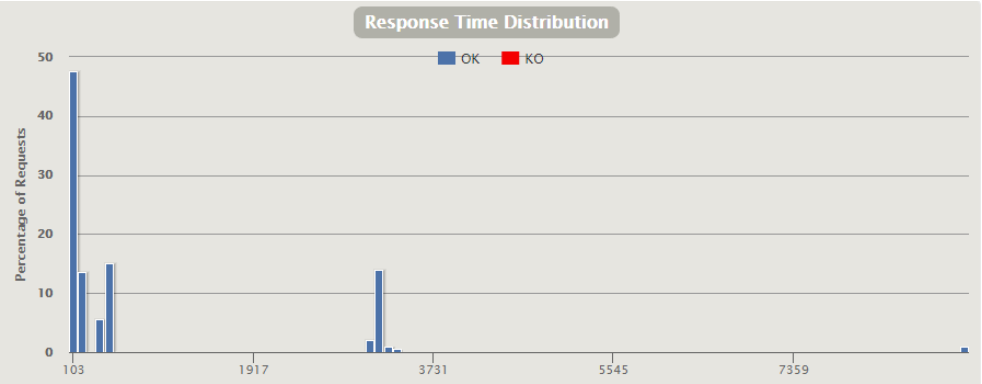
STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	200	200	0	0%	12.5	58	185	478	3209	3383	9127	813	1419
Otevři s...ledávače	50	50	0	0%	3.125	91	107	114	174	275	334	118	38
Najdi re... Praze 6	100	100	0	0%	6.25	174	478	3195	3234	9109	9127	1530	1732
Najdi re...z doménu	50	50	0	0%	3.125	58	75	79	107	136	145	76	16

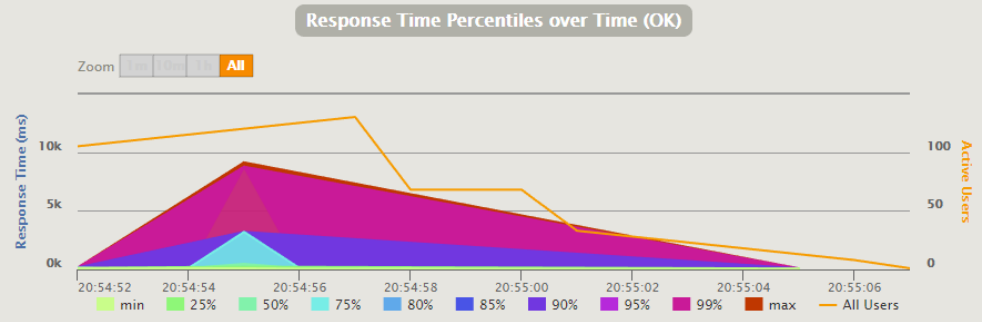
Obrázek A.2: Vyhodnocení jednotlivých dotazů a assertions.



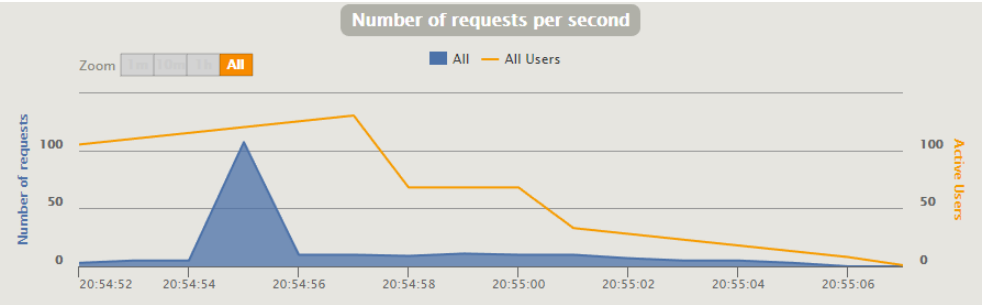
Obrázek A.3: Vývoj počtu aktivních uživatelů v čase.



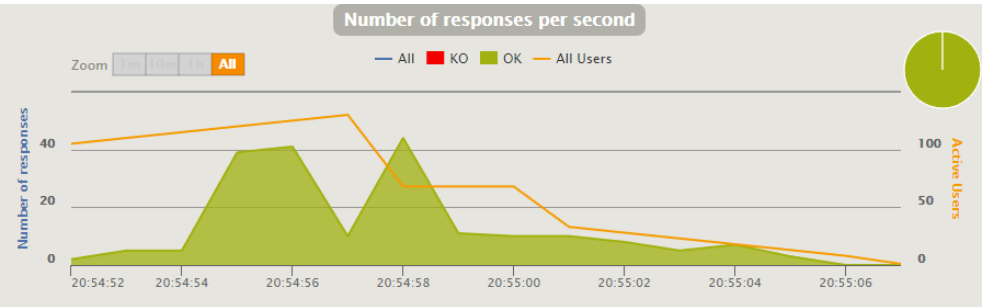
Obrázek A.4: Distribuce doby odezvy.



Obrázek A.5: Percentil doby odezvy v čase.



Obrázek A.6: Celkový počet položených dotazů v čase.



Obrázek A.7: Celkový počet obdržených odpovědí v čase.

```
{
  "simulation": "gatling.ExampleHttpSimulation",
  "simulationId": "examplehttpsimulation",
  "start": 1514928112352,
  "description": "",
  "scenarios": [
    "Bežný uživatel",
    "Pokročilý uživatel"
  ],
  "assertions": [
    {
      "path": "Global",
      "target": "max of response time",
      "condition": "is less than",
      "conditionValues": [
        100
      ],
      "result": false,
      "message": "Global: max of response time is less
         than 100",
      "values": [
        3144
      ]
    },
    {"path": "Global"...}
  ]
}
```

Zdrojový kód A.2: Výsledný report simulace ve formátu JSON

Zdrojový kód A.3: Výsledný report simulace ve formátu XML

```
1 <testsuite name="gatling.ExampleHttpSimulation" tests="2"
      errors="0" failures="1" time="0">
2     <testcase name="Global: max of response time is less than
          100" status="false" time="0">
3         <failure type="Global">Actual values: 3144</failure>
4     </testcase>
5     <testcase name="Global: percentage of successful requests
          is greater than 95" status="true" time="0">
6         <system-out>Global: percentage of successful requests
              is greater than 95</system-out>
7     </testcase>
8 </testsuite>
```

Základní deska	Gigabyte GA-970A-UD3
CPU	Intel Core i5-6300U @ 2.40GHz, 2 Cores
RAM	16 GB DDR4 SDRAM
Grafická karta	Intel HD Graphics 520
Pevný disk	512 GB SSD
Operační systém	Windows 10 Professional PRO 64-bit
IDE	IntelliJ Idea 2017.3.2 x64

Tabulka A.1: Hardware a software konfigurace použitá pro testování

Zdrojový kód A.4: Konfigurace databáze JanusGraph

```
gremlin.graph=org.janusgraph.core.JanusGraphFactory
storage.backend=berkeleyje
storage.directory=$CESTA
storage.transactions=false
```

Zdrojový kód A.5: Konfigurace databáze Neo4j

```
gremlin.graph=org.apache.tinkerpop.gremlin.neo4j.structure.
    Neo4jGraph
gremlin.neo4j.directory=$CESTA
gremlin.neo4j.conf.node_auto_indexing=true
gremlin.neo4j.conf.relationship_auto_indexing=true
```

Zdrojový kód A.6: Konfigurace databáze OrientDb

```
gremlin.graph=org.apache.tinkerpop.gremlin.orientdb.
    OrientGraph
orient-url=plocal:$CESTA
orient-user=root
orient-pass=rootpwd
orient-transactional=false
```

Příloha **B**

Seznam použitých zkratek

API Application programming interface

CRUD Create read update delete

CSV Comma-separated values

DSL Domain specific language

FTP File transfer protocol

GUI Graphical user interface

HTML Hypertext markup language

HTTP Hypertext transfer protocol

JDBC Java database connectivity

JMS Java message service

JSON JavaScript Object Notation

JVM Java virtual machine

LDAP Lightweight directory access protocol

OLAP Online analytical processing

OLTP Online transaction processing

POP3 Post Office Protocol

RDF Resource Description Framework

REPL Read eval print loop

REST Representational state transfer

SASL Simple Authentication and Security Layer

SMTP Simple Mail Transfer Protocol

TCP Transmission Control Protocol

URL Uniform Resource Locator

W3C World Wide Web Consortium

XML Extensible Markup Language

XMPP Extensible Messaging and Presence Protocol

Příloha C

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
reports ..... data z měření
src
    gremlin-gatling ..... zdrojové kódy implementace
    thesis ..... zdrojová forma práce ve formátu LaTeX
DP_Cervak_Marek_2017.pdf ..... text práce ve formátu PDF
```