MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Josef Holubec

Sběratel informací o výkonu dotazování v multi-modelových databázových systémech

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Ing. Pavel Koupil, Ph.D.
Studijní program: Informatika
Studijní obor: Informatika se specializací databáze a web

Praha 2025

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

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

V ..... dne ....

Podpis autora

Rád bych zde vyjádřil poděkování především vedoucímu práce Ing. Pavlu Koupilovi, Ph.D. za jeho trpělivost a odbornou pomoc. Dále bych rád poděkoval své rodině a přátelům za jejich pochopení pro věc a podporu během práce na tomto díle.

Název práce: Sběratel informací o výkonu dotazování v multi-modelových databázových systémech

Autor: Josef Holubec

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Ing. Pavel Koupil, Ph.D., Katedra softwarového inženýrství

Abstrakt: Ve světě se můžeme potkat s množstvím optimalizačních technik pro dotazování se nad relačními daty, ale neexistuje alternativa pro data multi-modelová. Pro vytvoření takového nástroje je ovšem potřebné sehnat adekvátní datovou sadu, nad níž by bylo možné model natrénovat. Jako je tomu u samotného modelu, vhodná datová sada zatím nebyla vytvořena. V této práci se tedy zaměříme na tvorbu nástroje, pomocí kterého bude možné vytvořit datovou sadu, která by umožnila tvorbu optimalizačního nástroje založeném na strojovém učení. Nejprve provedeme analýzu nástrojů, které jsou aktuálně dostupné a umožňují sběr statistických metadat z databázových systémů. Následně vybereme vhodné parametry, které by bylo dobré zahrnout do výsledné datové sady. Nakonec vytvoříme a popíšeme nástroj, jenž celý proces tvorby datové sady automatizuje.

Klíčová slova: Databázové metriky Multi-modelová data Výkon dotazů Relační databáze Grafové databáze Dokumentová databáze Datové sady

Title: Collector of information about query performance in multi-model database systems

Author: Josef Holubec

Department: Department of Software Engineering

Supervisor: Ing. Pavel Koupil, Ph.D., Department of Software Engineering

Abstract: In the world we can encounter many optimization techniques for querying over relational data, but none for multi-model data. To create one, we need to get an adequate dataset over which to train the model. As is the case with the model itself, a suitable dataset has not yet been created. Thus, in this thesis we will focus on creating a tool that can be used to create a dataset that would enable the creation of a machine learning based optimization tool. First, we will analyze the tools that are currently available and allow the collection of statistical metadata from database systems. We then select suitable parameters that would be good to include in the resulting dataset. Finally, we create and describe a tool that automates the whole process of creating the dataset.

Keywords: Database metrics Mutli-model data Query performance Relational databases Graph databases Document databases Dataset

Obsah

Úvod **3**

1 Rešerše **5**
1.1 Zabbix . . . 5
1.1.1 Architektura . . . 5
1.2 Prometheus . . . 7
1.2.1 Architektura . . . 7
1.3 Srovnání . . . 9

2 Analytická dokumentace **12**
2.1 Funkční a nefunkční požadavky . . . 12
2.2 Uživatelské role . . . 13
2.3 Případy užití . . . 13
2.4 Analýza sbíraných parametrů . . . 16
2.4.1 Databáze . . . 16
2.4.2 Kind . . . 18
2.4.3 Index . . . 21
2.4.4 Atribut . . . 23
2.4.5 Datový typ atributu . . . 26
2.4.6 Výsledek . . . 28

3 Návrhová dokumentace **30**
3.1 High level architektura . . . 30
3.2 Návrhové vzory . . . 32
3.2.1 Builder . . . 32
3.2.2 Abstract Factory . . . 32
3.2.3 Template Method . . . 33
3.2.4 Facade . . . 33
3.2.5 Adapter . . . 34
3.3 Použité technologie . . . 34
3.3.1 Java Database Connectivity (JDBC) . . . 34
3.3.2 PostgreSQL Driver . . . 34
3.3.3 Neo4j Driver . . . 35
3.3.4 MongoDB Driver . . . 35
3.3.5 H2 Database . . . 35
3.3.6 Spring boot . . . 36
3.3.7 Jackson . . . 36

4 Programátorská dokumentace **37**
4.1 Diagram tříd . . . 37
4.1.1 Server . . . 37
4.1.2 Datový model . . . 43
4.1.3 Wrappers . . . 47
4.1.4 Persistor . . . 51
4.2 Sekvenční diagram . . . 51

4.3 Aplikační rozhraní (API) . . . 54
4.3.1 DataModel . . . 54
4.3.2 Wrappers . . . 56
4.3.3 Server . . . 61
4.4 Implementační problémy . . . 62
4.4.1 Společné . . . 62
4.4.2 Neo4j . . . 64
4.4.3 MongoDB . . . 65
4.5 Testování . . . 66
4.5.1 Unit testy . . . 66
4.5.2 Integrační testy . . . 66

5 Uživatelská dokumentace **67**
5.1 Minimální HW a SW požadavky . . . 67
5.2 Instalace . . . 68
5.2.1 Dockerfile . . . 69
5.2.2 Docker Compose . . . 69
5.3 Spuštění . . . 72
5.4 Uživatelské rozhraní . . . 73
5.5 Ukázka použití . . . 75
5.5.1 Zavolání dotazu . . . 75
5.5.2 Zkušební dotazy . . . 78
5.5.3 Přidání instance . . . 79
5.5.4 Nastavení persistoru . . . 81
5.6 Chybová hlášení a jejich řešení . . . 81
5.6.1 Formát hlášek . . . 81
5.6.2 Rady pro řešení chyb . . . 82

Závěr **85**

Seznam použité literatury **88**

Seznam obrázků **89**

Seznam tabulek **90**

Seznam použitých zkratek **91**

Úvod

Pro zasazení této bakalářské práce do kontextu se potřebujeme seznámit s výzkumným projektem GAČR *Self-Adapting Management of Multi-Model Databases*, číslo projektu 23-07781S. Realizace vedená výzkumnou skupinou ADaM,[1] se dlouhodobě zabývá návrhem autonomních systémů pro správu multi-modelových dat. Vytvořená bakalářská práce tedy vznikla v rámci tohoto projektu, jenž se dlouhodobě zajímá o návrh schémat, jejich evoluci, propagaci změn do dat i dotazů, a také adaptivní optimalizaci výkonnosti [1].

V případě že chceme umět efektivně reagovat na změny a umožňovat inteligentní úpravy dotazů, potřebujeme k těmto činům disponovat nástroji, které budou schopné reportovat o dopadech nastalých změn. Informace tohoto typu se využívají hovoříme-li o tzv. *what-if* analýze. Jedná se o odhad, jenž se snaží určit, jakým způsobem by změna dotazu ovlivnila jeho výkonnost. Tento přístup pak slouží jako základ pro analytické rozhodování, nebo také jako vstup pro pokročilé metody optimalizace. Těmito metodami myslíme například optimalizování na bázi umělé inteligence či formálního modelování (např. ILP, heuristické vyhledávání, reinforcement learning apod.) [2].

Když se přesuneme do světa databázových systémů, tak jeden ze základních přístupů, jak se dostat k informacím o vyhodnocení jednotlivých dotazů, je použití nástroje `EXPLAIN`. Nástroj následně umožňuje nahlédnout do plánu pro evaluaci dotazu, jenž je sestaven plánovačem samotného databázového systému. Ze získaného plánu je následně možné vyčíst základní charakteristiky. Například se jedná o počty vykonaných operací, přístupy k použitým indexům a odhad na dobu zpracování. Formát poskytovaného plánu i parametry, které je možné získat, se ovšem napříč různými systémy liší. Jde nejen o poskytované informace, ale i o jejich rozdílnou reprezentaci. Například PostgreSQL poskytuje velmi detailní strom operací, MongoDB se zaměřuje především na informace o shardingu a agregace, a Neo4j poskytuje použité parametry formou grafu.

Tyto důvody znemožňují využívat pouze `EXPLAIN` výstupy pro jednotlivé systémy. Jejich nekonzistentnost pak výrazně ztěžuje možnost porovnání výkonnosti dotazů napříč jednotlivými systémy. Pokud se bavíme o systému MM-evocat [3], jehož cílem je přenášet dotazy mezi systémy s odlišnými datovými modely, je však taková porovnatelnost klíčová. Pro rozhodnutí, zda je nový dotaz výkonnější či alespoň srovnatelný s původním, potřebujeme mít dostupné metriky ke stejným charakteristikám, bez ohledu na konkrétní databázový systém.

Z tohoto důvodu je hlavním cílem této práce návrh a implementace modulu Collector – samostatné komponenty systému MM-evocat. V rámci práce se zaměříme na následující cíle:

C1: Vytyčení vhodných parametrů pro sběr metrik v kontextu tří vybraných zástupců různých datových modelů: PostgreSQL (relační model), MongoDB (dokumentový model) a Neo4j (grafový model). Umožníme tak srovnávat výkonnost dotazů nad systémy používající různé datové modely.

C2: Navrhneme architekturu modulu Collector, který bude schopen extrahovat

[1] https://www.ksi.mff.cuni.cz/area.html?id=multi-model-data

relevantní metriky pro výše vybrané systémy a ukládat sesbírané parametry do jednotného formátu

C3: Návrh architektury přizpůsobíme rozšiřitelnosti z důvodu budoucího přidávání podpory pro další databázové systémy

C4: Implementujeme REST API pro komunikaci s tímto modulem, čímž umožníme jeho integraci do systému MM-evocat.

Pro návrh Collectoru mohou být velkou inspirací již existující monitorovací systémy, jako je například Zabbix [4] nebo Prometheus. [5] Tyto systémy primárně neslouží k monitorování databázových dotazů, ale disponují architekturou, jež je založená na sběru různých metrik. Ke komunikaci využívají REST API a umožňují sesbíraná data vizualizovat. Na všech těchto úrovních je možné se jimi inspirovat a některé myšlenky zakomponovat do návrhu nového modulu.

Obsah práce. V kapitole 1 se budeme věnovat rešerši vybraných existujících systémů pro monitorování. Cílem bude získat inspiraci pro návrh architektury nového modulu. Při rešerši se zaměříme především na sbírané parametry, realizaci sběru těchto metrik a architekturu systémů. Kapitolu pak zakončíme srovnáním rozebraných systémů. V kapitole 2 se podíváme na funkční i nefunkční požadavky, popíšeme jednotlivé uživatelské role a následně připravíme model případů užití, na kterém budeme stavět při návrhu architektury systému. Kapitolu zakončíme analýzou sbíraných parametrů. V následující kapitole 3 se již začneme věnovat našemu modulu Collector. Detailně si popíšeme navrženou architekturu jednotlivých modulů systému, vybereme vhodné návrhové vzory a představíme si nástroje třetích stran využité pro implementaci systému. Dále, v kapitole 4, se budeme zabývat samotnou implementací celého modulu. Popíšeme si diagramy jednotlivých tříd, na což navážeme ukázkou sekvenčních diagramů pro sběr metrik v kontextu jednoho dotazu. Představíme si využité aplikační rozhraní, které je nutné znát, pokud budeme chtít systém rozšiřovat o podporu dalších databázových systémů. Na konci kapitoly se budeme věnovat problémům řešených při implementaci a stručně se vyjádříme k testování programu. Kapitola 5 pak nabídne uživatelskou dokumentaci. Nejprve si představíme postup k instalaci a spuštění programu, dále si ukážeme základní příklad použití a volání implementovaného REST API. Kapitolu pak ukončíme vstupem do stručného představení chybových hlášek a netradičního chování s doporučenými postupy k řešení vzniklých problémů. V závěru se pak vyjádříme k vytyčeným cílům a následně si představíme možnosti dalšího rozvoje systému.

1. Rešerše

V této kapitole, jak již název napovídá, se budeme věnovat rešerši existujících systémů. Nejedná se ovšem o systémy, které by vůči našemu programu byly nějakým způsobem konkurenční, jelikož slouží pro jiný účel. Hlavní myšlenkou této rešerše tak je prozkoumat, jak monitorovací systémy fungují, jaké sbírají parametry a jak jsou interně navrženy. Rešerše tedy slouží především k inspiraci pro budoucí implementaci a plynule na ni navážeme s kapitolou 2, kde se už budeme věnovat analytickému návrhu díla.

1.1 Zabbix

V této sekci se budeme bavit o monitorovacím systému *Zabbix* [4]. Tuto platformu vytvořil Alexei Vladishev a aktuálně se o její vývoj a údržbu stará stejnojmenná společnost se sídlem v Rize. Navzdory tomu, že je tento software vyvíjen uzavřenou společností, jedná se o open-source projekt, jenž je možné bezplatně využívat pod licencí *General public license version 2* (GPLv2). Společnost ovšem nabízí možnost technické podpory, jejíž objem se odvíjí od ceny a velikosti jednotlivých balíčků.

Zabbix je distribuovaný systém, s jehož pomocí lze monitorovat sítě, servery, databázové systémy nebo dokonce různá cloudová řešení. Používá k tomu flexibilní systém notifikací, kdy je možné nastavit upozornění pomocí mailu na nějaké nežádoucí změny či problémy ve sledovaném systému. Sbíraná data je pak možné efektivně vizualizovat pomocí webového rozhraní, kde je možné ovládat i různá nastavení samotného Zabbixu.

1.1.1 Architektura

Zde si podrobněji představíme jednotlivé komponenty aplikace. Podíváme se na to, jak spolu fungují a přiblížíme si tak softwarový návrh Zabbixu.

Databáze

Jednou z nejdůležitějších komponent systému je databáze. Nepřekvapivě má na starosti ukládání všech nasbíraných dat, ale zároveň jsou zde přítomné i informace o nastavení samotné platformy. Zabbix je kompatibilní s databázemi jako Oracle[1], MySQL[2], PostgreSQL[3] či MariaDB[4], pro kterou je ovšem potřeba mít nainstalovaný engine InnoDB.[5] Pro použití s Proxy serverem, o němž se ještě zmíníme podrobněji níže, je možné také použít databázový engine SQLite.[6]

[1] https://www.oracle.com/database/
[2] https://www.mysql.com
[3] https://www.postgresql.org
[4] https://mariadb.org
[5] https://mariadb.com/docs/server/server-usage/storage-engines/innodb
[6] https://sqlite.org

Zabbix server

Hlavou celého systému je Zabbix server, jeho hlavním úkolem je přijímání sbíraných dat a jejich následné ukládání do databáze. Sám pak také upozorní uživatele na pozorované nedostatky. Celý server je rozdělen do tří částí, kde první je samotný server starající se o výše uvedené úkony. Dále jde o webový server pro grafickou uživatelskou aplikaci a v neposlední řadě o databázi zmíněnou výše, se kterou všechny předchozí části komunikují.

Zabbix umožňuje fungování v tzv. *high availability* režimu (HA), pokud je potřeba mít garantovanou stálou dostupnost. Nativní řešení pak funguje tak, že pokud je k dispozici více Zabbix serverů, pak utvoří jeden *cluster* s uzly. Jeden z nich je v aktivním režimu a ostatní pouze napodobují všechny jím provedené operace. Uzly, které nejsou aktivní a procesy pouze kopírují, fungují v tzv. *standby* režimu. Pokud Aktivní server přestane fungovat, pak nějaký ze zbylých převezme jeho roli. Přesunutí této role funguje na principu voleb, kde ostatní uzly volí, kdo bude dalším nástupcem. Kromě tohoto nativního způsobu je údajně možné použít i řešení tohoto problému pomocí systémů třetích stran.

Webové rozhraní

Jde o další prvek, který umožňuje práci se Zabbixem odkudkoli a z prakticky jakéhokoli počítače. Většinou běží na stejném stroji jako hlavní server, ale není to pravidlem. Je v něm možné zobrazovat výsledky nasbíraných měření, či nastavovat samotnou monitorovací platformu.

Zabbix agent

Zabbix agent je komponenta běžící na cílovém stroji, kde funguje sledovaná aplikace. Stará se totiž o lokální sběr parametrů a veškeré výsledky posílá na *Zabbix server*. U novějších verzí Zabbixu jsou k dispozici dvě varianty agentů. Je zde k dispozici klasický *Zabbix agent*, který je napsaný v jazyce C a je tudíž lehký a kompatibilní s více platformami. Nebo je možné používat *Zabbix agent 2* psaný v relativně novém programovacím jazyce Go, který je tedy více flexibilní a jednodušeji rozšiřitelný pomocí dalších pluginů.

Oba agenti umožňují pasivní a aktivní získávání dat. Pasivní varianta funguje u obou stejným způsobem a to tak, že server pošle na agenta dotaz, aby mu získal nějakou informaci a agent tedy následně zjistí požadované hodnoty a pošle je serveru zpátky naměřené. U té aktivní mají oba obdobný princip základní funkcionality, ale Zabbix agent 2 pak nabízí nějaké možnosti rozšíření či větší kontroly nad prováděným sběrem dat. V základu jde o to, že agenti nejsou dotazovány serverem o nějaké informace, ale data pravidelně sbírají a sami je následně odesílají na server, když mají kompletní seznam. U lepšího Zabbix agenta pak je možnost tyto data posílat plánovaně (tj. např. v pravidelných intervalech) nebo kontroly parametrů z různých *pluginů* mohou být prováděny paralelně.

Zabbix proxy

Jde o komponentu, která není nutná k nasazení monitorovacího systému. Proxy server ovšem může hlavnímu serveru odlehčit, pokud jde o sběr dat a

monitorování dostupnosti jednotlivých uzlů. Je tak možné distribuovat a zrychlit práci, se kterou by se jinak Zabbix server musel vypořádat sám.

Pokud si uživatel chce zřídit proxy server, aby mohl práci v systému decentralizovat, pak je ovšem nutné počítat s tím, že proxy server vyžaduje mít vlastní databázi. Podpora je zde ovšem o něco méně obsáhlá než u hlavního Zabbix serveru. Vesměs je oficiálně možné používat pouze SQLite, MYSQL a PostrgreSQL systémy. Už u Oraclu Zabbix varuje, že jeho používání je sice možné, ale na vlastní riziko.

1.2 Prometheus

Prometheus [5] je open source monitorovací systém, původně postavený ve společnosti SoundCloud. Od svého počátku v roce 2012 se velké množství různých společností naučilo používat jeho služby, a tak si za dobu své existence vybudoval nemalou komunitu a to jak vývojářskou tak uživatelskou. Vývoj tohoto projektu, tak nezaštiťuje žádná komerční společnost, ale je závislý pouze na komunitě vývojářů.

Prometheus nabízí pro vizualizaci dat vlastní dotazovací jazyk *PromQL*[7]. Většina komponent celé architektury je naprogramována v jazyce Go, což umožňuje jednoduchou a rychlou instalaci.

1.2.1 Architektura

Podíváme se podrobněji na architekturu Promethea.

Datový tok

Prometheus monitoruje a sbírá data v časových řadách (angl. *time series*), kde každá naměřená hodnota obsahuje také časovou známku s přesností na milisekundy. Jeden záznam tedy obsahuje 64 bitové číslo s plovoucí desetinnou čárkou a následně čas, kdy byla hodnota naměřená v milisekundách. V tomto formátu se pak data i ukládají.

Úseky těchto řad jsou pak označeny pomocí názvu parametru, který se v řadě měřil a následujících popisků určujících další informace o naměřených parametrech. Každý popisek obsahuje název a pak vlastní hodnotu. Popisky s prázdnou hodnotou se berou jako neexistující. Pokud bychom tedy získali data pro parametr s celkovým počet HTTP požadavků, pak by tato řada mohla například obsahovat popisky `method=POST` a `handler=/api/tracks/`. Tyto údaje by pak tedy popisovaly počty požadavků v čase s použitou metodou `POST` a na relativní adrese `/api/tracks` pro naši uvažovanou webovou aplikaci, kterou chceme sledovat. Používání těchto značek, pak umožňuje dotazovacímu jazyku *PromQL* se dotazovat na údaje spadající do specifické kategorie a zpříjemňuje vizualizaci nasbíraných dat.

[7] https://prometheus.io/docs/prometheus/latest/querying/basics/

Prometheus server

Jde o hlavní komponentu celé architektury systému. Má na starosti získávání a následné ukládání dat. Následně poskytuje uložená data k vizualizaci a to za pomocí dotazovacího jazyka *PromQL*. Nepoužívá tedy žádnou externí databázi, ale sám plní tuto funkci.

Sběr dat pak funguje tak, že server kontaktuje příslušné úlohy (angl. *jobs*), o nichž se budeme bavit níže, jejichž adresu má buď staticky uloženou nebo jí získá pomocí modulu *Service discovery*. Tyto úlohy mu pak pošlou nasbírané výsledky v podobě časových řad, které jsou popsány výše. Server následně tyto řady zpracuje tak, že je ještě před uložením zanalyzuje a zjistí, zda je třeba upozornit uživatele na nějaké nestandardní hodnoty. Pokud se tak děje, tak následně upozorní další komponentu nesoucí název *AlertManager*, která se postará o to, aby uživateli například přes mail přišla upomínka, že se něco v systému děje.

Job

Instance je proces běžící na monitorovaném cíli, který se stará o sběr dat. Kolekce těchto instancí, které plní stejnou úlohu se nazývá *Job*. Tyto procesy jsou často duplikované, aby zlepšily škálovatelnost a výkonnost samotného sběru dat.

Když hlavní server pak žádá o data, rovnou kromě názvu parametru, o který má zájem, uvede k časové řadě také popisky obsahující identifikaci dané úlohy a instance, která poskytuje naměřené hodnoty.

Existuje také speciální varianta, a to tzv. *Short-lived job*, která není serverem přímo dotazována, ale v momentě kdy dojde k exitu příslušných instancí, tak odešle nasbíraná data do komponenty, která nese název *Pushgateway*, která výsledky dočasně uloží a je pak o ně dotazována serverem.

Klientské knihovny

Předtím, než jsou Instance z předchozího odstavce schopné sbírat data z monitorovaných systémů, je třeba využít klientské knihovny pro napsání instrumentace ke sběru dat. Je třeba tedy určit jaké parametry bude možné sbírat a dále i proces, jak je sbírat na úrovni systému, který chceme Prometheem sledovat.

Klientské knihovny jsou psané v různých jazycích a výběr závisí na jazyku, v němž je napsaná samotná aplikace. Pro programovací jazyky Go, Java (případně Scala), Python, Ruby a Rust jsou k dispozici nativní knihovny napsané vývojáři Promethea. Pro další výše neuvedené jazyky, je třeba použít neoficiální knihovny třetích stran. V nejhorším případě je nutné si napsat vlastní, pokud ani knihovna třetí strany neexistuje.

Na úrovni těchto knihoven se při sběru dat pro jednotlivé parametry pracuje s předem definovanými typy jednotlivých metrik, od kterých existují čtyři:

- Nejprve jde o tzv. *Counter*, jenž je využitý k reprezentaci monotónně rostoucích hodnot. Tato čísla tak mohou pouze růst a nebo být sníženy na nulu, pokud dojde k restartu celého měření.

- Následně se můžeme setkat s typem *Gauge*, kde jde především o hodnoty, které mohou libovolně růst a klesat.

- Jedním ze složitějších typů je pak *Histogram*, jenž naměřené hodnoty pro více parametrů rozmisťuje do uživatelem nastavitelných úseků. Nabízí pak možnost spočítat celkovou sumu všech naměřených hodnot či počet měřených parametrů.

- Posledním je pak *Summary*, který funguje stejně jako Histogram, ale nabízí navíc možnost získání informací v podobě spočítání různých kvantilových funkcí pro libovolně nastavitelné okno získaných dat.

Exporter

V některých případech je k extrakci dat nevhodné či dokonce nemožné použít klientskou knihovnu. Pak je nutné použít nějaký z dostupných exporterů pro tuto činnost. Některé z nich jsou opět psány pomocí komunity Promethea a lze je tak označit za oficiální.

Nejdůležitějším z exporterů je *JMX exporter*, který je schopný monitorovat data ze široké škály systémů běžících na *Java virtual machine* základu. Jde především o aplikace psané v Javě či jiné jazyky překládané do Javovského bajt kódu. Konkrétně je tak možné sledovat systémy jako Cassandra či Kafka a jiné další.

Vizualizace

S Prometheem je možné použít vizualizační nástroj jmenující se Grafana. Její backend je psán v jazyce Go a frontend pomocí TypeScriptu. Byla původně napsaná pro databázové systémy používající časové řady pro tok dat. Jako je právě Prometheus. V pozdější době se vyvinula a nyní je možné s její pomocí vizualizovat data i z některých relačních databází jako například MySQL.

1.3 Srovnání

Zde se podíváme na srovnání jednotlivých systému pro monitorování. Není zde srovnání s naším Sběratelem, neboť jak již bylo zmíněno v úvodu, nejedná se o konkurenční systém, ale chceme se pouze inspirovat monitorovacími systémy. Pro účely inspirace se bezesporu hodí podívat se, jak konkurenční systémy řeší dané problémy. Nejdůležitější je následně jaké mají různé přístupy k řešení důsledky, což nám může pomoci při výběru správného řešení.

Postupně si projdeme tabulku 1.1 a popíšeme si porovnávané parametry a hodnoty zadané v tabulce. Objasníme tím pro čtenáře možné nejasnosti, které by tímto mohly vzniknout ze špatného pochopení zápisu pozorovaných výsledků.

Open Source

Začneme vlastností vyjadřující, zda se jedná o open source projekt či nikoliv. Bohužel se tato vlastnost nedá nijak měřit a obecně se uvažuje, že aplikace buď touto vlastností disponuje či nikoliv. V případě Zabbixu je v tabulce uvedena hodnota ano s otazníkem. Význam je takový, že ačkoliv zdrojový kód Zabbixu je veřejně dostupný pod uvedenou licencí GPLv2, tak jeho komunita vlastně postrádá open source rozměr. Dle mého názoru je tento efekt způsoben politikou

	Prometheus	Zabbix
License	Apache 2.0	GPLv2
Open Source	✓	✓?
Technical Support	Community + Third party courses	Forum + Paid support from company
Client Technology	Go, Java, Python, Ruby, Rust	C, Go
Supported Databases	None - With inbuilt database	Oracle, PostgreSQL, MySQL, MariDB
Data Model	Time series	Relational
High Availability	✓	✓
Visualisation Tools	Grafana	Zabbix web application

✓ indikuje existenci dané vlastnosti pro systém.
✓? systém disponuje danou vlastností, ale není to zcela jednoznačné

Tabulka 1.1: Srovnání monitorovacích technologií

firmy, která zpoplatňuje využití lepší technické podpory navzdory tomu, že samotné používání systému zpoplatněné není.

Technical support

Ohledně technické podpory jsem nakonec zjistil, že v případě Zabbixu existuje fórum, na kterém se nejspíš dá dohledat řešení nějakých základních problémů. Za lepší technickou podporu od vývojářů Zabbixu při používání systému je ovšem nutné si připlatit. V případě Promethea pak fungují různé komunikační kanály, na kterých je možná se spojit s komunitou vývojářů a řešit své problémy. Některé firmy třetích stran nabízejí i kurzy na zacházení s Prometheem, po jejichž úspěšném absolvování obdrží účastník oficiální certifikát.

Client technology

Zabbix nabízí ke sběru dat na monitorovaných strojích vlastní agenty, které jsou psané v jazyce C nebo Go a následně je možné jejich funkčnost rozšiřovat vlastními pluginy. Prometheus se naproti tomu stará o správu oficiálních knihoven pro různé jazyky, pomocí kterých je uživatel schopen napsat si vlastní fungující alternativu pro Zabbix agenty.

Data Model

Jelikož Zabbix sbírá data v klasickém relačním formátu a nejde o časové řady, nabízí uživatelům benefit v podobě výběru databáze, do které se budou ukládat měřené statistiky. Výčet těchto databází je možné vidět v tabulce. Prometheus

pak obsahuje vlastní databázi, která data ukládá ve formátu časových řad a disponuje tak vhodnými nástroji pro jejich zpracování, například dotazovacím jazykem PromQL.

High Avaliability

Oba systémy jsou pak distribuované a umožňují tak fungování v High Availabity módu. Uplatňují se pak různé architektury pro sdílení a sběr dat. Úplnější popis je možné najít v předchozích sekcích, kde se architekturám věnujeme podrobněji.

Visualisation Tools

Zabbix má své vlastní webové rozhraní, které umožňuje vizualizovat nasbíraná data a není tak potřeba se pro tento účel integrovat s jinými aplikacemi. Naproti tomu Prometheus vlastním řešením nedisponuje, ale i v oficiální dokumentaci se doporučuje jej používat s nástrojem Grafana.

2. Analytická dokumentace

V této kapitole se podíváme na analytickou část návrhu programu. Plynule navážeme na předchozí kapitolu rešerše, kde jsme zjišťovali jaké techniky a architektury se používají pro řešení úloh podobných té naší. Nejprve shrneme funkční a nefunkční požadavky, které byly součástí zadání práce. Dále se budeme věnovat případu použití. Krátce se budeme věnovat uživatelským rolím a celou kapitolu zakončíme sekcí se sběrem parametrů.

2.1 Funkční a nefunkční požadavky

Jak jsme se již zmínili v úvodu do kapitoly, zde se podíváme na stručný výčet požadavků, jenž byly součástí zadání práce. Požadavky jsou rozděleny na funkční a nefunkční. Funkční ovlivňují činnost programu jako takovou a specifikují, co by měl program být schopen zvládnout. Oproti tomu nefunkční požadavky pojednávají o dodatečných podmínkách, které je nutné do implementace zakomponovat, ale již neovlivňují vlastní funkce.

Funkční požadavky

Funkční požadavky reflektují především požadavek na vytvoření datové sady o výkonu a exekuci různých dotazů pro různé databázové systémy.

F1: Sběr parametrů o výkonu dotazů (detailně rozebráno v sekci 2.4).

F2: Sběr statistických parametrů, např. velikost stránky, velikost databáze v bytech, velikost databáze v počtu stránek, velikost vyrovnávací paměti, velikost indexu, počet záznamů v indexu (detailně rozebráno v sekci 2.4).

F3: Analýza read-only dotazů (dotazy modifikující data nejsou povoleny).

F4: Názvy entit, kterých se dotaz přímo týká, např. názvy kindů, indexů, atributů, ... (detailně rozebráno v sekci 2.4).

F5: Analýza a zisk parametrů z exekučního plánu dotazu.

Nefunkční požadavky

Nefunkční požadavky reflektují především požadavky na obecnou modularitu a použitelnost programu.

N1: Modularita programu, aby byl snadněji rozšiřitelný.

N2: REST rozhraní pro komunikaci se serverem.

N3: Implementace wrapperů pro databázové systémy PostgreSQL, MongoDB a Neo4j.

N4: Serverová část bude implementována pomocí frameworku Spring Boot.

N5: Ukládání nasbíraných logů s parametry do systému MongoDB

2.2 Uživatelské role

Software ze své podstaty nerozlišuje žádné role uživatelů. Jde o aplikační server, který po spuštění nabízí výčet jednoduchých operací, které může provádět kdokoli, kdo je schopen se k němu po síti připojit. Jedná se tedy o jedinou roli, která je schopná provádět všechny podporované operace skrz poskytnuté REST rozhraní:

- Uživatel může nahlížet na připojené instance jednotlivých databázových systémů.
- Je schopen sbírat metadata o výkonu a exekuci jednotlivých dotazů pro jím specifikované aktuálně připojené instance.

K úpravě aktuálně připojených systémů je potřeba mít přístup ke konfiguračním souborům aplikačního serveru, kde je možné nastavit připojení k novým instancím podporovaných databázových systémů.

2.3 Případy užití

V této sekci si představíme možné případy užití programu. Pro lepší přehlednost dané problematiky je zde připravený diagram 2.1. Jednotlivé případy následně podrobněji probereme a vysvětlíme případné nejasnosti. Případy užití jsou celkem přímočaré a jednoduché. Plynule na tuto problematiku navážeme v sekci 4.2, kde podrobněji rozebereme především spouštění zařazených exekucí.

UC1 Vytvoření exekuce

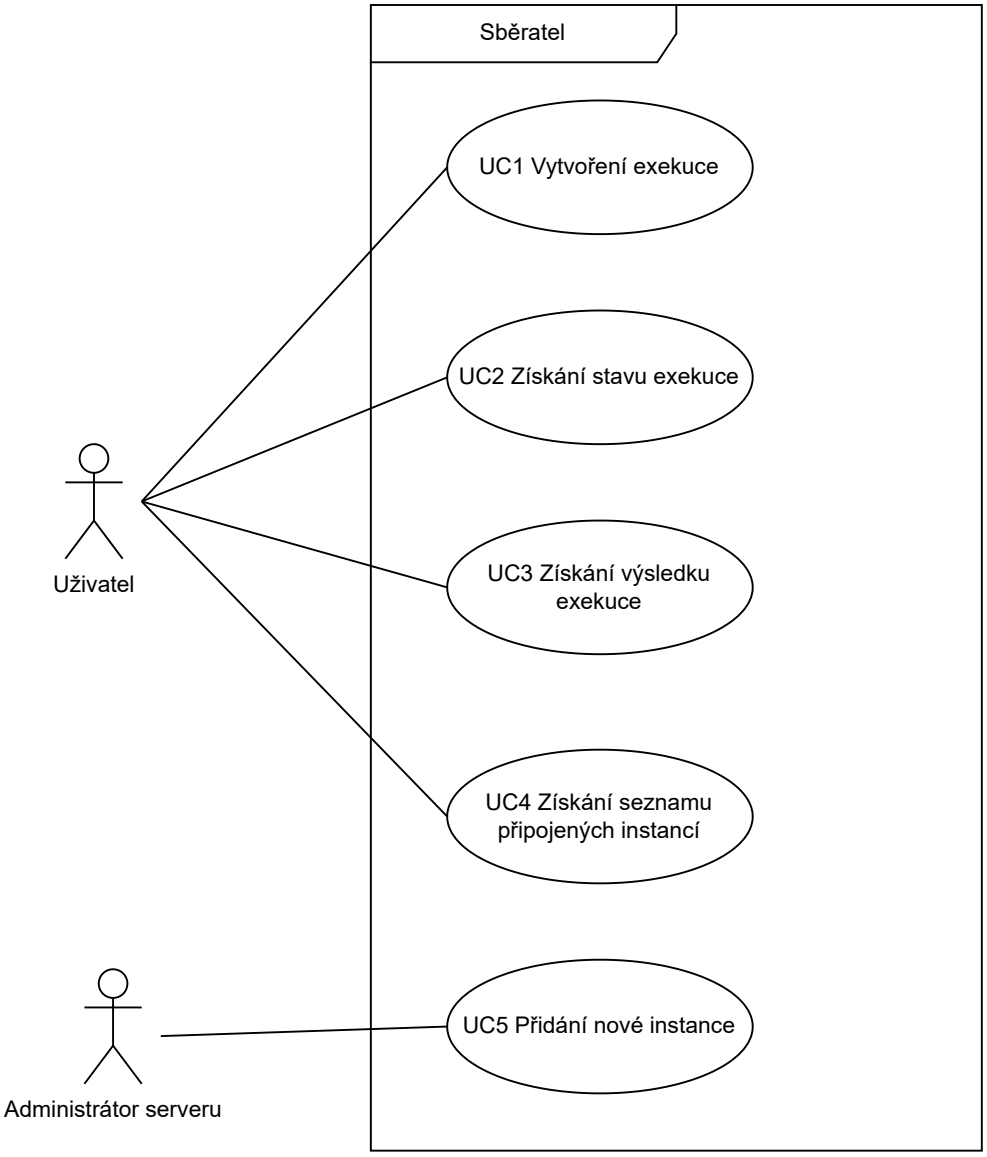
Normální tok

1. Uživatel pomocí REST-ového rozhraní specifikuje název instance a dotaz pro vytvoření exekuce
2. Server na-parsuje příchozí data z těla požadavku a vytvoří pro ně vhodný záznam
3. Server přidá záznam exekuce do fronty, z níž plánovač následně spouští vyhodnocení dotazů
4. Server pošle uživateli kladnou odpověď s identifikátorem nově vzniklé exekuce, alternativně v případě problému pošle chybu s chybovou hláškou

UC2 Získání stavu exekuce

Normální tok

1. Uživatel pošle požadavek s identifikátorem exekuce, u které se o stav zajímá
2. Server požadavek zpracuje a získá identifikátor exekuce
3. Server zjistí stav exekuce ve frontě



Obrázek 2.1: Případy užití

    - Pokud je záznam ve frontě, pak podle stavu vrací, že je čekající nebo běžící a v obou případech pokračuje bodem 5
    - Pokud záznam není ve frontě pokračuje bodem 4 a stav je ponechán nenalezen
4. Server zjistí stav exekuce v persistoru
    - Pokud je záznam v persistoru, stav je nastaven na dokončený a pokračuje dál
    - Pokud záznam v peristoru není přítomen, pak ponechává stav nenalezen a pokračuje dál
5. Nakonec vrátí uživateli zjištěný stav
    - Pokud je stav nenalezen, pak vrací chybu neexistující exekuce
    - Pokud je stav dohledán, pak vrací nalezený výsledek

UC3 Získání výsledku

Předpoklady

- Záznam buď neexistuje nebo je již zpracovaný – pro záznamy ve frontě se operace chová, jako kdyby exekuce neexistovala

Normální tok

1. Uživatel pošle požadavek s identifikátorem exekuce na získání výsledku
2. Server požadavek zpracuje a získá poslaný identifikátor
3. Pomocí identifikátoru získá výsledek z persistoru
4. Vrátí nalezený výsledek uživateli
    - Pokud nebyl výsledek nalezen, vrací uživateli chybu o neexistující exekuci

UC4 Získání seznamu připojených instancí

Normální tok

1. Uživatel pošle požadavek na server
2. Server požadavek zpracuje a deleguje na kontejner s wrappery
3. Kontejner podle načtených dat z konfiguračního souboru vytvoří seznam instancí
4. Server vrátí výsledek uživateli

UC5 Přidání nové instance

Předpoklady

- Může dělat pouze administrátor serveru, na kterém program běží

Normální tok

- Administrátor pozastaví aktuálně běžící server
- Upraví konfigurační soubor `application.properties`, specifikuje atributy potřebné pro připojení k nové instanci
- Opět spustí aplikaci
- Server při startu načte pozměněný konfigurační soubor a vytvoří všechny instance

2.4 Analýza sbíraných parametrů

Jak již název sekce napovídá, zde se budeme věnovat sbíraným parametrům. Parametry probereme postupně podle entit, ke kterým jsou vázané. Představíme si dotazy, kterými se dané parametry sbírají. Případně si ukážeme i jednotlivé výpočty, které se pro odhady parametrů, jenž není možné přímo zjistit, využívají. Při tvorbě dotazů a odhadů jsem čerpal především z dostupných dokumentačních materiálů na internetu pro analyzované systémy PostgreSQL [6], Neo4j [7] a MongoDB [8].

2.4.1 Databáze

Nejprve se podíváme na parametry související s databází, nad níž se dotaz spouští. Níže můžeme vidět tabulku 2.1, kde jsou vypsané všechny parametry, které v kontextu databáze sbíráme.

SystemName	PageSize	ByteSize	SizeInPages	CacheSize
PostgreSQL	✓	✓	–	✓
Neo4j	–	✓	–	✓
MongoDB	–	✓	–	✓

✓ : dotaz pro daný parametr existuje.
– : dotaz neexistuje a je nutné parametr odhadnout početně.

Tabulka 2.1: Srovnání parametrů pro databázi.

Tabulka indikuje, pomocí jednoduchých znaků ✓ a –, zda existuje dotaz pro získání daného parametru či nikoliv. Pokud takový dotaz neexistuje, pak je nutné jej alternativně odhadnout pomocí jiných parametrů a spočítat, či v některých

výjimečných případech je parametr pro nějaký systém kompletně ignorován a není sbírán.

Níže jsou vidět dotazy pro parametry z tabulky (tj. velikost stránky, velikost databáze v bytech a počtu stránek, velikost vyrovnávací paměti) – pokud dotaz nebyl objeven, pak je zde uveden výpočet, jakým se parametr odhaduje.

Velikost stránky

Velikost stránky je v jednotlivých systémech určena následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT current_setting('block_size');
```

- Neo4j neumožňuje. Nastaveno na 8192 bytů podle defaultního nastavení.
- MongoDB neumožňuje. Nastaveno na 4096 bytů podle defaultního storage enginu *Wired Tiger*.

Velikost databáze v bytech

Velikost databáze v bytech je v jednotlivých systémech určena následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT pg_database_size('<databaseName>');
```

- Neo4j umožňuje (✓) voláním dotazu:

```
CALL apoc.monitor.store()
```

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({dbStats: 1, scale: 1}).totalSize
```

Velikost databáze v počtu stránek

Velikost databáze v počtu stránek je v jednotlivých systémech určena následovně:

- PostgreSQL neumožňuje. Odhadnuto výpočtem: $\lceil \frac{n}{p} \rceil$, kde n je velikost databáze v bytech a p je velikost stránky.
- Neo4j neumožňuje. Odhadnuto výpočtem: $\lceil \frac{n}{p} \rceil$, kde n je velikost databáze v bytech a p je velikost stránky.
- MongoDB neumožňuje. Odhadnuto výpočtem: $\lceil \frac{n}{p} \rceil$, kde n je velikost databáze v bytech a p je velikost stránky.

Velikost nastavené vyrovnávací paměti

Velikost nastavené vyrovnávací paměti je v jednotlivých systémech určena následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT (CAST(setting as int) * pg_size_bytes(unit))
    AS shared_buffers
FROM pg_settings
WHERE name='shared_buffers';
```

- Neo4j umožňuje (✓) voláním dotazu:

```
SHOW settings
YIELD name, value
WHERE name="server.memory.pagecache.size"
```

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({serverStatus: 1})
    .wiredTiger
    .cache["maximum bytes configured"]
```

2.4.2 Kind

Dále se budeme věnovat relačním tabulkám, dokumentovým kolekcím, a vrcholům a hranám v grafu (souhrnně je budeme nazývat výrazem *kind*). Zde se budeme zabývat jejich parametry. Opět je zde k dispozici tabulka 2.2 nabízející srovnání pro podporu získávání daných parametrů mezi jednotlivými systémy.

SystemName	ByteSize	SizeInPages	RecordCount	ConstraintCount	AttributeNames
PostgreSQL	✓	✓	✓	✓	✓
Neo4j	–	–	✓	✓	✓
MongoDB	✓	–	✓	–	✓

✓ : dotaz pro daný parametr existuje.
– : dotaz neexistuje a je nutné parametr odhadnout početně.

Tabulka 2.2: Srovnání parametrů pro kind.

Z tabulky 2.2 bych se rád explicitně zmínil o jednom parametru. Konkrétně se jedná o názvy atributů pro daný kind. Na první pohled se může jevit zájem o tento parametr jako zbytečný, ale je potřebné si uvědomit, že chceme následně sbírat i parametry pro atributy. Což je důvod, proč potřebujeme znát jejich názvy a jsou tak důležitým parametrem. Níže je rozpis všech dotazů a odhadů využitých při zisku parametrů.

Velikost kindu v bytech

Velikost kindu v bytech je v jednotlivých systémech určena následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT pg_total_relation_size('<tableName>');
```

- Neo4j neumožňuje a není způsob, jak přímo zjistit. Dalo by se odhadnout pomocí počtu záznamů a vlastnostmi atributů. Řešeno průchodem všech entit s daným popiskem a počítání velikostí podle dokumentace o velikosti záznamů[1]: $n \cdot r + b$, kde n je počet záznamů, r je velikost jednoho záznamu uzlu nebo hrany a b je velikost všech atributů v bytech získaná průchodem všech uzlů s daným popiskem:

```
MATCH (n:<label>)
RETURN n
```

  A všech hran s daným popiskem:

```
MATCH ()-[e:<label>]->()
RETURN e
```

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({collStats: <collectionName>})
    .storageSize
```

Velikost kindu v počtu stránek

Velikost kindu v počtu stránek je v jednotlivých systémech určena následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT relpages
FROM pg_class
WHERE relname = '<tableName>';
```

- Neo4j neumožňuje. Odhadnuto výpočtem: $\lceil \frac{n}{p} \rceil$, kde n je velikost kindu v bytech a p je velikost stránky
- MongoDB neumožňuje. Odhadnuto výpočtem: $\lceil \frac{n}{p} \rceil$, kde n je velikost kindu v bytech a p je velikost stránky

[1] https://neo4j.com/developer/kb/understanding-data-on-disk

Počet záznamů v kindu

Počet záznamů v kindu je v jednotlivých systémech určen následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT reltuples
FROM pg_class
WHERE relname = '<tableName>';
```

- Neo4j umožňuje (✓) pro uzly voláním dotazu:

```
MATCH (n:<label>)
RETURN COUNT(n) AS count;
```

  A pro hrany voláním dotazu:

```
MATCH ()-[e:<label>]->()
RETURN COUNT(e) AS count;
```

  V praxi není ovšem ani jeden dotaz použit. Prochází se celá kolekce pro spočtení velikosti a při průchodu se uloží i počet záznamů Dotazy tedy nedává smysl použít. Teoreticky by mohlo být možné pomocí počtu záznamu velikost v bytech odhadovat, což ale nakonec není použito z důvodu ulehčení práce.

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({collStats: <collectionName>})
    .count
```

Počet omezení nad kindem

Počet omezení nad kindem je v jednotlivých systémech určen následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT relchecks
FROM pg_class
WHERE relname = '<tableName>';
```

- Neo4j umožňuje (✓) voláním dotazu:

```
SHOW constraints
YIELD labelsOrTypes
WHERE "<label>" IN labelsOrTypes
RETURN COUNT(*) AS count;
```

- MongoDB neumožňuje. Není vůbec podporováno, není možné zjistit ani dopočítat.

Názvy atributů pro kind

Názvy atributů jsou v jednotlivých systémech určeny následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT attname
FROM pg_stats
WHERE tablename = '<tableName>';
```

- Neo4j umožňuje (✓) pro uzly voláním dotazu:

```
CALL apoc.meta.nodeTypeProperties({includeLabels: ["<label>"]})
YIELD propertyName;
```

  A pro hrany volání dotazu:

```
CALL apoc.meta.relTypeProperties({rels: ["<label>"]})
YIELD propertyName;
```

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({
    aggregate: "<collectionName>",
    cursor: {},
    pipeline: [
        {
            $project: {
                "arrayOfKeyValue": { $objectToArray": $$ROOT }
            }
        },
        { $unwind: "$arrayOfKeyValue.k" },
        { $group: {
            _id: null,
            "allKeys": {$addToSet: "$arrayOfKeyValue.k"}
        }}
    ]
}).cursor.firstBatch[0].allKeys
```

2.4.3 Index

Nyní se podíváme na parametry sbírané v kontextu indexů. Při exekuci se konkrétní názvy indexů získají při parsování exekučního plánu. Jak je již dobrým zvykem, níže je pro toto téma srovnávací tabulka 2.3 pro jednotlivé systémy.

I v tomto případě bych rád explicitně zmínil jeden parametr. V systému PostgreSQL, pokud dotaz používá k získání výsledku indexy, není poznat, nad jakými tabulkami byly indexy použité. Jelikož chceme mít data kompletní a zahrnout do sběru parametrů i tyto tabulky, je nutné zpětně získat jejich názvy a přidat je do zpracování při analýze tabulek.

SystemName	ByteSize	SizeInPages	RecordCount	KindName
PostgreSQL	✓	✓	✓	✓
Neo4j	–	–	–	✓
MongoDB	✓	–	✓	–

✓ : dotaz pro daný parametr existuje.
– : dotaz neexistuje a je nutné parametr odhadnout početně.

Tabulka 2.3: Srovnání parametrů pro index.

Velikost indexu v bytech

Velikost indexu v bytech lze v jednotlivých systémech získat následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT pg_total_relation_size('<indexName>');
```

- Neo4j neumožňuje. Odhadnuto dle: $\lceil \frac{m}{3} \rceil$, kde m je počet bajtů získaný průchodem po entitách s konkrétními atributy, nad nimiž je index entity postaven.[2] Průchod uzlů je pomocí dotazu:

```
MATCH (n:<label>)
RETURN n.indexProperty1, n.indexProperty2, ...
```

  A průchod hran pomocí:

```
MATCH ()-[e:<label>]->()
RETURN e.indexProperty1, e.indexProperty2, ...
```

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({collStats: <collectionName>})
    .indexSizes
    .<indexName>
```

Velikost indexu v počtu stránek

Velikost indexu v počtu stránek lze v jednotlivých systémech získat následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT relpages
FROM pg_class
WHERE relname = '<indexName>';
```

[2] Odhad na třetinovou velikost indexu pochází od https://neo4j.com/developer/kb/understanding-data-on-disk.

- Neo4j neumožňuje. Odhadnuto výpočtem: $\lceil \frac{i}{p} \rceil$, kde i je velikost indexu v bytech a p je velikost stránky.
- MongoDB neumožňuje. Odhadnuto výpočtem: $\lceil \frac{i}{p} \rceil$, kde i je velikost indexu v bytech a p je velikost stránky.

Počet záznamů v indexu

Počet záznamů v indexu lze v jednotlivých systémech získat následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT reltuples
FROM pg_class
WHERE relname = '<indexName>';
```

- Neo4j neumožňuje. Při průchodu entitami při počítání velikosti indexu se počítá i jejich počet.
- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand(
    {count: <collectionName>, hint: <indexName>}
).n
```

Název kindu pro index

Název kindu lze v jednotlivých systémech získat následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT tablename
FROM pg_indexes
WHERE indexname = '<indexName>';
```

- Neo4j umožnuje (✓) voláním dotazu:[3]

```
SHOW indexes
YIELD name, labelsOrTypes
WHERE name = "<indexName>";
```

- MongoDB neumožnuje. Řešení není potřebné, název je možné získat z exekučního plánu dotazu.

2.4.4 Atribut

Zde se podíváme na parametry sbírané v kontextu jednotlivých atributů. K dispozici je opět tabulka 2.4 s přehledným srovnáním jednotlivých systémů o podpoře sbíraných parametrů.

[3]Dotaz však není potřebný, neboť název kindu je možné získat z exekučního plánu dotazu. Není tedy Sběratelem využíván.

SystemName	Mandatory	DistinctValuesCount	AttributeTypes
PostgreSQL	✓	✓	✓
Neo4j	✓	✓	✓
MongoDB	✓	✓	✓

✓ : dotaz pro daný parametr existuje.
– : dotaz neexistuje a je nutné parametr odhadnout početně.

Tabulka 2.4: Srovnání parametrů pro atribut.

Povinný atribut

Informace zda-li se jedná o povinný atribut lze v jednotlivých systémech získat následovně:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT a.attnotnull
FROM pg_class as c
INNER JOIN pg_attribute as a ON a.attrelid = c.oid
WHERE a.attname = '<colName>'
    AND c.relkind = 'r'
    AND c.relname = '<tableName>';
```

- Neo4j umožňuje (✓) pro uzly voláním dotazu:

```
CALL apoc.meta.nodeTypeProperties({includeLabels: ["<label>"]})
YIELD propertyName, mandatory
WHERE propertyName = "<propertyName>"
RETURN propertyName, mandatory;
```

  A pro hrany voláním dotazu:

```
CALL apoc.meta.relTypeProperties({rels: ["<label>"]})
YIELD propertyName, propertyTypes, mandatory
WHERE propertyName = "<propertyName>"
RETURN propertyName, propertyTypes, mandatory;
```

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({
    listCollections: 1,
    filter: {
        name: <collectionName>
    }
}).options.validator.$jsonSchema.required
```

  Každá kolekce má jeden validační dokument, v němž jsou podobné informace uloženy. Dotaz tedy přistupuje na tento dokument a následně vyčte, zda se jedná o povinný atribut kolekce.

Počet hodnot atributu

Počet unikátních hodnot pro atribut lze v jednotlivých systémech získat takto:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT COUNT(DISTINCT <columnName>)
FROM <tableName>
```

- Neo4j umožňuje (✓) pro uzly voláním dotazu:

```
CALL apoc.schema.properties.distinct("<label>", "<propertyName>")
YIELD value
RETURN size(value) AS count;
```

  Pro hrany jsem nenašel žádnou vhodnou alternativu a tento parametr se v kontextu hran tedy ignoruje.

- MongoDB umožňuje (✓) pomocí dotazu:

```
db.runCommand({
    aggregate: <collectionName>,
    cursor: {},
    pipeline: [
        { $group: {"_id": "$<fieldName>"}},
        { $count: "count"}
    ]
}).cursor.firstBatch[0].count
```

Datové typy atributu

Pro sběr parametrů o datových typech potřebujeme znát, o jaké typy se máme zajímat. Potřebujeme tedy zjistit pro každý atribut, jaké ukládá typy. To se ve všech systémech dá zjistit následujícím způsobem:

- PostgreSQL umožňuje (✓) voláním dotazu:

```
SELECT t.typname
FROM pg_class as c
INNER JOIN pg_attribute as a ON a.attrelid = c.oid
INNER JOIN pg_type as t ON a.atttypid = t.oid
WHERE a.attname = '<colName>'
    AND c.relkind = 'r'
    AND c.relname = '<tableName>';
```

- Neo4j umožňuje (✓) pro uzly voláním dotazu:

```
CALL apoc.meta.nodeTypeProperties({includeLabels: ["<label>"]})
YIELD propertyName, propertyTypes
WHERE propertyName = "<propertyName>"
RETURN propertyName, propertyTypes;
```

A pro hrany voláním dotazu:

```
CALL apoc.meta.relTypeProperties({rels: ["<label>"]})
YIELD propertyName, propertyTypes
WHERE propertyName = "<propertyName>"
RETURN propertyName, propertyTypes;
```

V praxi nejsou dotazy použity a místo toho se znovu využívají data nasbíraná průchodem kindu.

- MongoDB umožňuje (✓) voláním dotazu:

```
db.runCommand({
    aggregate: "<collectionName>",
    cursor: {},
    pipeline: [
        {
            $project: {
                fieldType: {"$type": "$<fieldName>"},
                _id: 0
            }
        },
        {
            $group: {
                _id: {fieldType: "$fieldType"},
                count: {$sum: 1}
            }
        }
    ]
}).cursor.firstBatch[0]
```

2.4.5 Datový typ atributu

V původním návrhu měl každý atribut právě jeden typ a parametry typu tudíž byly součástí samotného atributu. S přidáváním podpory pro NoSQL systémy je ovšem nutné umožnit pro jeden atribut existenci většího počtu typů pro jeden atribut. Vlastnosti datového typu pro atribut se tudíž od původního atributu přemístily do vlastní třídy a zaslouží si i vlastní sekci v kontextu sbíraných parametrů. K dispozici je čtenáři opět tabulka 2.5 se srovnáním sbíraných parametrů pro datové typy atributů.

SystemName	**ByteSize**	**Ratio**
PostgreSQL	✓	–
Neo4j	–	–
MongoDB	✓	✓

✓ : dotaz pro daný parametr existuje.
– : dotaz neexistuje a je nutné parametr odhadnout početně.

Tabulka 2.5: Srovnání parametrů pro datový typ atributu.

Velikost v bytech

- PostgreSQL umožňuje (✓) pomocí dotazu:

```
SELECT avg_width
FROM pg_stats
WHERE tablename = '<tableName>'
    AND attname = '<colName>';
```

- Neo4j neumožňuje. Velikosti jsou odhadovány pomocí již odkazované dokumentace.[4]

- MongoDB umožňuje (✓) pro vnořené dokumenty pomocí následujícího dotazu:

```
db.runCommand({
    aggregate: "<collectionName>",
    cursor: {},
    pipeline: [
        {
            $match: {
                "<fieldName>": { $type: "<typeName>" }
            }
        },
        {
            $project: {
                "size": { "$bsonSize": "$<fieldName>" }
            }
        },
        {
            $group: {
                _id: null,
                avg: { $avg: "$size" }
            }
        }
    ]
}).cursor.firstBatch[0].avg
```

  Pro řetězce pomocí následujícího dotazu:

```
db.runCommand({
    aggregate: "<collectionName>",
    cursor: {},
    pipeline: [
        {
            $match: {
                "<fieldName>": { $type: "<typeName>" }
            }
        },
        {
```

[4] https://neo4j.com/developer/kb/understanding-data-on-disk

```
            $project: {
                "size": { "$binarySize": "$<fieldName>" }
            }
        },
        {
            $group: {
                _id: null,
                avg: { $avg: "$size" }
            }
        }
    ]
}).cursor.firstBatch[0].avg
```

A pro numerické hodnoty je odhadován podle jejich předepsané velikosti dle dokumentace

Poměr zastoupení typu pro atribut

- PostgreSQL neumožňuje. V jeho kontextu taková hodnota nedává smysl. Pro náš systém je automaticky parametr plněn hodnotou 1, neboť každý sloupec má právě jeden datový typ.
- Neo4j neumožňuje. Odhadnuto průchodem kindu a počtem nalezených typů pro každý atribut.
- MongoDB umožňuje (✓) pomocí dotazu 2.4.4 pro získání typů, který během agregace zároveň počítá jejich výskyty, z nichž se podílem získá procentuální zastoupení typu pro daný atribut.

2.4.6 Výsledek

Nakonec se budeme zabývat analýzou a sbíranými parametry samotného výsledku. Každý dotaz zadaný uživatelem se spouští hned ze začátku s příkazem pro získání exekučního plánu. Z exekučního plánu lze ve všech systémech, pro něž jsem implementoval wappery, získat čas exekuce pro daný dotaz. Což je jeden z nejdůležitějších parametrů ukládaných pro výsledek spouštěného dotazu.

Následně se o výsledku sbírají obdobná statistická data jako v případě kindů z databáze. Při porovnání výkonu dotazu to tak umožňuje porovnat, jak moc byl dotaz selektivní, což může mít vliv na samotný čas exekuce. Bezesporu to tak umožňuje zajímavá porovnání. Když se pak začneme zajímat o možnosti sběru statistických dat tohoto typu, narazíme na problém v podobě neexistence kindu pro daný výsledek dotazu, takže nemáme entitu, vůči níž bychom se mohli na parametry dotazovat.

Toto by se dalo opravit vytvářením materializovaných pohledů, na které systémy často nahlíží jako na určité typy kindů. Takové řešení má hned několik problémů a je spíše cestou do pekel než řešením. První je fakt, že administrátor databázového systému by nám nejspíše nepoděkoval, pokud bychom pro každý dotaz vytvářeli nový materializovaný pohled. Dále by to zkreslovalo výsledky pro stejné dotazy, které by se vyhodnocovaly s časovým odstupem. Nakonec pak ne

všechny databázové systémy tvorbu pohledů podporují, například databáze Neo4j s pohledy vůbec nepracuje.

Řešením v našem případě bylo vytvořit speciální typ pro konzumaci výsledků, který podle typů atributů iterativně výsledek zpracuje a statistická data odhadne. Odhad se provádí na základě statistik sesbíraných ze všech relevantních kindů a indexů. Víme totiž, jak jsou atributy velké, jestli jsou povinné a jaké mají typy. Tudíž umíme velikosti při zpracování výsledku odhadnout podobnou logikou jako při počítání velikostí pro uzly či hrany určitého typu v systému Neo4j.

Dotaz pro získání plánu exekuce

Spouštění dotazu pro získání plánu exekuce podporují všechny systémy a tudíž jsem nakonec netvořil pro tento případ žádnou srovnávací tabulku. Plány se dají získat v systémech těmito způsoby:

- PostgreSQL umožňuje dotazem:

```
EXPLAIN (ANALYZE TRUE, FORMAT JSON) <query>
```

- Neo4j umožňuje dotazem:

```
PROFILE <query>
```

- MongoDB umožňuje dotazem:

```
db.runCommand({
    explain: {<query>},
    verbosity: executionStats
})
```

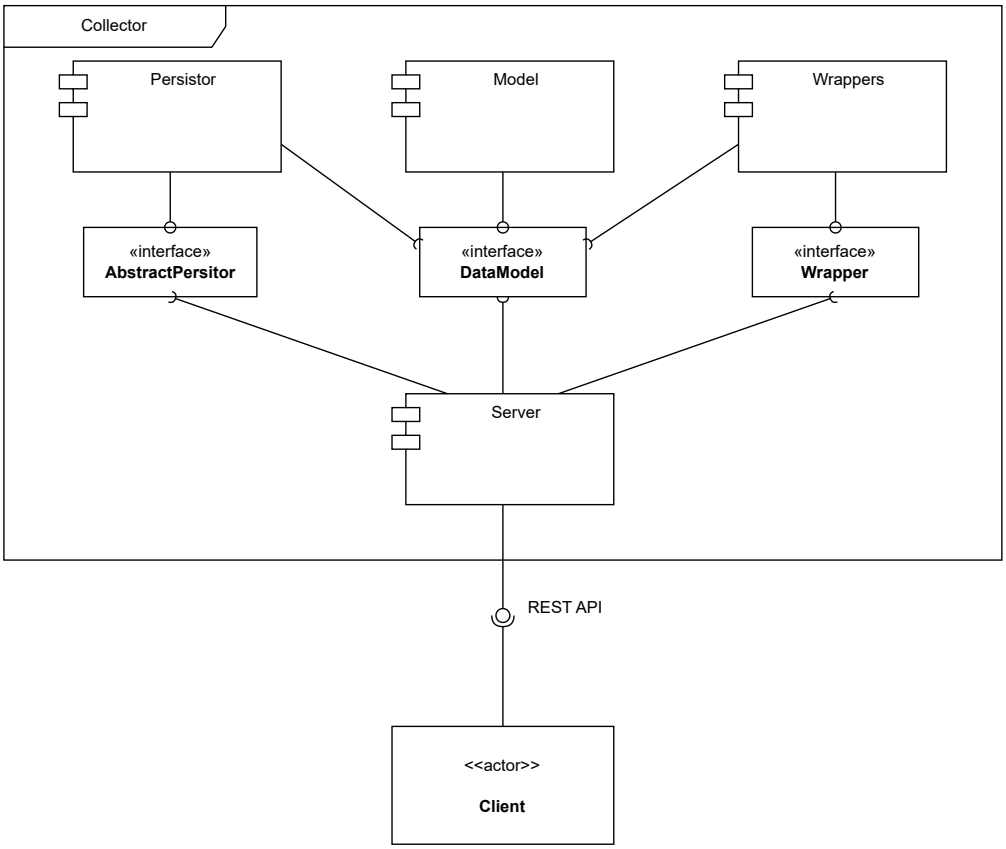
3. Návrhová dokumentace

V této kapitole se budeme věnovat návrhu celého programu. Představíme si architekturu modulů, použité návrhové vzory a pobavíme se o použitých technologiích.

3.1 High level architektura

Nejprve se podíváme na jednotlivé moduly Sběratele a představíme si jejich hlavní úkoly, které zastávají. Zároveň popíšeme rozhraní, které používají ke komunikaci.

Na obrázku 3.1 je ilustrován diagram modulů, podle kterého budeme postupovat a podrobněji dané aspekty rozebereme. Z diagramu je vidět, jak jsou ne sobě jednotlivé moduly závislé. Vnější komponenta s názvem *Collector* reprezentuje vlastní aplikaci. Uvnitř této komponenty jsou následně zobrazeny jednotlivé moduly a názvy rozhraní, které používají pro komunikaci mezi sebou. Zároveň je je zde přidaný klient, který komunikuje s aplikací pomocí server modulu.



Obrázek 3.1: Architektura modulů

Server Jelikož jsme téma serveru započali již v předchozím odstavci, budeme se v první řadě věnovat tomuto modulu. Mezi hlavní odpovědnosti tohoto modulu patří komunikace s klientem, který posílá pomocí REST rozhraní své požadavky na vyhodnocení databázových dotazů. Server pak deleguje tuto odpovědnost na zbylé moduly, které zprostředkovávají vykonání jednotlivých operací potřebných pro zpracování výsledku a vytvoření a uložení datového záznamu. Vyžaduje rozhraní pro komunikaci od všech zbývajících modulů aplikace.

Model Tento modul reprezentuje datový model celé aplikace. Poskytuje rozhraní v podobě veřejného rozhraní `DataModel`, který je podrobněji popsán v sekci 4.3. Umožňuje ostatním modulům pracovat s datovým modelem aplikace a zároveň kontroluje správnost a konzistentnost vkládaných dat. Nabízí také rozhraní pro čtení některých již nasbíraných parametrů, které je pak možné používat pro další výpočty. Nakonec umožňuje pomocí nástroje jackson[1], který je podrobněji popsán v sekci 3.3, sám sebe serializovat do formátu JSON. To se využívá pro účely logování případně pro ukládání do dokumentové databáze.

Wrappers V tomto modulu jsou implementace všech *wrapperů*. Jde o jednu z hlavních částí programu, kde se vytvoří instance datového modelu pro dotaz a nasbírají se do něj všechny hledané parametry. Potřebuje mít přístup k rozhraní datového modelu, aby do něj mohl zapisovat nalezené informace. Modul poskytuje rozhraní v podobě veřejného rozhraní `AbstractWrapper` pro účely spouštění dotazů z modulu server. Tento interface následně musí implementovat každý z wrapperů, které jsou součástí modulu. Rozhraní je oproti tomu v datovém modelu značně jednoduší, obsahuje pouze jedinou veřejnou metodu.

Veřejné metody rozhraní `AbstractWrapper` jsou následující:

- `executeQuery(String query)` – spustí zadaný dotaz `query`, vytvoří instanci datového modelu pro dotaz, naplní jej sesbíranými metadaty a vrátí instanci třídy `DataModel`. Nastane-li libovolný problém, vyhazuje výjimku `WrappersException`, která většinou obaluje skutečnou příčinu v podobě další výjimky.

Persistor Dalším modulem je *persistor*. Stará se o ukládání sesbíraných dat. Ačkoliv má přístup k celému rozhraní datového modelu, hlavně používá jeho metodu `toJson` pro serializaci a následné uložení. Poskytuje serveru rozhraní pro ukládání sesbíraných dat v podobě veřejného rozhraní `AbstractPersistor`. Server tedy po té co spustí wrapper se sběrem parametrů, tak uloží výsledek do persistoru.

Veřejné metody rozhraní `AbstractPersistor` jsou následující:

- `saveExecution(String uuid, DataModel model)` – uloží výsledek exekuce pro dané `uuid`. Pokud nastane nějaký problém, tak vyhazuje výjimku `PersistorException`.

- `saveExecutionError(String uuid, String errMsg)` – metoda uloží hlášení o chybě, se kterým skončila exekuce s identifikátorem `uuid`. Pokud nastane nějaký problém, tak vyhazuje výjimku `PersistorException`.

[1] https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind

- `getExecutionResult(String uuid)` – vrátí uložený výsledek dotazu identifikovantelného podle hodnoty `uuid`.
- `getExecutionStatus(String uuid)` – vrátí stav exekuce pro exekuci s daným `uuid`. Pokud nastane nějaký problém, tak vyhazuje výjimku `PersistorException`.

3.2 Návrhové vzory

Nepřekvapivě podle názvu, se budeme v této sekci zabývat návrhovými vzory, které použijeme pří implementaci a návrhu tohoto softwarového díla. V jednotlivých částech si shrneme motivaci a důvody použití konkrétního vzoru a zlepšení, kterého použitím vzoru docílíme, a pokusíme se tak velmi stručně shrnout základní vlastnosti použitých návrhových vzorů.

3.2.1 Builder

Libovolný výsledek dotazu z databáze bývá většinou datová množina, přes kterou je nutné kvůli jejímu zpracování iterovat. Abychom sjednotili jednotlivé typy výsledků z různých databázových systémů, program obsahuje nově definované typy `CachedResult` a `ConsumedResult`. Všechny původní výsledky se tak do těchto nově vytvořených tříd mapují. Většinou je nutné a zároveň i žádoucí výsledky zpracovávat při mapování iterativně. Jelikož je výhodou, aby nově zinicializované výsledky typu `CachedResult` a `ConsumedResult` mohly být pro další zpracování nemodifikovatelné, je velmi výhodné pro jejich vytváření použít návrhový vzor *Builder*. Získáme tím možnost výsledky zpracovávat postupně a nově namapované výsledky používat pouze ke čtení již nasbíraných dat, které v dalších činnost již nelze modifikovat. V kódové bázi jsou tedy typy `CachedResultBuilder` a `ConsumedResultBuilder`, které implementují tento návrhový vzor. Po zavolání metody `ToResult` pak inicializují objekt příslušného typu výsledku s nasbíranými daty z původních výsledků z libovolného databázového systému. Objekty typu `ChachedResult` nebo `ConsumedResult` tak následně mohou sloužit pouze ke čtení, což má hlavní přínos v lepší čitelnosti a přehlednosti kódu.

3.2.2 Abstract Factory

Dalším z tvořivých návrhových vzorů, které si zde představíme, je *Abstract factory*. Vzhledem k úzké provázanosti návrhového vzoru *Factory method* s *Abstract factory*, bychom také mohli uvažovat, že tento program implementuje i návrhový vzor *Factory method*. Jelikož používaním vzoru *Abstract factory* je použití vzoru *Factory method* vlastně důsledkem tohoto výběru, tak zde tato skutečnost není explicitně zmíněna a rozebrána.

Na různých místech v programu je nutné vytvářet různé výjimky, které dobře popisují špatné jevy nastalé v běhu programu. Každý wrapper může mít své vlastní specifické problémy, které mohou nastat při exekuci dotazu. Vzniká zde tedy potřeba efektivním způsobem centralizovat logiku vytváření výjimek s možností přizpůsobení této vlastnosti wrapperům pro konkrétní databázové systémy. Pro tento účel existuje v programu třída `ExceptionsFactory`, která má toto

vytváření výjimek na starosti. Obsahuje implementaci pro vytváření základních výjimek, které jsou společné pro více wrapperů. Tuto třídu je dále možné dědit a přepisovat její chování pro inicializaci společných výjimek, pokud je to nutné. Také je možné toto chování i dále rozšiřovat a vytvářet tak výjimky pro nové problematické případy. Proces vytváření výjimek je tedy sjednocený napříč celým programem. `ExceptionsFactory` tedy implementuje návrhový vzor *Abstract factory*. Jediný rozdíl oproti doporučení jak tento návrhový vzor implementovat je, že `ExceptionsFactory` obsahuje základní implementace většiny svých *factory method*. Hlavním důvodem je ulehčení práce při implementaci nových wrapperů, kde je tak možné přepsat pouze případy, které jsou oproti základnímu řešení něčím specifické.

3.2.3 Template Method

Algoritmus pro exekuci zadaného vstupního dotazu je nezávisle na různých databázových systémech velmi podobný. Vždy je potřeba navázat nějaké databázové spojení, vykonat zadaný dotaz nad databázovým systémem, získat pro tento dotaz i plán jeho exekuce a nakonec uložit nasbíraná data do nově vytvořené instance datového modelu. Algoritmus jako takový by nemělo být potřeba v budoucnosti zásadním způsobem měnit. Jedním z behaviorálních návrhových vzorů, který se velmi hodí pro implementaci tohoto problému je vzor *Template method*. Abstraktní wrapper, který je nutné implementovat pro vytvoření vlastního wrapperu, obsahuje metodu `executeQuery`, která se z modulu serveru spouští pro vykonání sběru dat zadaného dotazu. Tuto metodu není možné v potomcích přepisovat. V případě potřeby je možné přidat nové *hook* metody, které umožní rozšíření algoritmu o další kroky, které mohou být v budoucnu potřebné pro implementaci nových wrapperů. Prvním krokem algoritmu pro exekuci dotazu je vytvoření instance třídy `ExecutionContext`. Všechny další metody, které reprezentují jednotlivé kroky algoritmu a které je nutné v potomcích přepsat, následně očekávají `ExecutionContext` jako argument pro volání. Tento kontext je možné podědit a implementovat dle libosti a potřeby nově vznikajících wrapperů. Umožňuje to předávat libovolná data mezi jednotlivými kroky v algoritmu.

3.2.4 Facade

Modul datového modelu je implementován pomocí hierarchické struktury, kde je potřeba dynamicky vytvářet nové uzly podle komplexnosti zadaného dotazu. Kořen této struktury reprezentuje společná data nasbíraná pro celou množinu datových sad. Na nižších úrovních má pak každá relevantní tabulka uložená data v instanci právě jednoho uzlu. Analogicky jsou následně ukládané informace o sloupcích a typech. Jelikož určitě nechceme tuto problematiku řešit kdykoli budeme implementovat nový wrapper, nabízí se možnost tuto komplexitu schovat pod jednotné rozhraní. Necháme tak datový model, aby se sám staral o správnost svých dat a odpovídající konzistentnost vlastní struktury. Wrapperům tedy nabídneme jednotné rozhraní, které skrývá komplexitu datového modelu a nabízí jednotný bod pro komunikaci s modelem. Jelikož kořen hierarchie ani žádný další z jejích uzlů neví o existenci rozhraní `DataModel` ani o jeho implementaci `QueryDataModel`, jedná se o implementaci návrhového vzoru *Facade*.

3.2.5 Adapter

Každý databázový systém disponuje jiným driverem pro ovládání a komunikaci. Jedním z hlavních cílů tohoto programu, bylo zajištění modularity pro snazší rozšiřitelnost. Nad každým nativním ovladačem (angl. *driver*) pro databázi je nutné pro její integraci do programu vytvořit nový wrapper, který umožní programu s tímto databázovým systémem fungovat. Každý wrapper je tak implementací návrhového vzoru *Adapter*, neboť přizpůsobuje rozhraní nativních driverů našemu rozhraní pro abstraktní wrapper. Konkrétně se jedná o implementaci *object adapteru*, neboť každý wrapper ať už přímo či nepřímo udržuje referenci na nativní driver pro daný systém a implementuje pomocí něj svou funkcionalitu.

3.3 Použité technologie

Ukážeme si, jaké technologie budou použity pro implementaci jednotlivých modulů aplikace a pokusíme se objasnit výhody získané použitím daných technologií. Každému z modulů věnujeme na toto téma samostatnou část v rámci této sekce.

3.3.1 Java Database Connectivity (JDBC)

Nejprve si představíme technologii JDBC. Nejedná se přímo o driver pro žádnou databázi, ale o rozhraní pro práci s databázemi. Je součástí standardní edice jazyka *Java*[2], a k jejímu použití tak není potřebné definovat žádné explicitní závislosti. Přesto mi přijde dobré se o něm v tomto kontextu pobavit.

Protože se jedná o rozhraní bez konkrétní implementace, pro správné fungování je nutné využít konkrétní řadič. Ten je nutné nainstalovat a přidat do projektu jako další závislost. V kódu se pak tento driver pro vytvoření instance databázového připojení zaregistruje pomocí třídy `DriverManagaer`. Následně umožňuje rozhraní spouštět nad navázaným spojením různé SQL dotazy a následně zpracovávat jejich výsledky.

Teoreticky je tak možné využít toto API i pro ovládání nerelačních databází, ovšem je potřebné mít k tomu odpovídající *driver*. Takové drivery existují i pro některé NoSQL databáze, většinou ovšem mívají z technologických důvodů nějakým způsobem omezenou podporu některých funkcí, které jsou charakteristické pro datový model rozdílný od relačního.

3.3.2 PostgreSQL Driver

Jak již z názvu vyplývá, jedná se o řadič pro systém PostgreSQL.[6] Konkrétně jsem použil implementaci doporučenou výrobcem z oficiálních stránek dokumentace pro databázi. Ke komunikaci s databází je v programu využité výše popsané rozhraní JDBC. Driver se tak pouze zaregistruje pomocí třídy `DriverManageru`, čímž se naváže spojení s databází. S tímto spojením pak program po celou dobu životnosti pracuje. Řadič je kompletně psaný v jazyce Java a ke komunikaci s databází využívá její nativní síťový protokol. Nástroj jsem využil k implementaci

[2] https://docs.oracle.com/en/java/

wrapperu pro databázi PostgreSQL. Jedná se o software s kvalitní dokumentací a dobrou podporou.

3.3.3 Neo4j Driver

Dalším ze seznamu je řadič pro grafovou databázi Neo4j.[7] Jde o oficiální nástroj pro komunikaci s databází z Java aplikace. Software je vyvíjen v jazyce Java a s databází komunikuje pomocí Bolt protokolu. Bolt protokol je v mnoha věcech podobný HTTP, ale mezi hlavní rozdíly patří především fakt, že se řadí mezi stavové protokoly. Jde o protokol speciálně navržený pro komunikaci s databázemi. Obecně má Java okolo databáze Neo4j vytvořený pěkný ekosystém, který je vyvíjen společností Neo4j a je neustále rozšiřován o nové nástroje. Písmeno '`j`' v názvu databáze totiž znamená Java a společnost se tak snaží tomuto názvu tímto způsobem dostát. Driver umožňuje navázat spojení s databází a následně se dotazovat pomocí jazyka *Cypher*. Perfektně se tak nástroj hodil pro implementaci wrapperu pro databázi Neo4j.

3.3.4 MongoDB Driver

Tento řadič je kromě implementování příslušného wrapperu také použit pro implementaci *perzistoru*, což je modul zodpovědný za ukládání nasbíraných výsledků. Celý řadič je psán v Javě a jedná se o nativní nástroj pro komunikaci s databází MongoDB.[8] Když se dostaneme na stránky s dokumentací a budeme se zajímat o nástroje pro práci s Javou, zjistíme, že existují hned dvě varianty řadičů. Vybral jsem řadič pro synchronní komunikaci, který nám dává jistotu, že dotazy se vyhodnocují v pořadí, v jakém jsou volány ze zdrojového kódu. Některé výpočty dalších statistik jsou na tomto faktu závislé. Každá instance MongoDB wrapperu udržuje vlastní `MongoClient` objekt, neboť potřebujeme, aby každý wrapper mohl být připojený na svou vlastní databázi. Exekuce každého uživatelem zadaného dotazu pak běží paralelně, takže nevýhoda v podobě nižší propustnosti dotazů při použití synchronního řadiče se tímto alespoň částečně odbourá. Asynchronní řadič se nazývá *Reactive Streams* a nabízí API pro zpracovávání výsledků pomocí tzv. *streamů*.

3.3.5 H2 Database

Na začátku sekce jsme se bavili o technologii JDBC a hned tuto znalost opět využijeme. JDBC rozhraní je totiž využito pro komunikaci i s H2.[3] Systém H2 není klasickou databází, jak by člověk zpočátku předpokládal, ale jedná se o *in-memory databázi*. Jde tak o relační databázi, která ukládá data pouze do paměti RAM, což znamená, že data nejsou ukládána dlouhodobě. Aplikace používá tuto databázi pro implementaci asynchronní fronty pro odbavování požadavků na vyhodnocení dotazů zadaných uživatelem. Databáze podporuje používání dotazovacího jazyka SQL. Autorem je Thomas Mueller a celá databáze je psaná v jazyce Java. Jedná se o velmi lehkou implementaci, která zabírá jen řádově několik mega-bytů paměti.

[3] https://www.h2database.com/html/main.html

3.3.6 Spring boot

Jedná se o technologii, která byla doporučena již zadáním práce a jedná se tak o jeden z nefunkčních požadavků. Aplikace pro své vhodné fungování potřebuje fungovat jako server s vhodným REST rozhraním. Jelikož bychom nejspíš nechtěli implementovat vlastní komunikaci přes patřičné síťové protokoly, používáme pro tento účel frameworku Spring Boot.[4] Existuje alternativa v podobě čistého frameworku Spring.[5] Oba frameworky jsou v jádru prakticky stejné, jediný velký rozdíl a zároveň důvod, proč jsme použili Spring Boot, je nutnost konfigurace čistého Springu. Spring Boot je již přednastavený a funguje tedy s minimální nutností konfigurace, což v konečném důsledku značně zjednodušuje práci.

3.3.7 Jackson

Poslední nástroj, o kterém se zde zmíníme, je knihovna Jackson pro serializaci do datových formátů JSON a XML. V našem případě používáme tuto knihovnu hlavně pro účely serializace do formátu JSON. Konkrétně je použita k serializaci instance datového modelu, abychom jej mohli uložit do persistoru v podobě hierarchického dokumentu. Kromě serializace používáme knihovnu i k deserializaci. Součástí sběru dat pro libovolný vstupní dotaz je získání jeho exekučního plánu. V případě PostgreSQL tyto plány pro exekuci dotazů získáváme ve formátu JSON a potřebujeme je vhodně deserializovat, abychom byli schopni plány projít a získat z nich relevantní metadata pro Sběratele.

[4] https://docs.spring.io/spring-boot/index.html
[5] https://docs.spring.io/spring-framework/reference/index.html

4. Programátorská dokumentace

V kapitole se budeme bavit o technické dokumentaci programu. Určitě je vhodné si kapitolu prostudovat, pokud uvažujeme o rozšiřování Sběratele pro podporu nových databázových systémů. Nejprve shrneme v jednotlivých modulech diagramy tříd a jejich důležité veřejné metody. Dále pro vybrané případy užití popsané v sekci 2.3 se podíváme na sekvenční diagramy a přiblížíme tím čtenáři představu o tom, jak se provolávají mezi sebou jednotlivé třídy. Ukážeme si API pro rozšíření Sběratele o podporu pro nové databázové systémy a nakonec se budeme věnovat problémům, které jsem při práci a integraci nových databázových systémů byl nucen řešit. V úplném závěru se pak stručně podíváme na problematiku testování.

Zdrojové kódy aplikace jsou k dispozici na mém GitHub repozitáři[1].

4.1 Diagram tříd

V této sekci si postupně rozebereme koncept implementovaných tříd v jednotlivých modulech aplikace. Diagram tříd je vytvořený pro každý modul zvlášť, neboť tříd je v programu velký počet a ani v této sekci nebudeme rozebírat všechny třídy, které se ve zdrojovém kódu vyskytují. Pokusíme se shrnout především ty nejdůležitější části programu.

4.1.1 Server

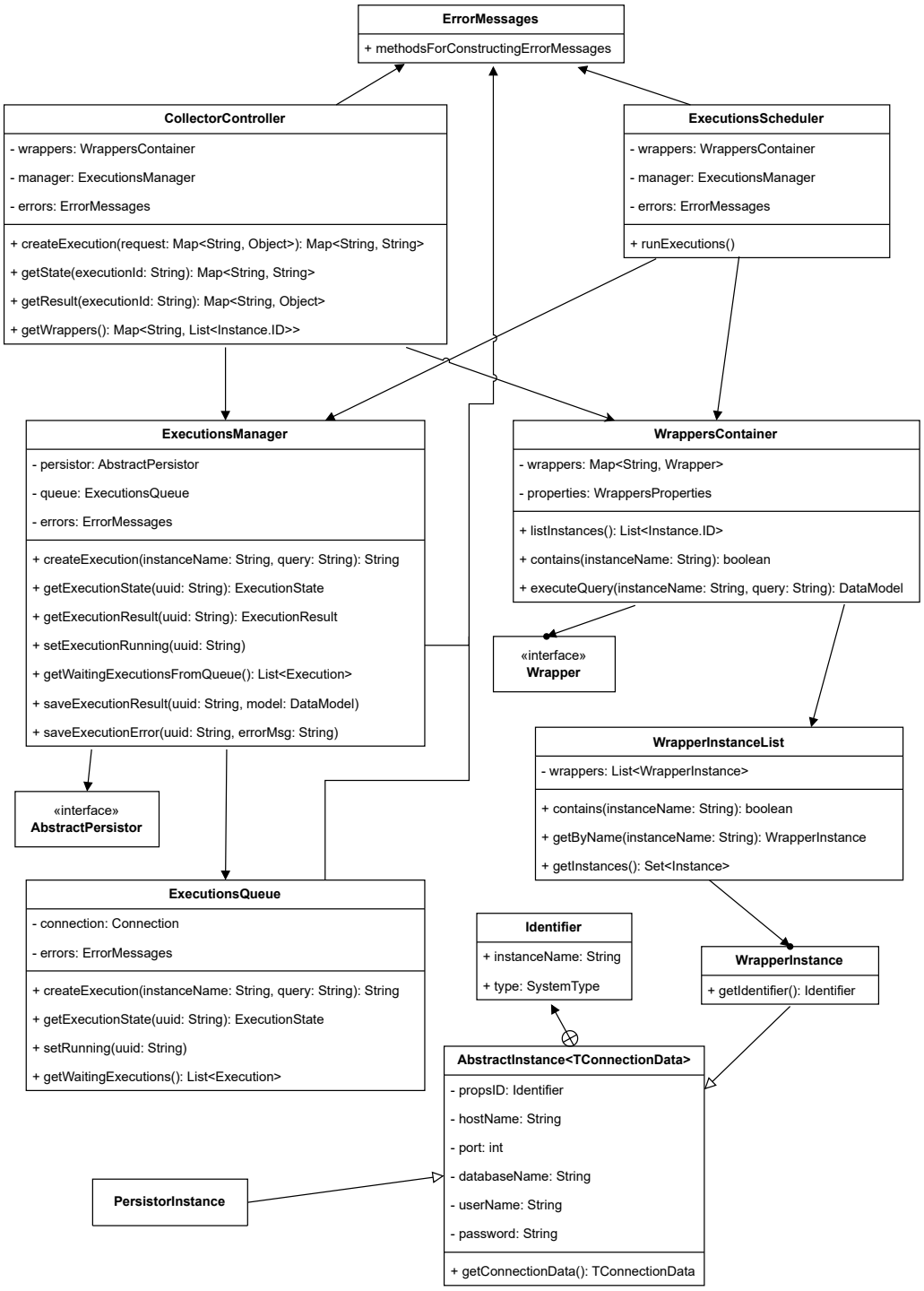
Začneme modulem nazvaným *Server*. Postupně si tak představíme jednotlivé prvky fungování samotné fronty pro zpracování exekucí a následně i komponenty, které umožňují načtení informací pro připojení jednotlivých wrapperů ze souboru `application.properties`. Pro lepší přehlednost mezi vztahy jednotlivých tříd v modulu a jejich základních vlastností je níže k dispozici diagram 4.1, který tyto skutečnosti graficky znázorňuje.

SystemType. Jedná se o jednoduchý výčtový typ, který svými hodnotami diverzifikuje jednotlivé typy systémů. Aktuálně nabývá hodnot `MongoDB`, `Neo4j` a `PostgreSQL` podle aktuálně podporovaných systémů na straně wrapperů. Parsuje se podle něj typ systému při načítání dat ze souboru `application.properties`. Poznamenejme, že pro větší přehlednost diagramu 4.1 není jeho součástí, neboť se jedná o jednoduchý typ, který nemá žádné komplexní vztahy s jinými třídami modulu.

ExecutionState. Jedná se o výčtový typ, který z obdobných důvodů jako ten předešlý není součástí diagramu. U všech exekucí se nepřímo eviduje, jaký mají stav, podle toho v jaké fázi zpracování se zrovna nachází. Tuto skutečnost reflektují hodnoty tohoto výčtového typu, který je vracen jako výsledek při zkoumání stavu exekuce:

- `Waiting` – exekuce ve frontě, které ještě nebyly spuštěné plánovačem.

[1] https://github.com/composerPigeon/Collector



Obrázek 4.1: Server diagram tříd

- `Running` – exekuce ve frontě, které již byly spuštěné plánovačem, ale zatím nedoběhly.
- `Processed` – exekuce již zpracované, mají výsledek již uložený v persistoru.
- `NotFound` – exekuci se nepodařilo dohledat.

Execution. Jednoduchá třída, která reprezentuje záznam jedné exekuce. Je poslední třídou, jenž není součástí diagramu, o které se budeme bavit. Není odkazována jinými třídami, pouze je použitá pro návratový typ funkce získávající seznam exekucí čekajících na spuštění plánovačem. Jelikož fronta je implementována pomocí *in-memory databáze*, odpovídá tento typ jednomu záznamu z tabulky exekucí, z něhož se fronta skládá. Ukládanými atributy exekuce tedy jsou:

- `String uuid` – umělě vytvořený identifikátor při zařazení do fronty.
- `TimeStamp added` – datum zařazení exekuce (podle tohoto data se nejprve zpracovávají ty nejdéle zařazené).
- `boolean isRunning` – flag indikující, jestli již byla exekuce spuštěna plánovačem či nikoliv.
- `String instanceName` – název instance, nad kterou se má exekuce spustit.
- `Sttring query` – dotaz zadaný uživatelem pro vyhodnocení.

ErrorMessages. Z diagramu 4.1 je patrné, že na třídu `ErrorMessages` udržuje referenci velké množství tříd z modulu. Třída je anotována pomocí `Component`, což z ní dělá tzv. *Bean* komponentu frameworku Spring Boot. Jedna z hlavních výhod, kterou toto přináší, je možnost využití *Dependency Injection*. Třída je tedy automaticky poskytována dalším komponentám, které se na ní referencují. Tento přístup bezesporu zlepšuje čitelnost kódu, neboť se nemusíme zabývat korektní inicializací a víme, že třída spolehlivě funguje i ve více vláknovém prostředí jako tzv. *Singleton*. Jak již napovídá samotný název třídy, definuje velké množství veřejných metod sloužících k vytváření srozumitelných chybových hlášek pro různé problémy. Nebudeme zde tedy procházet jednotlivé metody, pokud se čtenář zajímá o bližší informace ohledně chybových hlášek a jejich možných řešení, pak je toto téma probíráno v sekci 5.6.

CollectorController. Třída je také komponentou frameworku Spring a je tedy patřičně anotována. Zároveň je `FrontControllerem` aplikace. Obsahuje tedy veřejné metody, které jsou mapované na jednotlivé REST požadavky od uživatelů, jenž budou systém používat. Metody jsou tedy velmi podobné REST rozhraní aplikace a mají až na pár drobností velice podobnou signaturu. Z tohoto důvodu je v této sekci nebudeme podrobně rozebírat. V případě zájmu se podrobnému popisu rozhraní věnujeme v sekci 5.4.

Třída udržuje zároveň reference na instance tříd `ExecutionsManager` a `WrappersContainer`, jelikož na ně deleguje implementované požadavky uživatelé. Jelikož kdykoliv během zpracování požadavků může nastat libovolná chyba, udržuje třída také referenci na výše popsanou instanci třídy `ErrorMessages`, pomocí které

vytváří vhodné chybové hlášky a následně se snaží upozornit uživatele na vzniklé problémy vhodnými odpověďmi na zadané požadavky.

ExecutionsScheduler. Jak již plyne z názvu třídy, jedná se o plánovač spouštění jednotlivých exekucí. Veškerá funkcionalita je programována pomocí frameworku Spring. Třída je klasickou komponentou. Má tedy na sobě anotaci `Component`. Kromě třídy je ovšem anotována i její jediná veřejná metoda `runExecutions` pomocí `Scheduled`, kde parametr `fixedRate` v milisekundách určuje, jak často se metoda pravidelně spouští. Tato metoda pak pouze spustí nad wrappery všechny aktuálně čekající exekuce ve frontě. Toto se děje v nastaveném intervalu pěti vteřin, který je možné v případě potřeby upravit.

Třída obsahuje kromě `ErrorMessages` reference na instance `ExecutionsManager`, `WrappersContainer`. S jejich pomocí pak vykonává popsané spouštění exekucí.

ExecutionsManager. Třída `ExecutionsManager` má na starosti správu životnosti všech exekucí. Řeší tedy přepínání mezi jednotlivými exekučními stavy, zařazení exekucí do fronty a také získávání výsledků. Všechny tyto operace zprostředkovává pomocí tříd `ExecutionsQueue` a `AbstractPersistor`, na které udržuje referenci a deleguje na ně jednotlivé operace. Třída má tedy několik veřejných metod, jež odpovídají těmto operacím:

- `createExecution` – metoda která na vstupu očekává název instance a dotaz, pro které vytvoří novou exekuci a vrátí její identifikátor.
- `getExecutionState` – na vstupu očekává identifikátor exekuce, následně pro něj vrátí její aktuální stav. Nejprve se kouká do fronty a pokud zde exekuce není, pak se jí snaží najít v persistoru.
- `setExecutionRunning` – na vstupu očekává identifikátor exekuce, kterou pak ve frontě nastaví na již běžící. Metodu používá plánovač poté co spustí exekuci.
- `getExecutionResult` – pro identifikátor exekuce vrátí její výsledek, pokud byla exekuce již ve stavu `Processed`, v opačném případě končí chybou nenalezeno.
- `getWaitingExecutionsFromQueue` – vrátí z fronty kolekci exekucí ve stavu `Waiting`
- `removeExecutionFromQueue` – smaže z fronty exekuci s daným identifikátorem. Když doběhne vyhodnocení dotazu exekuce, pak se její záznam maže z fronty.
- `saveExecutionResult` – pro zadaný identifikátor a instanci `DataModel` s nasbíranými daty uloží informace do persistoru.
- `saveExecutionError` – pro identifikátor exekuce a chybovou hlášku uloží tyto informace do persistoru jako chybu při zpracování exekuce.

Při inicializaci očekává několik parametrů, kde by chtělo zmínit především instanci třídy `PersistorInstance` a `Initializers`. Oba parametry slouží k inicializaci persitoru při vytváření komponenty. Třídě `PersistorInstance` se budeme věnovat níže, a proto ji nyní přeskočíme. Měli bychom si tedy alespoň krátce něco povědět o třídě `Initializers`. Slouží jako mapa pro udržování konstruktorů pro jednotlivé typy systémů a tuto funkci plní aktuálně hlavně pro wrappery. Pokud by do budoucna program disponoval implementacemi více persistorů, pak bude plnit stejnou funkci i pro ně. Umožňuje to tedy přepnout systém persistoru pouze přepsáním `application.properties` a restartováním aplikace. Pro více informací o třídě `Initializers` je vhodné se podívat na sekci 4.3, kde rozebereme, jak je nutné tuto třídu upravit pro registraci nových systémů. Částečně tak zde budeme řešit i principy jejího fungování, a proto v této části dokumentace není plně popsána.

ExecutionsQueue. Jedná se o vlastní implementaci celé fronty. Při inicializaci používá privátní vnořenu třídu `ConnectionBuilder`, která má za úkol korektně sestavit spojení pro databázi H2. Má částečně stejné rozhraní jako výše popsaná třída `ExecutionsManager`, neboť operace z ní jsou delegovány na samotnou frontu, pokud to situace vyžaduje. Konkrétně se jedná o metody:

- `createExecution` – pro zadaný název instance a dotaz vytvoří nový záznam exekuce a vloží jej do fronty (tj. do tabulky `executions`).
- `getExecutionState` – pro identifikátor exekuce, pokud se nachází v tabulce a má atribut `isRunning` nastaven na true, pak vrací, že je ve stavu `Running`. V opačném případě vrací stav `Waiting`, nebo stav `NotFound`, pokud záznam ve frontě chybí.
- `setRunning` – pro identifikátor exekuce nastaví atribut `isRunning`.
- `getWaitingExecutions` – vratí z tabulky executions všechny záznamy seřazené podle času vytvoření mající atribut `isRunning` nastavený na false.

AbstractInstance. Jedná se o abstraktní a generickou třídu s parametrem `TConnection`, která je předkem pro instance wrapperů a persistorů. Data pro připojení k jednotlivým wrapperům ale i k persistoru jsou si velmi podobná. Abychom se tedy nemuseli při implementaci opakovat, obsahuje třída definici všech atributů důležitých pro připojení k jednotlivým databázím. Zároveň má několik veřejných metod, které umožňují zisk informací na-parsovaných ze souboru `application.properties`.

- `getInstanceName` – vrací název instance.
- `getSystemType` – vrací název typu systému.
- `getConnectionData` – jedná se o abstraktní metodu, která vrací instanci generického typu `TConnectionData`, což je kontejner pro data k připojení k databázím, který se liší pro wrappery a persistor.

Identifier. Jedná se o jednoduchou vnořenou třídu ve výše popsané třídě `AbstractInstance`. Obsahuje dva atributy `systemType` a `instanceName`. Jak název napovídá, jedná se o identifikátor načtených instancí. Důležité jsou především u wrapperů a tento typ se používá hlavně pro výčet připojených instancí, kde uživateli vypisuje přesně tyto informace.

PersistorInstance. Jedná se o implementaci abstraktní třídy `AbstractIdentifier` určené pro parsování dat pro připojení pro persistor. Tato třída implementuje metodu `getConnectionData`, která vrací generický parametr třídy typu `AbstractPersitior.ConnectionData`. Parsování je realizováno pomocí frameworku Spring. Třída je tak pouze vhodně o-anotována, čímž se o samotnou implementaci parsování nemusíme prakticky starat. Je pouze potřeba dát si pozor na jednotlivé názvy datových položek v konfiguračním souboru a názvy parametrů v konstruktoru, které musí vzájemně odpovídat.

WrapperInstance. Podobně jako je `PersistorInstance` konkrétní implementací třídy `AbstractInstance`, tak v tomto případě se bavíme o implementaci stejné funkcionality pro wrappery. Oproti `PersistorInstance` disponuje navíc veřejnou metodou `getIdentifier`, která vrací identifikátor instance. Slouží pro výše popsaný scénář, kdy pro připojené instance chceme uživateli vypsat jejich výčet.

Je důležité poznamenat, abyste třídu `AbstractInstance` implementovali s parametrem `AbstractWrapper.ConnectionData`.

WrapperInstanceList. Jedná se o kolekci výše popsaných instancí pro wrappery. Obsahuje následující jednoduché veřejné metody pro práci s instancemi wrapperů:

- `contains` – pro název instance vrací boolean podle toho, zda se nachází v kolekci či nikoliv.
- `getByName` – vrací instanci pro zadaný název.
- `getInstances` – vrací všechny instance, které má v kolekci. Slouží hlavně pro enumeraci všech načtených instancí.

Tuto třídu jsem chtěl původně odstranit, jelikož její funkcionalita není výjimečná a podobných výsledků by člověk dosáhl i pomocí jednoduchého seznamu nebo množiny. Ale protože třída zastřešuje parsování dat pro připojení jednotlivých wrapperů, jedná se o důležitou třídu, ačkoliv se jedná o triviální implementaci kolekce. Je tedy popsána potřebnými anotacemi z frameworku Spring a jednotlivé wrappery jsou v konfiguračním souboru zapisovány jako prvky listu.

WrappersContainer. Pro načtené instance wrapperů ovšem potřebujeme následně samotné wrappery inicializovat, když je to potřebné, abychom je mohli používat v aplikaci. O tuto skutečnost se pomocí již zmíněné třídy `Initializers`, která již byla zmíněna výše a bližší podrobnosti se může čtenář dozvědět v sekci 4.3, stará právě kontejner pro wrappery. Spouštění vyhodnocování dotazů se tak děje pomocí této třídy, která má uloženou mapu všech načtených wrapperů pro názvy instancí. Veřejné metody pro `WrappersContainer` tak jsou následující:

- `listInstances` – získá seznam připojených instancí wrapperů.
- `contains` – zkontroluje existenci dané instance podle zadaného názvu.
- `executeQuery` – pro název instance a dotaz spustí nad příslušným wrapperem předaný dotaz.

4.1.2 Datový model

Nyní si představíme datový model, který je jedním z modulů programu a který je ilustrován na obrázku 4.2 pomocí diagramu tříd. Na začátku exekuce uživatelem zadaného dotazu se vytvoří nová instance třídy `QueryDataModel`. Při inicializaci se vytvoří i instance třídy `QueryData`, na níž `QueryDataModel` obsahuje referenci. Následně se takto vytvořený objekt pod veřejným rozhraním `DataModel` předává jednotlivým krokům algoritmu pro sběr metadat exekuce.

`DataModel` obsahuje převážně metody pro nastavení různých sbíraných parametrů a několik metod pro získání již načtených dat. Přístup k již uloženým parametrům pak slouží především pro další výpočty s již získanými daty. U některých systémů se jedná o jediný způsob, jak je některé informace vůbec možné zjistit.

`DataModel`, který prošel kompletním algoritmem pro exekuci dotazu a obsahuje sesbíraná data, se pak pomocí metody `toJson` serializuje do *persitoru*. `QueryDataModel` tedy implementuje i metodu `toJson` a serializaci do řetězce formátu json řeší pomocí knihovny Jackson. Tato knihovna je v programu použitá vícekrát a přesnější popis je obsažen v sekci 3.3 s názvem Použité technologie.

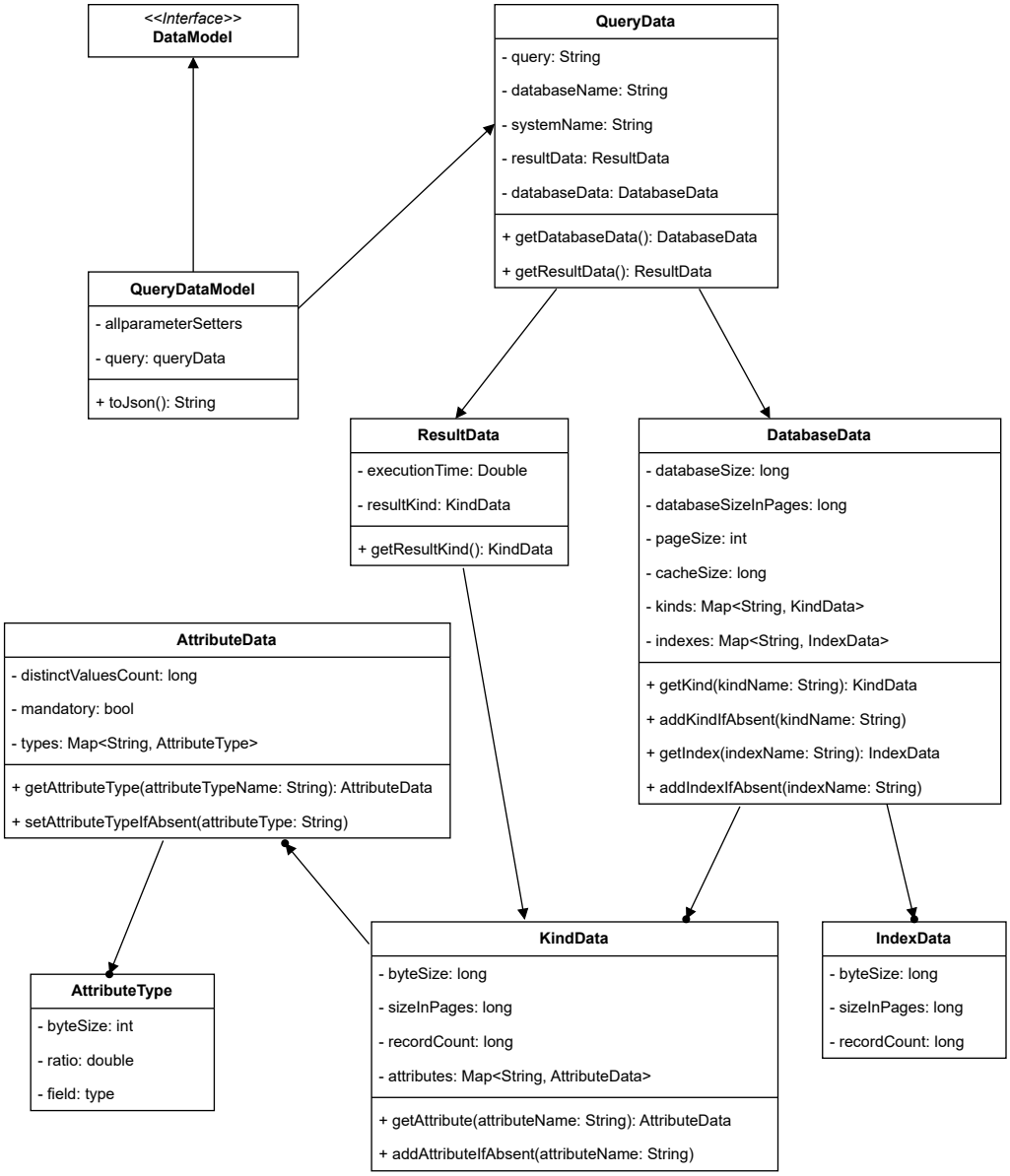
Na obrázku 4.2 vidíme, jak se na sebe navzájem odkazují jednotlivé třídy. Hlavní komponentou modulu je veřejné rozhraní `DataModel`, které definuje rozhraní pro ukládání všech nasbíraných parametrů během exekuce dotazu. Třída `QueryDataModel` následně implementuje toto rozhraní a zprostředkovává přístup do stromové hierarchie, jejíž kořenem je instance třídy `QueryData`. Třída `QueryDataModel` pak udržuje referenci na kořen této stromové struktury. Třídy, které reprezentují jednotlivé uzly hierarchie, nemusí být a ani nejsou dostupné z jiných modulů aplikace.

QueryData. Tato třída je kořenem hierarchie ukládající data a obsahuje metadata o samotné exekuci spuštěné uživatelem. Interně se skládá z těchto datových položek:

- `query` – ukládá dotaz, který uživatel zadal na vstupu.
- `databaseName` – ukládá název databáze, nad kterou je dotaz spouštěn.
- `systemName` – ukládá název databázového systému, nad kterým je vykonáván dotaz.

A implementuje metody:

- Obsahuje metody pro přístup k výše popsaným datovým položkám, tzv. *gettery* a *settery*.
- `getDatabaseData` – získá uzel reprezentující sbírané parametry pro databázi.



Obrázek 4.2: Modul datového modelu

- `getResultData` – získá uzel reprezentující parametry výsledku.

DatabaseData. Reprezentuje sbírané parametry pro databázi. Interně se třída skládá z těchto datových položek:

- `databaseByteSize` – reprezentuje velikost databáze v bytech.
- `databaseByteSizeInPages` – velikost databáze v počtu stránek.
- `pageSize` – reprezentuje zjištěnou velikost stránky pro daný systém.
- `cacheSize` – velikost paměti, kterou si databáze vyhrazuje pro účely kešování dotazů.

A implementuje metody:

- Obsahuje metody pro přístup k výše popsaným datovým položkám, tzv. *gettery* a *settery*.
- `getKind` – získá kind příslušného názvu. Pokud neexistuje, pak skončí s výjimkou `DataModelException`.
- `addKindIfAbsent` – vytvoří uzel pro nový kind, pokud neexistuje.
- `getIndex` – získá index příslušného názvu. Pokud neexistuje, pak skončí výjimkou `DataModelException`.
- `addIndexIfAbsent` – vytvoří uzel pro nový index, pokud neexistuje.
- `getKindNames` – vrátí množinu názvů všech kindů, které obsahuje.
- `getIndexNames` – vrátí množinu názvů všech indexů, které obsahuje.
- `getPageSize` – získá uloženou velikost stránky.

ResultData. Reprezentuje metadata nasbíraná o výsledku dotazu. Interně se třída skládá z těchto datových položek:

- `executionTime` – ukládá čas vyhodnocení a vrácení dotazu v milisekundách.

A implementuje metody:

- `setExecutionTime` – nastavení času vykonání dotazu.
- `getResultKind` – vrátí uzel zodpovědný za kind pro výsledek.

KindData. Jedna instance této třídy uchovává metadata o jedné datové sadě podobných entit. Jedná se například o tabulku v relačním modelu, kolekci dokumentů pro hierarchický model a kolekci entit se stejným popiskem pro grafový model. Interně se třída skládá z těchto datových položek:

- `byteSize` – ukládá velikost datové sady v bytech.
- `sizeInPages` – ukládá velikost datové sady v počtu stránek.
- `recordCount` – ukládá počet záznamů pro danou datovou sadu.
- `constraintCount` – ukládá počet deklarovaných omezení nad danou datovou sadou.

A implementuje metody:

- Obsahuje metody pro přístup k výše popsaným datovým položkám, tzv. *gettery* a *settery*.
- `getAttributeData` – vrátí uzel pro příslušný atribut. Pokud neexituje, pak vyhazuje výjimku `DataModelException`.
- `addAttributeIfAbsent` – vytvoří nový uzel pro atribut, pokud neexistuje.

IndexData. Jedna instance této třídy uchovává metadata o právě jednom indexu, který byl pro vyhodnocení dotazu na straně databáze použit. Interně se třída skládá z těchto datových položek:

- `byteSize` – ukládá velikost indexu v bytech.
- `sizeInPages` – ukládá velikost v počtu stránek.
- `recordCount` – ukládá počet záznamů v indexu.

A implementuje metody:

- Obsahuje metody pro přístup k výše popsaným datovým položkám, tzv. *gettery* a *settery*.

AttributeData Jedna instance této této třídy ukládá metadata pro právě jeden atribut z datové sady. Interně se třída skládá z těchto datových položek:

- `distinctValuesCount` – ukládá počet unikátních hodnot pro daný atribut.
- `mandatory` – reprezentuje informaci, zda se jedná o povinný atribut datové sady.

A implementuje metody:

- Obsahuje metody pro přístup k výše popsaným datovým položkám, tzv. *gettery* a *settery*.
- `getMaxByteSize` – projde typy atributů a vrátí největší velikost v bytech.
- `getAttributeType` vrací uzel reprezentující konkrétní typ atributu. Pokud takový neexistuje pak končí výjimkou. `DataModelException`
- `addAttributeTypeIfAbsent` – vytvoří nový uzel pro typ, pokud neexistuje.

AttributeType Jedna instance třídy reprezentuje metadata pro právě jeden datový typ pro konkrétní atribut. Každý atribut má alespoň jeden takový typ, ale může jich být i více pokud to dotazovaný systém podporuje. Interně se třída skládá z těchto datových položek:

- `byteSize` – velikost v bytech pro konkrétní datový typ.
- `ratio` – s hodnotami z intervalu $I = (0, 1]$ reprezentuje relativní četnost tohoto datového typu pro atribut.

A implementuje metody:

- Obsahuje metody pro přístup k výše popsaným datovým položkám, tzv. *gettery* a *settery*.

4.1.3 Wrappers

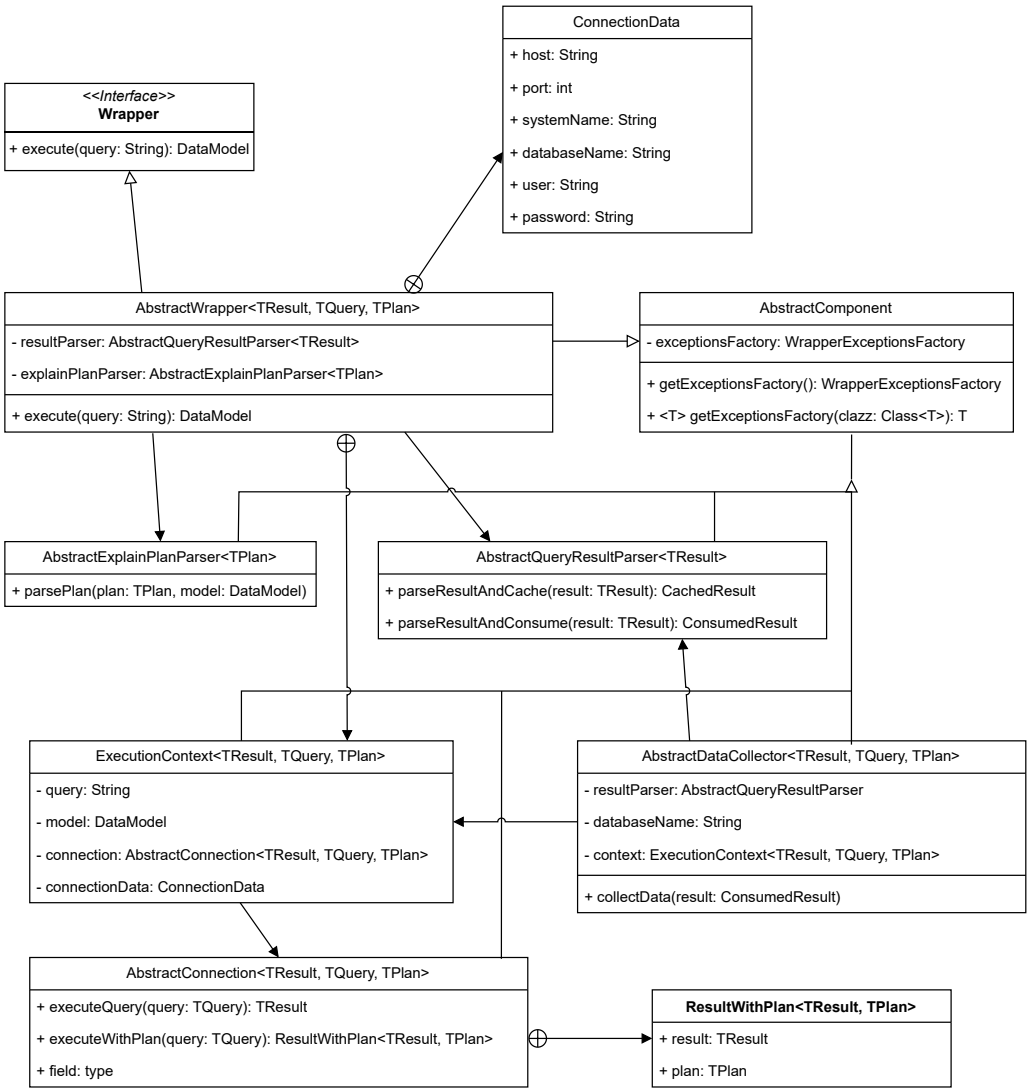
Jde o jeden z nejkomplikovanějších modulů celé aplikace. Nebudeme si tak představovat všechny třídy, které jsou jeho součástí. Naopak se podrobně podíváme na abstraktní třídy tohoto modulu, na nichž si vysvětlíme základní principy fungování. Níže můžeme vidět diagram tříd 4.3, ze kterého by mělo být patrné, jaké mají mezi sebou jednotlivé třídy vztah a jakým způsobem jsou na sobě závislé

Wrapper. Jedná se o velmi jednoduché rozhraní, které je poskytováno modulu server, aby byl schopen spouštět jednotlivé exekuce. Více podrobností je možné o tomto rozhraní zjistit v sekci 3.1. Existuje především proto, aby třída `AbstractWrapper` mohla být generická a mohla poskytovat příjemnější rozhraní při programování nových wrapperů.

AbstractComponent. Po výše popsaném rozhraní `Wrapper` se jedná asi o nejabstraktnější třídu celého programu. Všechny komponenty, ze kterých se Wrapper jako takový skládá, budou mít pravděpodobně nějaké podobné vlastnosti. Chceme tedy, aby všechny komponenty dědily od této abstraktní třídy, které definuje jejich jednotné rozhraní pro dané společné vlastnosti. Zatím jde pouze o správu reference na instanci typu `WrapperExceptionsFactory`. Jedná se o továrnu pro vytváření výjimek se srozumitelnými hláškami. Jelikož problémy mohou nastat v prakticky jakékoliv třídě, je tato továrna spravována právě v `AbstractComponent`. Všechny komponenty tak mají k dispozici protected metodu `getExceptionsFactory`, pomocí které mohou přistupovat k instanci dané továrny. Není vhodné vytvářet nové instance `WrapperExceptionsFactory` za běhu exekuce, pokud k tomu programátor nemá velmi dobrý důvod.

AbstractWrapper. Jedná se o jednu z nejdůležitějších tříd celého programu. Jak již napovídá její název jde o abstraktní třídu. Implementuje rozhraní `Wrapper` a vlastní tedy veřejnou metodu `executeQuery`. Třída je generická a má tři generické parametry, které mají následující význam.

- `TResult`: typ nativního výsledku pro používaný driver databáze.
- `TQuery`: typ nativního dotazu pro používaný driver databáze.



Obrázek 4.3: Wrappers diagram tříd

- `TPlan`: typ vráceného exekučního plánu dle používaného driveru databáze.

Generické parametry jsou dále užívané v dalších komponentách samotného wrapperu a mají stejný význam, jaký je zde uveden.

Instance této třídy žije prakticky po celou dobu běhu aplikace. Wrapper se inicializuje v momentě, kdy se nad ním spouští první dotaz. Inicializované wrappery pak existují dokud nedojde k zastavení aplikace.

Udržuje reference na instance tříd `AbstractQueryResultParser` a `AbstractExplainPlanParser`, které tím pádem mají stejnou životnost jako popisovaný `AbstractWrapper`. Nakonec má dvě vnořené třídy. První a ta méně zajímavá je třída `ConnectionData`, která pouze ukládá data načtená pro připojení ze souboru `application.properties`. Druhá, která je podstatně zajímavější, je třída `ExecutionContext`. O této třídě se však ještě budeme bavit, a proto ji pro zatím přeskočíme.

`AbstractWrapper` má velké množství *protected* method, které jsou zde pro definování chování vlastní implementace nového wrapperu. Tomuto tématu se ovšem budeme věnovat až v následující sekci 4.3, což je také místo, kde je možné se dozvědět více.

AbstractQueryResultParser. Opět se budeme bavit o abstraktní třídě, která je generická s jedním parametrem `TResult`. Jejím hlavním úkolem je parsovat získané výsledky z původního typu `TResult` na naše výsledky typů `CachedResult` a `ConsumedResult` dle volby. Definuje dvě veřejné metody, které je nutné doimplementovat podle zvoleného řadiče databáze.

- `parseResultAndCache` – jde o metodu, která původní výsledek vezme a na-parsuje do CachedResult, čímž jej celý uloží do paměti. Je vhodné to používat pouze pokud nás zajímají vrácené hodnoty a víme, že výsledek nebude příliš velký.
- `parseResultAndConsume` – metoda, která výsledek projde a uloží pouze statistická metadata o výsledku. Výsledek se neukládá a pouze se zpracuje. Vhodné použít pokud nás zajímají statistiky výsledku, který by jinak byl příliš velký.

Důvod je takový, že většina nativních výsledků, o které se musí vývojář starat při komunikaci s databází, implementuje rozhraní `AutoCloseable`. Většinou pak při exekuci dalších dotazů nad daným zdrojem dochází k jeho recyklaci a můžeme o předchozí výsledky přijít, pokud jsme je nestihli zpracovat. Návrh by měl programátora donutit se nad tímto faktem zamyslet a implementovat vhodně řešení, aby pak mohl parsované výsledky dále používat bez starostí. Zároveň získáme jednotné rozhraní výsledků, což zjednodušuje čitelnost kódu mezi jednotlivými implementacemi wrapperů.

AbstractExplainPlanParser. Jde o abstraktní a generickou třídu, která má jeden parametr `TPlan`. Parsuje získaný exekuční plán pro vstupní dotaz uživatele. Definuje veřejnou metodu `parsePlan`, kterou je třeba implementovat pro konkrétní wrapper. Tato metoda na vstupu očekává instanci `DataModel` a instanci typu `TPlan`, což je typ pro plán, který chceme zpracovat. Následně pak parametry naparsované z exekučního plánu ukládá do předaného objektu typu `DataModel`.

AbstractDataCollector. Tato třída má na starosti sběr všech dodatečných parametrů pro které je potřebné volat další pomocné databázové dotazy. Je opět abstraktní a generická s parametry `TResult`, `TQuery` a `TPlan`. Definuje veřejnou metodu `collectData`, která na vstupu očekává výsledek vstupního dotazu. Tuto metodu je nutné implementovat pro konkrétní wrapper. Třída nabízí rozhraní v podobě *protected* metod pro usnadnění této implementace, které podrobněji probereme v následující sekci 4.3. Životnost této třídy je po dobu vyhodnocování vstupního dotazu. Je to z toho důvodu, že kvůli sběru pomocných dotazů, potřebujeme referenci na instanci typu `AbstractConnection`, které umožňuje komunikovat s databází pod konkrétním spojením. Reference není přímá a je udržována přes kontext, o kterém se budeme bavit vzápětí.

ExecutionContext. Při vyhodnocování dotazu potřebujeme mít možnost nějakým způsobem předávat mezi jednotlivými funkcemi všechny kontextové parametry jedné exekuce dotazu. K tomu slouží tato třída, která všechny potenciálně důležité parametry shromažďuje a nabízí jednoduché rozhraní k přistupování. Jako jedna z mála zde popsaných tříd není abstraktní. Při vytváření nového wrapperu není potřeba se starat o její životní cyklus, pokud programátor nepotřebuje vlastní derivát této třídy s větším množstvím parametrů. Postup je pak podrobněji popsán v následující sekci 4.3, kde zároveň je podrobněji popsáno i rozhraní této třídy. Jedná se o třídu pouze generickou s typy `TResult`, `TQuery`, `TPlan`.

Životnost této třídy je pouze v kontextu vyhodnocení jednoho vstupního dotazu. Dává to dobrý smysl, neboť pokud nechceme třídu recyklovat, pak by nemělo smysl udržovat při životě velké množství instancí pro již vykonané dotazy. Ohledně recyklace by se pak člověk mohl zamyslet nad tím, jestli by to nebyl efektivnější přístup. Což necháváme částečně jako otevřenou otázku a co se týče výkonu by to mohla být jedna z cest, jak jej vylepšit. Na druhou stranu, když se pro každý dotaz vytváří nová instance, pak je v kódu daleko lépe čitelnější životnost tohoto objektu.

Při vytváření instance očekává na vstupu konstruktoru instanci samotného wrapperu. Je to z důvodu, aby se nastavila reference na `AbstractConnection` pro `ExecutionContext`. Všechny komponenty pak k připojení přistupují právě přes kontext, abychom zjednodušili rozhraní metod a předávání parametrů. Proces připojení je řešen pomocí privátních metod a je skrytý pro programátora, který bude implementovat nový wrapper, aby se předešlo tomu, že reference bude nechtěně přenastavena. Aby měl objekt `ExecutionContext` přístup k privátním položkám wrapperu a byl schopen referenci správně nastavit, je tak celý tento typ implementovaný jako vnořená třída `AbstractWrapper`.

Dále pak drží referenci na instanci datového modelu, ke kterému se v komponentách přistupuje pomocí kontextu. Nebo ukládá řetězec se vstupním dotazem a referenci na instanci `ConnectionData`, aby bylo možné přistupovat k těmto informacím v různých komponentách wrapperu během exekuce.

AbstractConnection. Vracíme se k popisu abstraktních tříd, které je nutné implementovat při vytváření vlastního wrapperu. Jde o abstrakci připojení pro databázi, které slouží k držení zdrojů pro databázový řadič po dobu exekuce dotazu. Jedná se o lehké zdroje, které se doporučuje nedržet v paměti příliš dlouho, u kterých se předpokládá například efektivní recyklace jednotlivých instancí. Naproti tomu reference na velké a drahé zdroje jsou drženy v samotném wrapperu. Třída pak zprostředkovává samotné spouštení dotazu a je nutné implementovat toto pomocí zdrojů originálního řadiče. Celá třída pak implementuje rozhraní `AutoCloseable`, aby umožnila lepší rozhraní pro práci s uvolňováním držených zdrojů. Třída je opět parametrická s typy `TResult`, `TQuery`, `TPlan`. Je zde nadefinováno několik veřejných metod pro spouštění dotazu nad spojením, které si stručně představíme.

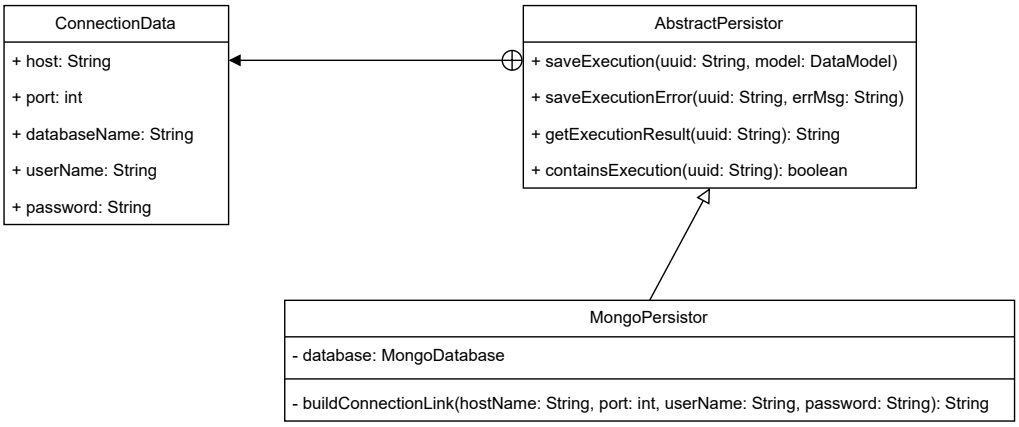
- `executeQuery` – exekuce klasického dotazu.
- `executeWithExplain` – exekuce dotazu, u kterého chceme rovnou získat i jeho exekuční plán, výsledek dotazu je typu `ResultWithPlan<TResult, TPlan>`.

- `isOpen` – kontroluje, zda je `AbstractConnection` otevřená, vrací boolean.

ResultWithPlan. Je vnořenou třídou `AbstractConnection` a je generická s parametry `TResult` a `TPlan`. Jedná se pouze o záznam reprezentující výsledek s plánem jako dvojici.

4.1.4 Persistor

Jedná se o nejjednodušší modul celé aplikace a obsahuje nejmenší množství tříd. Níže je zobrazený diagram 4.4 tohoto modulu pro přehlednost.



Obrázek 4.4: Persistor diagram tříd

ConnectionData. Jedná se o analogickou třídu `AbstractWrapper.ConnectionData` pro persistor. Jde tedy o vnořenou třídu v `AbstractPersistor` a ukládá data pro připojení instance persistoru.

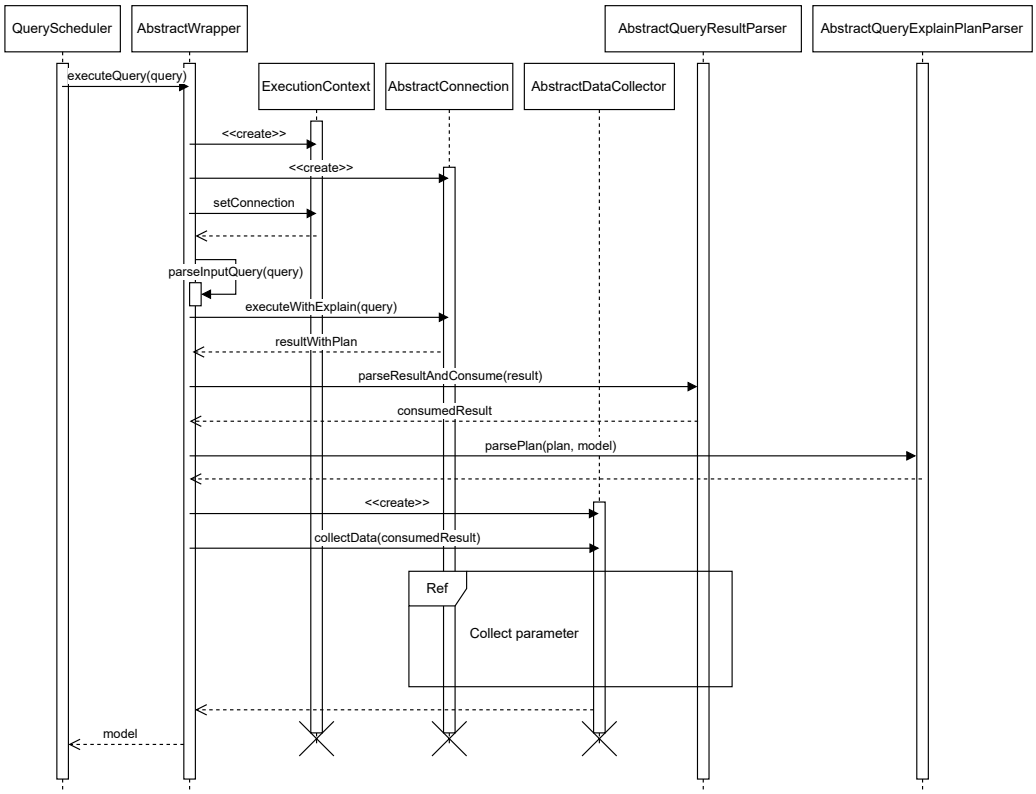
AbstractPersistor. Jedná se o poskytované API tohoto modulu, jeho veřejné metody jsou podrobně popsány v sekci 3.1. Modul server pomocí veřejných metod této třídy volá jednotlivé akce, které potřebuje pro danou exekuci provézt.

MongoPersistor. Implementuje rozhraní `AbstractPersistor` a zprostředkovává ukládání logů do databáze MongoDB. Drží referenci na nativní rozhraní řadiče `MongoDatabase`, pomocí něhož je připojený na patřičnou databázi.

4.2 Sekvenční diagram

V této sekci se budeme bavit o sekvenčních diagramech. Přiblížíme tak čtenáři použití jednotlivých funkcí popsaných v předchozí kapitole 4.1. Rozebereme zde hned dva diagramy, ačkoliv se dohromady jedná o pouze jednu prováděnou operaci. Diagramy na sebe odkazují, neboť pro lepší přehlednost jsem z diagramu pro exekuci dotazu vyndal část o sběru parametrů v objektu `AbstractDataCollec-`

tor. Vyňatá část o sběru parametrů je pak zobrazena v následujícím diagramu. Budeme se jim tedy věnovat postupně.

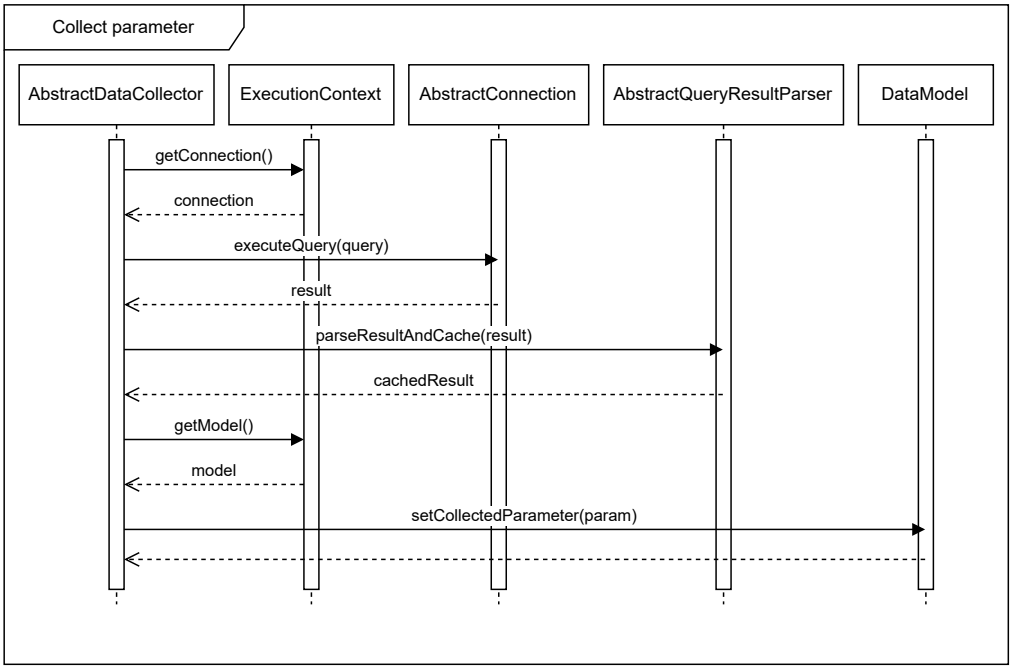


Obrázek 4.5: Sekvenční diagram exekuce dotazu

Pokud se podíváme na diagram 4.5 pro exekuci dotazu. Nejspíš si na první pohled všimneme, že spouštěčem akce není uživatel, ale objekt `QueryScheduler`. Uživatel pouze vytvoří záznam ke spuštění, ale samotné spouštění má na starosti onen plánovač. V diagramu chybí ještě předcházející část výběru správné instance. Tato pasáž mi přijde na fungování celkem triviální a nakonec jsem ji do diagramu nezahrnul hlavně z důvodu vyšší přehlednosti.

Prakticky ihned po zavolání funkce `executeQuery` nad konkrétním wrapperem, dojde k vytvoření instance `ExecutionContext`. Během tohoto procesu dojde i k vytvoření nové instance datového modelu pro sběr parametrů. Objekt datového modelu v sekvenčním diagramu není vyobrazen, jelikož se na všech místech, kde se používá, se předává pouze jako parametr a nejsou na něm volány žádné jeho metody. Po vytvoření objektu `AbstractConnection` se připojení zaregistruje pro již vytvořený kontext. Následně se provede parsování vstupního dotazu na originální typ řadiče `TQuery`. Přes `AbstractConnection` se následně spustí zadaný dotaz se ziskem exekučního plánu. Následuje parsování výsledku na typ `ConsumedResult` a plánu s datovým modelem, do něhož se uloží z plánu načtené informace. Nyní je potřeba sesbírat zbylé parametry o vyhodnocení dotazu, k čemuž potřebujeme vytvořit instanci třídy `AbstractDataCollector`. Na nově vytvořeném objektu pak zavoláme metodu `collectData`, během které dojde ke sběru všech parametrů. Tento postup si podrobněji popíšeme v následujícím

odstavci pomocí diagramu 4.6. Po doběhnutí metody `collectData` se následně instance datového modelu vrací zpět plánovači. Ten se pak musí postarat o korektní uložení do persistoru.



Obrázek 4.6: Sekvenční diagram sběru parametru

Přišel čas představit si sběr parametrů. Pro jednoduchost je v diagramu 4.6 zobrazen pouze sběr jediného parametru. Metoda `setCollectedParam` v modelu vůbec neexistuje, ale je zde použitá pro demonstrativní účely. Analogicky, pokud sbíráme parametr o velikosti databáze, pak budeme volat metodu `setDatabaseByteSize` a podobně. Rozhraní datového modelu je popsáno v sekci 4.3. V diagramu 4.6 je tak vyobrazen pouze sběr jednoho parametru. Zjednodušeně by se dalo říci, že tato operace se vykonává opakovaně podle počtu sbíraných parametrů. Není to úplně stoprocentní pravda, neboť některé parametry se pouze odhadují a dopočítávají, pak nedochází k dohledávání pomocí databázových dotazů. Což je scénář, který zde není vyobrazen, ale obecně se jedná o jednodušší variantu.

Z diagramu 4.6 je patrné, že `ExecutionContext` poskytuje `AbstractCollection` přístup k připojení či datovému modelu. Pomocí připojení pak zavolá objekt metodu `executeQuery` pro získání výsledku pomocného dotazu. Původní výsledek se následně zpracuje pomocí metody `parseResultAndCache`. Z kešovaného výsledku načtený parametr se nakonec uloží do datového modelu pomocí adekvátní metody. Pokud se jedná o názvu nějakých entit, pak se většinou předávají dál k získání podrobnějších informací o dané entitě.

4.3 Aplikační rozhraní (API)

V této části se budeme zabývat především rozšiřitelností programu o nové databázové systémy. Postupně se prokoušeme důležitým rozhraním, které je potřeba znát, pokud se rozhodneme Sběratele dále rozšiřovat o podporu dalších systémů. Pro implementaci nové podpory jsou stěžejní dva moduly. Jedná se o modul datového modelu a modul samotných wrapperů. V této sekci se budeme podrobně věnovat jednotlivým konceptům rozhraní a metodám. Mohlo by tak být užitečné pro získání většího přehledu si nejprve alespoň zběžně prohlédnout předchozí sekci 4.1. Jsou zde lépe popsané především vzájemné vztahy jednotlivých tříd, kterým zde již nebude věnován takový prostor.

4.3.1 DataModel

Pokud se rozhodneme rozšiřovat systém o sběr nových parametrů, určitě je vhodné prostudovat si sekci 4.1.2, kde se věnujeme podrobněji návrhu datového modelu. Zde si představíme rozhraní, které je nutné znát, pokud implementujeme vlastní wrapper a budeme chtít pracovat s datovým modelem aplikace. Prakticky celá tato část je věnována pouze popisu jednotlivých veřejných metod rozhraní `DataModel`, které je poskytované modulem datového modelu.

Veřejné metody rozhraní jsou následující:

- `createForQuery` – statická metoda pro vytváření prázdných instancí datového modelu pro konkrétní dotaz, systém a databázi.
- `toJson` – metoda pro serializaci instance datového modelu do formátu json pro ukládání a zápis.
- `setResultExecutionTime` – nastaví pro výsledek čas vykonání dotazu.
- `setResultByteSize` – nastaví velikost výsledku v bytech.
- `setResultSizeInPages` – nastaví velikost výsledku v počtu stránek.
- `setResultRecordCount` – nastaví počet záznamů ve výsledku dotazu.
- `setResultAttributeTypeByteSize` – nastaví velikost typu pro atribut ve výsledku v bytech. Pokud atribut se zadaným názvem neexistuje, vytvoří nový.
- `setResultAttributeTypeRatio` – nastaví velikost poměr pro daný typ atributu ve výsledku. Pokud atribut se zadaným názvem neexistuje, vytvoří nový.
- `setDatabaseByteSize` – nastaví velikost databáze, nad kterou byl dotaz spuštěn, v bytech.
- `setDatabaseSizeInPages` – nastaví velikost databáze v počtu stránek.
- `setDatabaseCacheSize` – nastaví velikost paměti pro kešování v bytech.
- `setPageSize` – nastaví velikost jedné stránky v bytech.

- `getPageSize` – získá nastavenou velikost stránky pomocí předchozí metody. Vhodné pro následné výpočty s velikostí stránek.
- `setKindByteSize` – nastaví velikost zadaného kindu v bytech. Pokud kind zadaného názvu neexistuje, vytvoří nový.
- `setKindSizeInPages` – nastaví velikost zadaného kindu v počtu stránek. Pokud kind zadaného názvu neexistuje, vytvoří nový.
- `setKindRecordCount` – nastaví počet záznamů zadaného kindu. Pokud kind zadaného názvu neexistuje, vytvoří nový.
- `setKindConstraintCount` – nastaví počet omezení nad zadaným kindem. Pokud kind zadaného názvu neexistuje, vytvoří nový.
- `addKind` – vytvoří nový prázdný kind s daným názvem.
- `getKindNames` – získá názvy aktuálně držených kindů pro datový model.
- `setIndexByteSize` – nastaví velikost zadaného indexu v bytech. Pokud index zadaného názvu neexistuje, vytvoří nový.
- `setIndexSizeInPages` – nastaví velikost zadaného indexu v počtu stránek. Pokud index zadaného názvu neexistuje, vytvoří nový.
- `setIndexRowCount` – nastaví počet záznamů zadaného indexu. Pokud index zadaného názvu neexistuje, vytvoří nový.
- `addIndex` – vytvoří nový prázdný index s daným názvem.
- `getIndexNames` – získá názvy indexů pro instanci datového modelu.
- `setAttributMandatory` – nastaví zda se jedná o povinný atribut v datové sadě. Pokud entity se zadanými názvy neexistují, vytvoří nové.
- `setAttributeValueRatio` – nastaví hodnotu vyjadřující distribuci hodnot v daném sloupci. Pokud entity se zadanými názvy neexistují, vytvoří nové.
- `getAttributMaxByteSize` – získá maximální velikost v bytech pro daný atribut. Pokud nějaká ze zadaných entit neexistuje, vyhazuje výjimku `DataModelException`.
- `setAttributeTypeByteSize` – nastaví velikost typu v bytech. Pokud entity se zadanými názvy neexistují, vytvoří nové.
- `setAttributeTypeRatio` – nastaví poměr pro daný typ v atributu . Pokud entity se zadanými názvy neexistují, vytvoří nové.
- `addAttributeType` – vytvoří nový typ pro zadaný atribut s daným názvem. Pokud entity se zadanými názvy neexistují, vytvoří nové.
- `getAttributeTypeByteSize` – získá velikost zadaného typu v bytech. Pokud nějaké z entit podle názvu není nalezena, pak skončí výjimkou `DataModelException`.

4.3.2 Wrappers

Zde si představíme nejdůležitější metody a třídy, pro rozšiřitelnost aplikace, které je vhodné znát z modulu wrapperů. Nejprve začneme popisem tříd, které jsou pouze zmíněny v předchozí sekci, jelikož v souhrnném pohledu na aplikaci nejsou až tak důležité. Jde o typy výsledků `CachedResult` a `ConsumedResult`. Při rozšiřování aplikace s nimi ovšem budete nuceni pracovat a proto si projdeme jejich rozhraní. Dále se budeme věnovat jednotlivým abstraktním třídám, které je nutné implementovat pro správné fungování wrapperu.

CachedResult. Pokud jsou výsledkem dotazu hodnoty, které nás zajímají, pak potřebujeme uložit celý výsledek dotazu. Původní výsledek dotazu na tento typ mapuje korektní implementace `AbstractQueryResultParser`. Tato třída tak kešuje výsledek tak jak je. V praxi stačí pouze projít získaný původní výsledek a hodnoty zapsat do instance typu `CachedResult`. Následně disponuje rozhraním inspirovaným u typu `ResultSet` z rozhraní JDBC:

- `next` – metoda, která vrací boolean, jestli je možné se přesunout na další záznam. Slouží k iteraci.
- `containsAttribute` – kontroluje, zda záznam obsahuje atribut se zadaným názvem.
- `getInt` – pro zadaný název atributu vrací jeho hodnotu přetypovanou jako integer.
- `getString` – pro název atributu vrací hodnotu jako řetězec.
- `getDouble` – pro název atributu vrací hodnotu jako typ double.
- `getBoolean` – pro název atributu vrací hodnotu jako boolean.
- `getLong` – pro název atributu vrací hodnotu jako typ long.
- `getDocument` – pro název atributu vrací hodnotu jako instanci `Document`. Používáno v kontextu MongoDB.
- `getList` – generická metoda podle typů prvků v seznamu, který chceme získat. Pro zadaný název a třídu se tak snaží hodnotu naparsovat jako seznam instancí dané třídy.
- `getRecordCount` – vrací počet záznamů ve výsledků.

CachedResult.Builder. Možná jste již zaregistrovali, že konstruktor třídy `CachedResult` je privátní. Je to z důvodu, že proces vytváření výsledků je iterativní a nemusí být úplně triviální. Vytváření instancí je tak implementováno pomocí návrhového vzoru Builder. Rozhraní builderu je následující:

- `addEmptyRecord` – přidá na konec prázdný záznam, do kterého je následně možné zapisovat.
- `toLastRecordAddValue` – do posledního záznamu pro zadaný název atributu přidá zadanou hodnotu.

- `toResult` – metoda, která s nasbíranými záznamy zinicializuje instanci typu `CachedResult`.

ConsumedResult. Dalším typem výsledků jsou ty, u nichž si nepotřebujeme pamatovat získané hodnoty. Důležité jsou pro nás metadata, toho co jsme z dotazu získali. Opět se parsování původních výsledků na typ `ConsumedResult` implementuje ve třídě `AbstractQueryResultParser`. Podporované rozhraní je následující:

- `getAttributeNames` – získá seznam názvů všech nalezených atributů.
- `getAttributeTypes` – pro název atributu získá seznam nalezených datových typů hodnot v atributu.
- `getAttributeTypeRatio` – pro název atributu a specifikovaný datový typ vrací relativní četnost typu pro daný atribut.
- `getByteSize` – získá velikost celého výsledku v bytech.
- `getRecordCount` – vrací počet záznamů ve výsledku.

ConsumedResult.Builder. Opět jako u `CachedResult` je tento výsledek vytvářen pomocí návrhového vzoru Builder. Builder je vnořená třída typu `ConsumedResult`. Pro každý typ a název atributu si pamatuje jejich absolutní čestnost. Z čehož je následně instance typu `ConsumedResult` schopna vypočítat všechny potřebné výsledky. Disponuje následujícím rozhraním:

- `addRecord` – fakticky pouze inkrementuje počet záznamů, ale sémanticky přidává nový záznam.
- `addAttributeType` – pro název atributu přidá pozorovaný typy. Typy jsou evidované pomocí jejich názvů a jsou předávány jako řetězce.
- `addByteSize` – inkrementuje celkovou velikost výsledku v bytech. Při průchodu původního výsledku se počítá s tím, že pro každý záznam odhadneme jeho velikost v bytech, což je hodnota, o kterou chceme velikost pravidelně inkrementovat.

AbstractWrapper. Nejdůležitější abstraktní třída, kterou je potřebné implementovat. Všechny wrappery implementují rozhraní `Autoclosable`, jelikož drží zdroje pro připojení k databázi, které je při destrukci nutné korektně uzavřít. Je tak potřebné v první řadě přepsat metodu `close`. Následně je potřeba definovat korektně chování metody `executeQuery`, což je složitější proces, na který plynule navážeme v dalších částech popisu. Nejprve je nutné si uvědomit, že metodu není možné přepsat, což je záměr. Pokud nahlížíte do kódu zjistíte, že metoda je `final`. Je to z důvodu abychom se vyhnuli „špagetovému“ kódu, kdy kompletní implementace bude pouze v jedné třídě a metodě. Přesto je možné definovat chování této metody. Používáme k tomu návrhový vzor Template Method. Veřejná metoda `executQuery` je template metoda a pro definování jejího chování musíme přepsat její pomocné metody. Níže popsané abstraktní třídy je nutné všechny implementovat, neboť pomocné metody předpokládají jejich existenci. V potomkovi je nutné přepsat všechny tyto pomocný metody:

- `createConnection` – slouží k inicializaci vaší implementace potomka třídy `AbtractConnection`, předávaný parametrem je instance třídy `ExecutionContext`. Ačkoliv `ExecutionContext` vlastní referenci na `AbstractConnection` v tuto dobu není nastavena. Nastavení se provede automaticky hned po úspěšné inicializaci všeho spojení.

- `parseInputQuery` – vstupní dotaz je vždy řetězec. Systémy mohou však dotazy vyžadovat v jiném typu (například u MongoDB je nutné aby to byla instance `Document`), k parsování vstupního dotazu do správného nativního typu slouží tato metoda. Vstupní parametr je opět `ExecutionContext`, který obsahuje referenci na vstupní dotaz. Ten je tak předáván pouze jako součást kontextu.

- `createDataCollector` – slouží k inicializaci vaší implementace potomka třídy `AbstractDataCollector`. Předávaným parametrem je opět `ExecutionContext`, který obsahuje jak data pro připojení, tak referenci na instanci `DataModel`, které se mohou v tomto kontextu hodit.

Pokud znáte návrhový vzor *Template Method*, pak víte, že kromě pomocných metod existují také *hook* metody. Hook metody není nutné implementovat, pokud nejsou potřebné. Metody jsou v předkovi implementované prázdným tělem. Nabízí možnost algoritmus rozšířit o další kroky, které by se mohly hodit pro nově vytvářené systémy. Některé hook metody jsou součástí zdrojového kódu již teď, jde o následující:

- `setDependenciesBeforeExecutionIfNeeded` – metoda pro nastavení speciálních závislostí před exekucí dotazu. Například u MongoDB `MongoQueryResultParser` potřebuje ke svému fungování `AbstractConnection`, závislost je tak nastavena v této metodě. Metoda je volána po vytvoření kontexu a připojení ale před spuštěním vstupního dotazu.

- `removeDependenciesAfterExecutionIfPossible` – pokud pomocí výše uvedené metody nastavíte nové závislosti, nejspíše se pak hodí možnost je opět anulovat, aby stav mezi exekucemi byl validní, pokud entity mají životnost pouze v kontextu exekuce.

Výše popsané jsou klasické metody, které opravdu nemají žádnou funkčnost. Systém ovšem disponuje i speciálními *hook* metodami, které definují nějaké chování, ale je možné je volitelně přepsat, pokud je potřeba jejich funkčnost nějakým způsobem rozšířit či upravit. Jde o následující:

- `createExecutionContext` – metoda pro inicializaci instance kontextu, není potřeba přepisovat pokud nepoužíváte vlastní implementaci `ExecutionContext`.

Opustíme již návrhový vzor Template Method a metodu `executeQuery`. Pro správné fungování je ještě nutné přepsat následující protected metody volané v konstruktoru při inicializaci wrapperu:

- `createResultParser` – je nutné přepsat a slouží k inicializaci vlastního parseru pro výsledky. Musí vracet typ `AbstractQueryParse`, používá se pak při exekuci všech dotazů.

- `createExplainPlanParser` – jde o obdobný princip jako u předchozí metody. Tentokrát se jedná o vaší implementace typu `AbstractExplainPlanParser`, který se pak užívá pro parsování plánu dotazu.

AbstractQueryResultParser. Je potřeba implementovat jeho rozhraní, tudíž správné mapování výsledků na typy `CachedResult` či `ConsumedResult`. Životnost této třídy je stejná jako u wrapperu, jelikož ve většině případů není potřebné držet reference na objektu žijící pouze v kontextu jedná exekuce.

AbstractExplainPlanParser. Je potřeba implementovat jeho rozhraní pro pasrování plánu vstupního dotazu a uložení získaných parametrů do předané instance `DataModel`. Obdobně jako `AbstractQueryResultParser` má stejnou životnost jako `AbstractWrapper` prakticky ze stejných důvodů.

AbstractConnection. Je potřeba implementovat rozhraní této třídy. Podrobněji je popsáno v sekci 4.1. Důležité je uvědomit si, že životnost této instance je pouze po dobu jedné exekuce. Implementuje rozhraní `Autoclosable` a měla by tedy držet referenci na zdroje, které se dají recyklovat a nejsou těžké na inicializaci.

ExecutionContext. Je parametrem prakticky všech pomocných a *hook* metod v metodě `executeQuery`. Slouží k předávání dat během exekuce mezi jednotlivými kroky. V optimálním případě by nemělo být nutné implementovat vlastní. Pokud ovšem potřebujete předávat při exekuci mezi jednotlivými kroky speciální parametry, pak je možné podědit původní implementaci a vytvořit vlastní. V metodách pak bude nutné si instanci v případě potřeby potřeby přetypovat, abyste mohli přistupovat ke svým rozšířením. Životnost této komponenty je pouze v kontextu jedné exekuce. Jde o vnořenou třídu `AbstractWrapper` a je tak možné přistupovat k privátním vlastnostem `AbstractWrapper`. V případě sběratele parametrů (tj. instance třídy `AbstractDataCollector`) je `ExecutionContext` předán jako parametr při inicializaci a následně je nepřímo přístupný během celého procesu pro sběr dat, o tomto tématu si ještě povíme více v dalších částech této kapitoly. Pokud by to bylo nutné je možné přidat `ExecutionContext` jako předávaný parametr například při parsování exekučního plánu, což ale zatím nebylo potřeba. `ExecutionContext` má následující rozhraní pro přístup k parametrům:

- `getConnection` – jedná se o dvě metody, obě vrací referenci na instanci. `AbstractConnection`. V jednom případě je však možné přidat ještě typ jako generický parametr, na který se to následně přetypuje.
- `getInputQuery` – vrátí vstupní dotaz.
- `getModel` – vrátí instanci modelu pro exekuci.
- `getDatabaseName` – vrátí název databáze, v níž běží exekuce.
- `getSystemName` – vrátí název systémů, nad kterým běží exekuce. Zatím není používáno, ale možná se bude hodit. Zatím je zde pro úplnost.

AbstractDataCollector. Jedná se třídu, u níž by se dalo bezesporu tvrdit, že je srdcem celého procesu. Jak již bylo popsáno v sekci 4.1, jedná se o třídu zodpovědnou za sběr statistických parametrů. Při její implementaci je potřeba myslet na fakt, že životnost jedná instance je po dobu exekuce jednoho dotazu. Udržuje referenci na `ExecutionContext`, který je pak následně nepřímo přístupný v potomcích. Třída tedy obsahuje *protected* metody, které mají vývojáři usnadnit implementaci vlastního *DataCollectoru*:

- `executeQuery` – vykoná zadaný dotaz a vrátí `CachedResult`, použije k tomu vaší implementaci `AbstractConnection` a `AbstractQueryResultParser`.
- `executeQueryAndConsume` – stejný případ jako předchozí metoda, s rozdílem, že vrací výsledek typu `ConsumedResult`.
- `getModel` – vrací datový model aktuální exekuce, do kterého chceme ukládat sbírané parametry.
- `getDatabaseName` – vrací název databáze, nad kterou dotaz běží.

Pokud to bude vhodné v kontextu většího množství systémů je možné rozšířit do jisté míry i podporu protected metod, například pro rozšíření rozhraní pro komunikaci s kontextem.

WrappersExceptionsFactory. Třída, v níž by měly být definované všechny metody pro vytváření chybových hlášek. Blíže je popsána v sekci 4.1. Zde bych chtěl hlavně zmínit, že pokud programátor potřebuje vlastní hlášky specifické pro daný systém, je možné tuto třídu podědit a rozšířit o vlastní tvorbu hlášek. Následně je pak součástí všech abstraktních tříd wrapperu getter pro tuto instanci (viz. `AbstractComponent` v sekci 4.1). Je zde i generický getter pro získání instance specifického typu, který je pak možné i vhodné používat, abyste nemuseli třídu vlastnoručně neustále přetypovávat. Zde je výčet metod vytvářejících chybové hlášky v původní třídě `WrappersExceptionsFactory`:

- `connectionIsNull` – přístup na instanci `AbstractConnection`, která má hodnotu null nebyla pravděpodobně inicializována.
- `connectionNotOpen` – přístup na zavřenou `AbstractConnection`.
- `connectionNoNotInitialized` – chyba během inicializace instance třídy `AbstractConnection`.
- `queryExecutionFailed` – chyba při spuštění dotazu.
- `queryExecutionWithExplainFailed` – chyba při spuštění dotazu s exekučním plánem.
- `cataCollectionFailed` – chyba při sběru parametrů.
- `dataCollectorNotInitialized` – chyba během inicializace instance třídy `AbstractDataCollector`.
- `parseInputQueryFailed` – chyba při parsování vstupního dotazu.

- `parseExplainPlanFailed` – chyba při parsování exekučního plánu.
- `cacheResultFailed` – chyba při parsování dotazu na typ `CachedResult`.
- `consumeResuktFailed` – chyba během parsování dotazu a následného převodu na typ `ConsumedResult`.
- `wrapperInitializationFailed` – problém při inicializaci wrapperu.
- `unsupportedOperatio` – nepodporovaná operace.

Hlášky chyb, jejich možné příčiny a důsledky jsou řešené v jiné části, konkrétně v sekci 5.6. Zde se o metodách zmiňujeme jen z důvodu, aby měl programátor lepší přehled o již existujícím rozhraní a neimplementoval věci, které jsou již součástí řešení.

4.3.3 Server

V momentě, kdy máme implementovaný wrapper pro nový databázový systém, který chceme podporovat, nemáme ještě zcela vyhráno. Nyní je potřeba v modulu serveru tento wrapper vhodně zaregistrovat, aby se korektně inicializoval ze souboru `application.properties` a server s ním pak mohl pracovat.

SystemType. Tento výčtový typ je podrobněji popsán v sekci 4.1. Pokud vytváříme podporu pro nový systém je potřebné daný výčtový typ rozšířit o novou hodnotu pro nově přidávaný systém.

Initializers. V této fázi bychom měli mít hotový a připravený wrapper. Následně je nutné zaregistrovat jeho inicializaci, a k tomu slouží třída `Initializers`. Je potřeba, aby konstruktor vaší implementace wrapperu vyžadoval pouze parametr typu `AbstractWrapper.ConnectionData`. Nemělo by být nutné tento typ nikterak upravovat. Třída `Initializers` má dvě privátní položky v podobě map, do kterých si pro každý systém ukládá jejich inicializátor jako funkci se signaturou `Function<AbstractWrapper.ConnectionData, AbstractWrapper>`. Jinými slovy očekává na vstupu `AbstractWrapper.ConnectionData`, kterou získáme ze třídy `WrapperInstance` pomocí metody `getConnectionData` a následně vrací instanci `AbstractWrapper`. Disponuje následujícím rozhraním pro inicializaci:

- `registerWrapper` – zaregistruje zadaný inicializátor wrapperu pro uvedený typ systému.
- `registerPersistor` – zaregistruje stejným způsobem inicializátor Persistory. Pro persistory má separátní mapu s inicializátory.
- `initializeWrapper` – pro zadanou instanci třídy `AsbtractWrapper.ConnectionData` a systém najde v mapě vhodný inicializátor a inicializuje pomocí něj novou instanci třídy `AbstractWrapper`.
- `initializePersistor` – funguje stejný způsobem jako `intializeWrapper`, ale pro implementace persistoru.

Pro registraci wrapperu, či persitoru je potřebné ale udělat pouze jednu věc. Tou jest v privátní metodě `initialize` a anotací `PostConstruct` přidat řádat se zavoláním registrace pro váš nový systém a předat do metody odkaz na konstruktor wrapperu. V příadě nové implementace persistoru se postupuje prakticky stejným způsobem. Výsledek by mohl vypadat nějak takto:

```
@PostConstruct
private void _initialize() {
    // registrace ostatních wrapperů...
    registerWrapper(SystemType.NewSystem, MyNewWrapper::new);

    // registrace persitorů...
}
```

4.4 Implementační problémy

V této části si představíme problémy, na které jsem během práce narazil. Přiblížíme si důvody, které mě vedly ke konkrétním řešením, a další uvažované alternativy. Sekce je rozdělena na několik částí podle toho, jakých systémů se řešené problémy týkají.

4.4.1 Společné

Nejprve se budeme věnovat problémům, které se týkaly většího množství systémů a daly by se tedy kategorizovat jako společné. Některé vytyčené problémy se nemusejí týkat všech databázových systémů, ale pouze nějaké netriviální podmnožiny. Dalo by se to tedy shrnout tak, že tato část nepojednává o problémech, které souvisí pouze s jediným systémem. Problémy tohoto typu jsou pak pokryté v částech níže.

Vícetypové sloupce. Relační databáze mají pevně definované schéma tabulek a s tím souvisí i omezení typu nad jednotlivými sloupci. Když definujeme novou tabulku, každý její sloupec může být omezen pouze jediným typem. V případě NoSQL databází je však situace diametrálně odlišná. Záleží na konkrétním systému a na tom, jestli se jedná o tzv. *schema-less* nebo *schema-based* variantu. Řešení je celkem prosté, ale bezesporu je dobré se nad tímto problémem zamyslet a vůbec si uvědomit, že taková situace může nastat. Datový model tak umožňuje pro jeden sloupec výskyt většího množství typů. Je možné do datového modelu uložit i relativní četnosti těchto typů v kontextu jednoho sloupce. Je dokonce umožněno, aby pro libovolný sloupec chyběly informace o typu úplně, pokud nebylo možné je zjistit.

Chybějící hodnoty. Všechny systémy dále nabízejí možnost nějakým způsobem pracovat s chybějícími hodnotami. Z charakteristik jednotlivých databází plyne, že každá volí svou vlastní strategii, jak řeší tento problém. Všechny tyto přístupy tak bylo nutné sjednotit. V databázi MongoDB se chybějící položky v datech opravdu nevyskytují a jsou tak v dokumentech vynechány. U Neo4j je to řešené analogickým způsobem přeneseným na jednotlivé uzly a hrany v grafu.

Přístup relačních databází je naproti tomu takový, že pokud není hodnota v relaci přítomna, pak sloupec na tomto místě nabývá speciální hodnoty `null`. Program tak o sloupcích zjišťuje, jestli umožňují výskyt chybějících hodnot. Řešení je takové, že do datového modelu je u sloupce možné uložit, jestli jde o povinný sloupec či nikoliv. Což odpovídá tomu, může-li sloupec obsahovat chybějící hodnoty. U relačních databází zjistit tuto informaci bývá celkem triviální, neboť ji databáze ukládá do metadat o tabulkách a stačí tak pouze zjistit, v jaké meta-tabulce se informace nachází. Naproti tomu u databáze MongoDB je potřeba zjistit, jestli pro danou kolekci dokumentů existuje schéma, ze kterého se informace o tom, jestli je daná položka povinná, dá naparsovat. V opačném případě je nutné předpokládat, že položka může v nějakých dokumentech zcela chybět. U Neo4j je situace o něco složitější, neboť ačkoliv pro graf může být definované schéma, tak dostat se k těmto metadatům není pomocí klasických dotazů v jazyce Cypher možné. Je potřeba použít externí knihovnu APOC, která je schopná zjistit metadata o položkách pro konkrétní uzel, kde je už možné se k informaci dostat.

Přetečení velikostí. Při původním návrhu datového modelu jsem ukládal nasbírané velikosti o jednotlivých entitách v bytech do proměnných typu `int`. Během implementace a práce s databázemi jsem si však uvědomil, že tato varianta je silně nedostačující a je téměř jisté, že i pro ne až tak velké databáze by mohlo docházet k přetečení daných hodnot. Následně by tak naměřené velikosti vůbec neodpovídaly zkoumané realitě. Naměřené hodnoty pro jednotlivé velikosti entit jsou tak výhradně typu `long`.

Pohledy. Dále bylo potřeba vyřešit práci s pohledy v jednotlivých systémech. Nejjednodušší řešení se nabízelo u Neo4j, kde se problém může jednoduše ignorovat. Nenabízí totiž žádnou podporu pro vytváření pohledů. Pokud uživatel chce něco takového použít, je nucen vytvořit si nový typ vlastních uzlů, které pak bude používat v dotazech jako pohledy.

U MongoDB a PostgreSQL už tento problém není tak triviální. Oba systémy podporují tvorbu klasických a zároveň i materializovaných pohledů. V případě klasických pohledů se jedná o variantu, kdy pro pohled v databázi neexistuje žádná relace, která by jej definovala. Pohled je tvořený pomocí dotazu, který se v systému efektivněji kešuje, když dochází k jeho opětovnému vykonávání. Na druhou stranu při vytváření materializovaného pohledu se v systému vytvoří nový typ relace. Dotaz se tak opětovně nevyhodnocuje. Je ovšem náročnější tuto novou tabulku udržovat v případě aktualizace dat v původních tabulkách, nad kterými je materializovaný pohled vytvořen.

Nabízely se tak dvě strategie, jak se s problémem vypořádat. První variantou bylo detekovat, zda dochází k použití pohledu. Pohledy by se tak ve výsledcích chovaly jako tabulky, ale k výsledkům by se přidala nějaká výčtová hodnota, zda se jedná o klasický pohled, materializovaný či normální tabulku. Implementace tohoto přístupu však není úplně jednoduchá, protože bychom potřebovali zjišťovat metadata o pohledech, což obnáší zjistit metadata mezivýsledku po vyhodnocení dotazu definující pohled. Uživatelem zadaný dotaz by se tak musel rozdělit v momentě, kdy došlo k vyhodnocení pod-dotazu, který je ekvivalentní dotazu, jenž definuje pohled, nasbírat důležité hodnoty a až následně by se pokračovalo klasickým způsobem, kdy se vyhodnotí celkový výsledek.

První variantu jsem tak nakonec zavrhl, protože implementace není úplně triviální a zároveň druhý případ daleko lépe reflektuje skutečné chování materializovaných a klasických pohledů uvnitř systému při vyhodnocování uživatelem zadaných dotazů. Druhou variantou tedy je přístup, u něhož s klasickými pohledy se zachází jako s dotazy a v případě materializovaných pohledů jako s tabulkami. Sbírají se tak metadata skutečných tabulek, nad kterými je klasický pohled sestaven. Pro pohled je tak potřeba najít tyto tabulky, což se dá zjistit z exekučních plánů dotazu. K materializovaným pohledům se pak náš program chová, jako kdyby se jednalo o klasické tabulky. Tyto tabulky, jelikož v systému skutečně existují, mají v databázi vlastní metadata o velikostech a datových typech, které pak zahrneme do celkového výsledku o vyhodnocení dotazu.

Jednotný formát výsledků. Vzhledem k tomu, že se jedná o různé systémy, které disponují různými dotazovacími jazyky. Tak také jejich knihovny a řadiče, které byly určené pro práci s nimi v jazyce Java, obsahují různé typy pro práci s výsledky zadaných dotazů. K podpoření rozšiřitelnosti programu jsem se tak rozhodl vytvořit vlastní typy, které ukládají výsledky dotazů. Tento přístup umožňuje vhodnou abstrakci jednotlivých komponent samotného wrapperu a zároveň pomohl výsledky rozdělit na dvě varianty, které odpovídají požadavkům vytvořeného programu. V některých případech nás totiž nezajímá samotný výsledek, ale pouze jeho metadata a statistické informace. Aplikace proto obsahuje dva typy `CachedResult` a `ConsumedResult`, které tento koncept implementují. Instance `CachedResult` je výsledek, který kešuje všechny informace a celý se uloží do paměti. Naproti tomu `ConsumedResult` je typ, který původní výsledek pouze "zkonzumuje"a následně má uložené pouze jeho statistické informace.

4.4.2 Neo4j

Zde se budeme zabývat problémy, které byly charakteristické pro implementaci wrapperu Neo4j. Největším problémem bylo získání informace ohledně velikosti kindů.

Velikosti entit. Grafová databáze Neo4j nesbírá data o celkové velikosti a počtu uzlů s daným popiskem. Množina uzlů s jedním konkrétním popiskem pro nás představuje entitu ekvivalentní tabulkám v relačních databázích, což je i důvod, proč nás tato informace zajímá. Neobjevil jsem žádnou knihovnu, která by toto uměla zprostředkovat. Byl jsem tak nucen záležitost sám implementovat a spokojit se pouze s odhadovanou velikostí entit na základě informací, které jsem dohledal.

Pro jednotlivé typy atributů je možné dohledat, jak se ukládají na disk a jakou by měly mít velikost. Zároveň je zde popsáno jak vypadá vytvořený index nad deklarovanými vlastnostmi. [2] Na základě těchto dohledaných informací se dá jednoduše spočítat odhadovaná velikost dat na disku. Algoritmus funguje tak, že prochází všechny entity v databázi s daným popiskem a postupně akumuluje výsledek podle atributů v každém uzlu. Byla možnost spočítat tyto hodnoty odhadem, k tomu bychom ovšem potřebovali uzel o průměrné velikosti, k čemuž

[2] zdroj: https://neo4j.com/developer/kb/understanding-data-on-disk

bychom museli stejně projít všechny uzly v databázi. Nemáme totiž jistotu, že první entita bude dobrý reprezentant celé množiny. Další alternativou pro počítání odhadu velikosti byla možnost využít knihovnu APOC spočítat vše na základě informací o vlastnostech pro jednotlivé entity. Problémy tohoto přístupu jsou hned dva. První je ten, že samotné statistiky pro vlastnosti daných entit jsou odhadované na základě zvolení množiny reprezentantů. A druhý problém je, že nám chybí informace o procentuálním výskytu jednotlivých atributů, které nejsou povinné. Nakonec jsem oba přístupy pro odhad této informace zavrhl a implementoval jsem výše popsaný postup.

4.4.3 MongoDB

Sekce popisuje problémy spojené s databází MongoDB. Jedná se především o problém s parsováním jazyka mongosh a nutnost implementace vlastní iterace nad získanými výsledky.

Parsování mongosh. Nativní řadič pro Javu vůbec nepodporuje zadávání dotazu jako řetězce v jazyce *mongosh*. Práce s řadičem je výhradně podporována pouze pomocí speciálních typů pro jednotlivé entity odpovídající například kolekci dokumentů, databázi, nebo samotným dokumentům. Očekává se, že programátor tedy bude užívat vhodné metody na těchto typech, které jsou velmi podobné metodám z jazyka *mongosh*. Jediná alternativa k tomuto přístupu je využívání rozhraní pro spouštění *Database command*[3]. Jeden *command* odpovídá jednomu JSON dokumentu, který definuje operaci mající se nad databáze provést. Toto rozhraní, tak umožňuje dynamicky měnit vstupní obsah podle uživatele, což je hlavní důvod proč jsem zvolil toto rozhraní k implementaci komunikace s databází MongoDB.

Většina uživatelů pracujících s databází MongoDB je zvyklá na používání dotazovacího jazyka *mongosh*. Jak jsem již zmínil na začátku, nativní řadič pro Javu práci s *mongosh* dotazy nepodporuje. Ačkoliv jsem se nejprve snažil najít pro Javu knihovnu, která by toto uměla vyřešit, byl jsem neúspěšný.

Aplikace tak obsahuje vlastní překladač, který jsem sám vytvořil, který danou problematiku řeší. Překladač očekává na vstupu řetězec v dotazovacím jazyce *mongosh*, ke kterému vytvoří ekvivalentní *command* popisující zadaný dotaz. Nejedná se o plnohodnotnou podporu tohoto jazyka, jsou podporovány pouze některý funkce. Jedná se především o funkce, které zprostředkovávají pouze operace čtecího typu. Bohužel z důvodů, ke kterým se ještě vrátíme, bychom mohli prohlásit, že Sběratel podporuje pouze funkci `find` a několik dalších funkcí operujících nad vzniklým kurzorem. Aplikace tak nepodporuje funkce `aggregate`, `count` a `distinct` ačkoliv jejich podpora byla původně zamýšlena.

Možná by se mohlo jevit, že hlavním důvodem byly problémy při parsování dotazů u uživatele. Ve skutečnosti jsou v překladači fragmenty kódu, které tuto problematiku implementují, ale jsou zakomentované. Jelikož používáme rozhraní pro exekuci dotazů pomocí *command*, přišli jsme o možnost iterace nad výsledky. Výsledkem každého volání je dokument, který má omezenou velikost. V tomto dokumentu se nachází v seznamu omezené množství dokumentů, které jsou součástí výsledku. Kořenový dokument se parsuje a pomocí identifikátoru kurzoru

[3] https://www.mongodb.com/docs/manual/reference/command/

a podobných informací je pak implementovaná možnost iterace přes všechny dokumenty z výsledku. Zde přichází zpět na scénu problém s podporou funkcí `aggregate`, `count` a `distinct`. Jejich výsledky a exekuční plány jsou navzájem celkem dost odlišné, což zároveň velmi komplikuje záležitost s iterováním přes výsledky.

Když jsem si uvědomil výše popsané problémy, tak jsem se rozhodl, že bude lepší když danou funkcionalitu nebude systém alespoň prozatím podporovat. V časovém horizontu, který jsem měl k dispozici, jsem neviděl jako realistické, že bych byl schopen vytvořit slušnou podporu pro výše popsané funkce. Je potřeba si uvědomit, že funkce `aggregate`, je v databázi MongoDB velmi silný nástroj a struktura výsledku je tak velmi variabilní.

4.5 Testování

V sekci se krátce zmíníme o testování aplikace. Rád bych totiž čtenáři představil, jak sem nad tímto tématem v uvažoval. Nakonec jsem většinu operací testoval manuálně a na realizaci automatických testů již nezbyl dostatek času. Přesto bych zde rád popsal nějaké myšlenky či nápady, které mne během tvorby díla napadli. Mohli by někomu pomoci při implementaci automatizovaných testů, pokud to do budoucna bude potřebné.

4.5.1 Unit testy

Testování pomocí unit testů nedává v kontextu tohoto softwarového díla příliš velký smysl. Důvod je především ten, že je program velmi závislý na databázových systémech, což dělání přínosných unit testů celkem komplikuje. Jediný případ, kde by dávalo nejspíš smysl implementovat unit testy, je v kontextu třídy `ConsumedResult`. Mohlo by dávat smysl zaměřit se na testování samotného počítání statistik, kde například velikosti v bytech se počítají inkrementálně. Bohužel převážně z časových důvodů a jiných priorit při práci na projektu jsem nakonec testy nerealizoval.

4.5.2 Integrační testy

Automatizované integrační testy pro aplikaci by mohly být v budoucnu celkem přínosné. Jedná se o celkem pracnou záležitost. Nakonec jsem tedy volil možnost manuálního testování. Vyhodnotil jsem tuto variantu jako méně časově náročnou. Nakonec jsem tedy sehnal datovou sadu, jejíž data jsem po jednodušších transformacích nahrál do jednotlivých systémů. Jelikož jsem věděl jak jsou data velká, celkem snadno se mi pak kontrolovaly jednotlivé výsledky. Program jsem tak testoval inkrementálním způsobem, kdy po implementaci nové funkce jsem ihned danou věc testoval. Když jsem při testování narazil na nějaký problém, pak jsem se rozhodoval, zda se vyplatí danou věc podporovat či nikoliv. Tyto problémy jsou většinou popsané v předchozí sekci 4.4.

5. Uživatelská dokumentace

V předchozích kapitolách jsme si prošli prakticky celým procesem analýzy a vývoje softwaru. Zbývá tedy poslední věc. Představit si uživatelskou dokumentaci. Nejprve se podíváme na minimální softwarové a hardwarové požadavky. Plynule na to navážeme popisem instalace a spuštění. Abychom uživateli poskytli kromě popisu taky nějakou praktickou ukázku, popíšeme si základní příklad použití systému. Nakonec rozebereme signifikantní chybové hlášky a nastíníme jejich možné příčiny a tudíž i způsoby, jak vzniklé problémy řešit.

5.1 Minimální HW a SW požadavky

Oficiálně by se dalo říci, že program nemá žádná omezení na hardware či operační systém. Je závislý pouze na existenci JRE či nástroje docker. Záleží na tom, zda se rozhodneme pouštět program v kontejneru či nikoliv. Celá aplikace pak běží v kontejneru, který si sám stáhne všechny potřebné závislosti a nastaví prostředí. Zde popsané požadavky by tak nemělo být potřebné řešit pro potřeby instalace s puštění programu, pokud nenastane během instalace žádný problém.

Program je psaný v jazyce Java ve verzi 17. Pro práci s programem a hlavně pokud budeme aplikaci rozšiřovat, je pak důležité mít nainstalovaný JDK 17[1]. V případě pouštění aplikace pomocí kontejneru není potřeba mít tuto závislost nainstalovanou, neboť kontejner se o vše postará a vytvoří vhodné prostředí pro fungování aplikace. Projekt používá pro sestavení a management závislostí nástroj zvaný Gradle[2] ve verzi 8.4.

Aplikace pro databázi MongoDB používá aktuálně řadič ve verzi 4.11.1. Tento řadič plnohodnotně funguje s databázemi MongoDB verze 6 a 7. Pro databázi MongoDB verze 8 je pak vhodné použít řadič vyšší verze, ačkoliv výrobce uvádí že by měl fungovat v režimu, kde nepodporuje nové funkce databáze. Sběratel byl však testován s verzí 6, tato verze je používaná i v případě že spouštíme aplikaci přes docker. K databázi MongoDB je jedna z nejdůležitějších podmínek fungování se storage enginem WiredTiger[3]. Pokud budete používat jinou variantu, může nastat nedefinované chování v kontextu sběru parametrů, které jsou na enginu závislé. V lepším případě budou takové parametry ve výsledku pouze chybět, v horším budou končit výjimkou. Například již velikost stránky je nastavena právě podle tohoto enginu.

V případě databáze Neo4j používá aplikace řadič verze 5.28.5, který by měl být kompatibilní s verzemi databáze od 4.4 po 5.X. Pro testování a v kontejneru se používá verze 5.26.0. Jedná z nejdůležitějších závislostí, které sběratel hojně využívá je knihovna APOC. Bez této knihovny nebude sběr dat vůbec možné realizovat, neboť u parametrů, k jejichž sběru je to potřebné, bude pak sběr končit výjimkou. Je tedy potřeba, aby plugin byl součástí instance Neo4j.

Ke komunikaci s databází PostgreSQL využívá aplikace řadič verze 42.7.3. U poslední verze řadiče se aktuálně udává, že by měla být kompatibilní s verzemi databáze 8.2 a vyšší. Očekával bych že v případě našeho řadiče to bude velmi

[1] https://www.oracle.com/java/technologies/downloads/#java17
[2] https://docs.gradle.org/8.4/userguide/userguide.html
[3] https://www.mongodb.com/docs/manual/core/wiredtiger/

podobné. A tedy Sběratel by měl fungovat s verzemi databáze PostgreSQL 8.2 a vyšší. Pro testování a při kontejnerizaci se využívá verze PostgreSQL 14.5.

5.2 Instalace

V této části se budeme zabývat instalací programu. Jelikož k fungování Sběratele potřebujeme nástroj pro kontejnerizaci, je potřeba nejprve nainstalovat docker. Vhodné je stáhnout si jejich oficiální aplikaci Docker Desktop[4], kde je možné následně pomocí pěkného rozhraní spravovat jednotlivé kontejnery na vašem počítači. Po úspěšné instalaci či v případě vašeho přesvědčení, že máte stažený docker, si můžete tuto skutečnost ověřit spuštěním v terminálu.

```
> docker -v

Docker version 23.0.5, build bc4487a
```

Další prerekvizitou k vlastní instalaci je mít nainstalovaný git. Je to z důvodu, že instalace proběhne klonováním gitového repozitáře. Pokud nemáte na svém stroji nainstalovaný verzovací systém git, pak je možné jej nainstalovat například zde[5]. Ověřit instalaci je možné podobným způsobem jako v případě dockeru.

```
> git -v

git version 2.40.1
```

Nyní je potřeba přesunout se do složky, kam chceme stáhnout všechny zdrojové soubory projektu. Zde je potřeba naklonovat soubory z repozitáře. To se dá provézt následujícím příkazem.

```
> git clone https://github.com/composerPigeon/Collector.git

Cloning into 'Collector'...
remote: Enumerating objects: 2184, done.
remote: Counting objects: 100% (876/876), done.
remote: Compressing objects: 100% (353/353), done.
remote: Total 2184 (delta 317), reused 716 (delta 190),
pack-reused 1308 (from 1)
Receiving objects: 100% (2184/2184), 409.20 KiB | 4.87 MiB/s, done.
Resolving deltas: 100% (758/758), done.
```

V případě že plánujeme dělat v kódu nějaké změny a budeme pracovat na rozšiřování funkcí systému, je potřeba před naklonováním projektu si udělat vlastní fork.

Pokud klon proběhl dobře, je nutné přesunout se do složky naklonovaného projektu. Můžeme provézt například tímto příkazem, pokud používáme operační systém Linux či MacOS.

[4] https://www.docker.com
[5] https://git-scm.com/downloads

```
> cd Collector
```

Nyní je potřeba projekt zkompilovat. Potřebujeme mít totiž před kontejnerizací vytvořené `.jar` soubory naší aplikace, se kterými je následně možné vytvořit docker image aplikace. Jak již bylo zmíněno dříve projekt používá k sestavení a managementu závislostí nástroj zvaný Gradle. Nemělo by být potřeba nic instalovat, neboť nástroj je součástí projektu. Provedeme následující příkaz.

```
> ./gradlew clean build

BUILD SUCCESSFUL in 2s
15 actionable tasks: 11 executed, 4 up-to-date
```

V tento moment bychom mohli prohlásit instalaci za hotovou. Pokud čtenáře nezajímají další podrobnosti, je tak možné pokračovat další sekcí 5.3. Aplikaci pro účel demonstrace jejích základních funkcí pouštíme pomocí nástroje docker. V další části se tedy budeme věnovat všem skriptům a konfiguračním souborům, které umožňují spustit aplikaci v kontejneru se všemi databázemi.

5.2.1 Dockerfile

Začneme nejprve jednodušší variantou. Popíšeme si soubor `Dockerfile`, který se nachází ve složce server.

```
FROM openjdk:17
WORKDIR /collector
COPY server/build/libs/*.jar app.jar

#EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Příkaz `FROM` určuje jaký image je základem pro vytvoření image naší aplikace. V našem případě zde uvedeme právě JDK s patřičnou verzí, které je potřebné pro kompilaci a spuštění programu. Dále `WORKDIR` vytvoří v kontejneru pracovní složku s názvem *collector*, ve které budeme pracovat. Příkaz `COPY` všechny soubory s kompilovaným kódem překopíruje do souboru `app.jar` do kontejneru. Zakomentovaný `EXPOSE` nemá v tomto případě žádný vliv na vytvoření docker image, jelikož přístupnost portů je řešena v souboru `docker-compose.yaml`, o němž se ještě budeme bavit. Nakonec `ENTRYPOINT` je příkaz, který spustí `.jar` soubory naší aplikace. `Dockerfile` tak popisuje jak má vypadat image pro naší aplikaci. Víceméně pouze zkopíruje soubory aplikace na správné místo, odkud pak aplikaci spouští docker, když uživatel vytvoří a spustí kontejner.

5.2.2 Docker Compose

Pro konfiguraci jednotlivých kontejnerů a jejich sítě používáme nástroj Docker Compose. V kořenovém adresáři projektu je tak soubor `docker-compose.yaml`, kde jsou definované všechny důležité vlastnosti pro spuštění kontejnerů. Souborem

se tedy postupně prokoušeme, abychom si vysvětlili důležitá nastavení. Budeme postupovat po jednotlivých službách. Součástí projektu je i složka *databases*, kam se ukládají data z databází a zároveň jsou zde inicializační skripty pro vytvoření demonstračních dat. Konfigurační soubor `docker-compose.yaml` vypadá takto.

```
services:
  server:
    build:
      dockerfile: server/Dockerfile
    container_name: server
    ports:
      - "8080:8080"
    depends_on:
      - mongodb
      - postgres
      - neo4j

  mongodb:
    container_name: mongodb
    image: mongodb/mongodb-community-server:6.0-ubi8
    restart: always
    ports:
      - "27017:27017"
    environment:
      - MONGODB_INITDB_ROOT_USERNAME=mongo
      - MONGODB_INITDB_ROOT_PASSWORD=password
      - MONGODB_INITDB_DATABASE=admin
    volumes:
      - ./databases/mongodb/data/db:/data/db
      - ./databases/mongodb/init.d/:/docker-entrypoint-initdb.d/

  postgres:
    container_name: postgres
    image: postgres:14.5
    restart: always
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_PASSWORD=password
      - POSTGRES_USER=postgres
    volumes:
      - ./databases/postgres/data/db:/var/lib/postgresql/data
      - ./databases/postgres/init.d/:/docker-entrypoint-initdb.d/

  neo4j:
    container_name: neo4j
    image: neo4j:5.26.0
    restart: always
    ports:
      - "7474:7474"
      - "7687:7687"
    environment:
```

```
      - NEO4J_AUTH=neo4j/password
      - NEO4J_PLUGINS=["apoc", "apoc-extended"]
      - NEO4J_initial_dbms_default__database=sales
      - EXTENSION_SCRIPT=/script/import_data.sh
    volumes:
      - ./databases/neo4j/data/db:/data
      - ./databases/neo4j/data/plugins:/plugins
      - ./databases/neo4j/data/logs:/logs
      - ./databases/neo4j/import:/import
      - ./databases/neo4j/script:/script
```

Pomocí atributu `container-name` nastavujeme pro všechny kontejnery jejich název, který je pak viditelný v aplikaci Docker Desktop po spuštění. V atributu `image` u databází nastavíme docker image jaký používají k vytvoření kontejneru. Jde o položky, které je nutné stáhnout před prvním spuštěním, a proto nejspíš bude toto spuštění trvat delší dobu. Následné u databází se nachází atribut `restart`, kterým nastavíme kdy chceme kontejner restartovat. V případě že dojde k restartu docker deamon, pak se uplatní uvedená konfigurace pro restart kontejnerů.

Všechny kontejnery pak obsahují sekci `ports`, kde se mapují porty na kontejneru na porty na lokálním stroji či uvnitř docker sítě pro komunikaci mezi kontejnery. Kontejnery na sebe v rámci této sítě vidí pomocí názvů služeb uvedených v `docker-compose.yaml`. Názvy služeb jsou tedy hodnoty pro `hostName` jednotlivých wrapperů v souboru `application.properties`, o kterém se budeme podrobněji bavit v sekci 5.5.

V sekci `environment` pak pro všechny databáze nastavujeme přihlašovací údaje pro kořenového uživatele. Toto nastavení není bezpečné, neboť Sběratel má administrativní práva k jednotlivým databázím a může tak činit prakticky cokoliv. Při nasazení se určitě vyplatí pro Sběratele vytvořit vlastní roli, kterou mu v databázi přiřadit. Pro demonstrativní účely ovšem nastavení postačí a navíc je jednoduché na vysvětlení. Virtuální disky popíšeme v kontextu jednotlivých služeb, stejně tak jako některá nastavení ze sekce `environment`.

Server. Služba startuje image aplikace pomocí výše popsaného `Dockerfile`. Následně je definováno v `depends-on`, že je závislá na všech ostatních službách. Což je opět jednoduché nastavení, které zajistí, že po nastartování budou spuštěny všechny kontejnery.

Postgres. V části `environment` pouze nastavujeme přihlašovací údaje pro hlavního uživatele. Zajímavá je pak část `volumes`. Zde mapujeme složku `databases/postgres/data/db` pro použití jako adresář pro data z databáze. Dále zde mapujeme složku `databases/postgres/init.d`, která obsahuje inicializační SQL skript pro vytvoření dat a nastavení databáze pro účely demonstrace. Po startu služby se docker do této složky podívá a postupně v abecedním pořadí nad databází spustí jednotlivé skripty. Skript končí příkazem `ANALYZE`, jelikož data jsou příliš malá a databáze tak po inicializaci sama příkaz nespustí. Nemusí tak být uložené statistiky tabulek a Sběratel pak nemusí fungovat úplně spolehlivě.

Mongodb. V `environment` nastavujeme název administrativní databáze na *admin* a nastavujeme údaje pro administrativního uživatele. V části `volumes` pak dochází k analogickému mapování složek na virtuální disky jako v případě PostgreSQL. Opět jsou zde tedy na začátku spouštěny inicializační skripty. V tomto případě jsou hned dva, `queries.js` se stará o vytvoření databáze *queries* a v ní kolekce *executions*. Skript `sales.js` se pak stará o vytvoření stejnojmenné databáze a následně ji plní daty pro demonstrativní účely spuštění programu.

Neo4j. Nakonec se budeme zabývat nastavením služby pro grafovou databázi. V části `environment` kromě nastavení informací o administrativním uživateli a přejmenování databáze na *sales*, dochází také k nastavení pluginů, které pro databázi chceme používat. Po prvním spuštění se tak tyto pluginy stahují. Dále je zde `EXTENSION-SCRIPT`, který má na starosti načtení ze složky import do databáze. Skript je psaný v shellu a je možné jej najít ve složce `databases/neo4j/script`. V části `volumes` se následně mapují všechna potřebná úložiště. Všechny se mapují do složky `database/neo4j/data`. Výjimka je mapování složky se skriptem a následně složky s daty k importu ve formátu CSV do úložiště *import*.

5.3 Spuštění

V případě že instalace proběhla úspěšně, měli byste mít stažený celý projekt. V předchozí sekci 5.2 jsme si představili, jak vypadají soubory pro konfiguraci docker kontejnerů. Nyní je nutné aplikaci spustit.

```
> docker compose up --attach server
```

Příkaz `docker compose up` postaví a spustí všechny kontejnery popsané v souboru `docker-compose.yaml`. Parametr `-attach server` pak zajistí, že do konzole budou posílány výpisy pouze pro službu server. Pokud vše proběhlo správně, měli byste vidět dlouhý výpis logů při inicializaci všech komponent důležitých pro Spring. Většinou se snaží server připojit k persistoru již před inicializací MongoDB, pak během inicializace vznikne výjimka při připojování k peristoru. Aplikace se ovšem zkouší připojit opakovaně a po té co se nastartuje MongoDB to vždy funguje. Neměl by tedy nastat žádný markantní problém. Na poslední řádce byste pak měli vidět výpis podobný následujícímu příkladu. Výpis je zalomený, aby byl dobře čitelný:

```
<datetime> INFO <processId> --- [collector] [main]
    c.c.matfyz.collector.server.Application :
    Server is running on port 8080.
```

Spouštění může trvat delší dobu, neboť se stahují všechny docker závislosti, kterých není málo. Většinou vše proběhlo v řádu několika minut, ale záleží na rychlosti internetu. V momentě, kdy se vám do konzole vypíše již dříve zmíněná hláška, je Sběratel připraven k použití.

5.4 Uživatelské rozhraní

V této sekci si podrobně představíme REST-ové rozhraní serveru, abychom jej mohli začít používat. Popíšeme jednotlivé endpointy a zároveň si přiblížíme používaný formát komunikace. Každý název odstavce endpointu se skládá z použité HTTP metody a následně cesty k danému zdroji. Součástí cesty jsou pak proměnné jako `<server>` nebo `<executionId>`. Proměnná pro server je prostě jen adresa serveru, na které Sběratel běží. Naproti tomu `<executionId>` je vždy potřeba nahradit uživatelem za identifikátor exekuce o který se uživatel v kontextu operace zajímá.

POST <server>/query

- **Operace:** pro název instance a zadaný dotaz vytvoří exekuci, kterou zařadí do fronty pro budoucí spuštění
- **Požadavek:** server očekává požadavek ve formátu JSON. Předaný objekt musí mít dva řetězcové atributy *instance* a *query*, kam uvede vstupní dotaz a následně název instance nad níž chce dotaz spustit. Zároveň se očekává že bude nastavená hlavička na `Content-type: application/json`. Příklad těla požadavku:

```
{
    "instance": "postgres",
    "query": "SELECT * FROM table WHERE column > 5"
}
```

- **Odpověď:** pokud vše proběhlo správně, pak vrátí uživateli odpověd s kódem 200 a v těle odpovědi bude JSON dokumemt s jediným atributem *executionId*. Hodnota atributu je identifikátor nově vytvořené exekuce pro zadaný dotaz.

GET <server>/query/<executionId>/state

- **Operace:** získá stav exekuce podle `executionId`
- **Požadavek:** server neočekává žádný obsah v těle požadavky. Identifikátor je získán z cesty.
- **Odpověď:** pokud vše proběhlo dobře, vrátí server json objekt se dvěma atributy. První je `executionId` s identifikátorem exekuce pro přehlednost a druhý atribut `state` vrací dohledaný stav exekuce, možné hodnoty jsou:
  - `Waiting`: exekuce čeká ve frontě na spuštění
  - `Running`: exekuce aktuálně běží (tj. již byla spuštěna, ale zatím nedoběhla)
  - `Processed`: exekuce již doběhla a bude mít uložený výsledek v persistoru

  Příklad:

```
{
    "executionId": "6c69b25e-faf9-4fef-960a-cb9a148347cb",
    "state": "Processed"
}
```

GET <server>/query/<executionId>/result

- **Operace:** pokud je exekuce ve stave `Processed`, dohledá její výsledek a vrátí uživateli.
- **Požadavek:** server neočekává žádné tělo požadavku, neboť identifikátor je součástí cesty
- **Odpověď:** v případě kladné odpovědi dostane uživatel výsledek ve formátu JSON. Objekt má dva atributy `executionId` a následně buď `model` nebo `error` v závislosti na tom, zda exekuce doběhla bez problémů či nikoliv. Pokud výsledek není v persistoru nalezen, pak vrací chybu s kódem 404 not found. Příklad kladné opovědi:

  ```
  {
      "executionId": "6c69b25e-faf9-4fef-960a-cb9a148347cb",
      "model": { ... }
  }
  ```

  Model může být celkem velký a není zde tedy vypsán. Jde o JSON objekt s nasbíranými parametry, pro lepší představu sbíraných atributů a struktury modelu je dobré si prostudovat sekci 4.1.2 Příklad odpovědi pro chybnou exekuci:

  ```
  {
      "executionId":"f5d9413c-bc14-45a3-b68e-609af87f6342",
      "error": "..."
  }
  ```

  Opět pro přehlednost chybí příklad přesné hlášky. Více informací k chybovým hláškám je možné dohledat v sekci 5.6

GET <server>/instances/list

- **Operace:** získá seznam připojených instancí
- **Požadavek:** server neočekává tělo požadavku. Požadavek nemá žádné parametry.
- **Odpověď:** v případě kladné odpovědi dostane uživatel výsledek ve formátu json. Objekt má jeden atribut `instances`, v nemž se nachází list připojených instancí. U každé instance se vypíše systém a název instance. Příklad odpovědi:

```
{
    "instances":[
        {"systemType":"Neo4j","instanceName":"neo4j"},
        {"systemType":"PostgreSQL","instanceName":"postgres"},
        {"systemType":"MongoDB","instanceName":"mongo"}
    ]
}
```

5.5 Ukázka použití

V sekci se zaměříme na základní použití systému pro generování datové sady. Nejprve se zaměříme na spuštění dotazu a všechny operace, které se dají během evaluace používat. Nakonec si pak představíme návod, jak zaregistrovat novou instanci či persistor.

5.5.1 Zavolání dotazu

Nejprve se budeme věnovat příkladu užití pro zavolání dotazu. Obecně si vyzkoušíme představené REST rozhraní představené v předchozí sekci 4.3. Příklady jsou demonstrované pomocí nástroje curl, který je potřebné nainstalovat, pokud tomu tak není. Alternativou je využití jednoduchého klienta psaného v pythonu, který je součástí projektu. Je potřeba mít nainstalovaný python verze 3, v nejlepším případě verzi 3.12. Nakonec jsem se kvůli nutnosti instalovat python rozhodl demonstrovat příklady pomocí nástroje curl. Neboť jde o známý nástroj a na většině počítačů je k dispozici bez nutnosti instalace.

Předpoklad je, že nyní se nám podařilo spustit aplikaci a můžeme tedy zkoušet, co Sběratel umí. Budeme si nejprve hrát na neznabohy. Rádi bychom zjistili, jaké instance jsou připojené ke sběrateli, abychom věděli na jaké instance můžeme volat dotazy. To provedeme následujícím příkazem a níže vidíme získaný výsledek.

```
> curl -X GET http://localhost:8080/instances/list

{
    "instances":[
        {"systemType":"Neo4j","instanceName":"neo4j"},
        {"systemType":"PostgreSQL","instanceName":"postgres"},
        {"systemType":"MongoDB","instanceName":"mongo"}
    ]
}
```

Je vidět, že pro demonstraci jsou ke Sběrateli připojeny právě tři instance, pro každý systém jedna. Chceme-li tedy spustit dotaz nad instancí *postgres*, budeme postupovat následujícím způsobem.

```
> curl -X POST \
>   http://localhost:8080/query \
>   -d '{ "instance": "postgres", "query": "SELECT * FROM customer" }'\
>   -H "Content-Type: application/json"
```

```
{
    "executionId":"6c69b25e-faf9-4fef-960a-cb9a148347cb"
}
```

Všimněme si, že musíme správně nastavit hlavičku. Následně dostaneme výsledek s identifikátorem nově vzniklé exekuce, který se vyplatí si zkopírovat pro další použití

Předtím než se budeme zajímat o výsledek evaluace dotazu, je vhodné se přesvědčit, zda dotaz exekuce již doběhla. To můžeme provézt tímto příkazem. Je důležité vložit do cesty požadavku správný identifikátor.

```
> curl -X GET \
>   http://localhost:8080/query/6c69b25e-faf9-4fef-960a-cb9a148347cb/state

{
    "executionId":"6c69b25e-faf9-4fef-960a-cb9a148347cb",
    "state":"Processed"
}
```

Pokud se přesvědčíme, že exekuce již proběhla, můžeme postupovat dále. Pokud je ze stavu exekuce patrné, že exekuce ještě čeká ve frontě či zrovna dochází k jejímu zpracování, je vhodné chvíli počkat. Požadavek na získání stavu exekuce je v takovém případě vhodné zopakovat.

Pokud víme, že exekuce již doběhla, můžeme se podívat na její výsledek. Pokud během evaluace dotazu nenastala žádná chyba, měli bychom vidět následující příklad. Model je pro účely demonstrace formátovaný, což pro výsledek neplatí, pokud nepoužíváte nějaké nástroje pro tento účel.

```
> curl -X GET \
>   http://localhost:8080/query/6c69b25e-faf9-4fef-960a-cb9a148347cb/result

{
    "executionId":"6c69b25e-faf9-4fef-960a-cb9a148347cb",
    "model":{
        "systemName":"PostgreSQL",
        "databaseName":"sales",
        "query":"SELECT * FROM customers",
        "databaseData":{
            "databaseSize":9085731,
            "databaseSizeInPages":1110,
            "pageSize":8192,
            "cacheSize":134217728,
            "tables":{
                "customers":{
                    "recordCount":50,
                    "constraintCount":1,
                    "sizeInPages":1,
                    "byteSize":24576,
                    "attributes":{
                        "fname":{
```

```
                        "mandatory":false,
                        "distinctValuesCount":38,
                        "types":{
                            "varchar":{"ratio":1.0,"byteSize":7}
                        }
                    },
                    "lname":{
                        "mandatory":false,
                        "distinctValuesCount":38,
                        "types":{
                            "varchar":{"ratio":1.0,"byteSize":7}
                        }
                    },
                    "id":{
                        "mandatory":false,
                        "distinctValuesCount":50,
                        "types":{
                            "int4":{"ratio":1.0,"byteSize":4}
                        }
                    },
                    "age":{
                        "mandatory":false,
                        "distinctValuesCount":45,
                        "types":{
                            "int4":{"ratio":1.0,"byteSize":4}
                        }
                    },
                    "addressid":{
                        "mandatory":false,
                        "distinctValuesCount":50,
                        "types":{
                            "int4":{"ratio":1.0,"byteSize":4}
                        }
                    }
                }
            }
        },
        "indexes":{}
    },
    "resultData":{
        "resultKind":{
            "recordCount":50,
            "sizeInPages":1,
            "byteSize":1300,
            "attributes":{
                "fname":{
                    "types":{
                        "varchar":{"ratio":1.0,"byteSize":7}
                    }
                },
                "lname":{
```

```
                        "types":{
                            "varchar":{"ratio":1.0,"byteSize":7}
                        }
                    },
                    "id":{
                        "types":{
                            "int4":{"ratio":1.0,"byteSize":4}
                        }
                    },
                    "age":{
                        "types":{
                            "int4":{"ratio":1.0,"byteSize":4}
                        }
                    },
                    "addressid":{
                        "types":{
                            "int4":{"ratio":1.0,"byteSize":4}
                        }
                    }
                }
            },
            "executionTime":0.932
        }
    }
}
```


V modelu vidíme, že se dotaz týkal tabulky *customers*, ve které bylo 50 záznamů. Do výsledku se pak dostalo také 50 záznamů. Neodpovídá zde například velikost pro tabulku v bytech a výsledku v bytech. Ačkoliv se může zdát, že se jedná o chybu, databáze většinou reportují prostor alokovaný pro danou tabulku včetně možnosti alokace metadat ohledně tabulky. Pro výsledek se ovšem počítají velikosti manuálně přesně podle vrácených sloupců. Z toho plyne, že hodnoty nemusejí být stejné. U výsledku dále například vidíme, jak dlouho trvalo vyhodnocení dotazu na straně databáze.

5.5.2 Zkušební dotazy

V ukázce jsme si představili jeden jednoduchý dotaz pro spuštění nad databází PostgreSQL. Chtěl bych zde ukázat ještě pár možných dotazů, pro PostgreSQL a další databáze, na kterých je možné si systém vyzkoušet.

PostgreSQL. V relačním modelu máme relace: `Customer(id, fname, lname, age, adressId)` a `Address(street, housenumber, city, zipcode)`, kde addressId je reference do tabulky `Address`

- Zákaznící, kteří jsou v důchodovém věku

```
SELECT *
FROM customer
WHERE age > 65
```

- Zákazníci, kteří bydlí ve Springfieldu

```
SELECT c.*
FROM customer AS c
    INNER JOIN address AS a ON c.addressId = a.id
WHERE a.city = 'Springfield'
```

Neo4j. V grafu Neo4j jsou uzly se stejnými vlastnostmi jako v případě relačního modelu. Máme tedy uzly pro zákazníky a adresy.

- Všichni zákazníci

```
MATCH (c:Customer) RETURN c
```

- Zákazníci, kteří jsou v důchodovém věku

```
MATCH (c:Customer) WHERE c.age > 65 RETURN c
```

- Zákazníci, kteří bydlí ve Springfieldu

```
MATCH (c:Customer)-[:HAS_ADDRESS]->(a:Address)
WHERE a.city = "Springfield"
RETURN c
```

MongoDB. V případě hierarchického modelu máme opět stejná data, ale tentokrát pouze jednu kolekci dokumentů s názvem `customers`, kde adresy jsou vnořeným objektem.

- Všichni zákazníci

```
db.customers.find()
```

- Zákazníci, kteří jsou v důchodovém věku.

```
db.customers.find({age: { $gt: 65 }})
```

- Zákazníci, kteří bydlí ve Springfieldu

```
db.customers.find({"address.city": "Springfield"})
```

5.5.3 Přidání instance

Uživatel se pravděpodobně nesmíří s možností využívat pouze databáze poskytnuté pouze pro demonstrativní účely v rámci dockeru. Nejspíš bude chtít přidat vlastní nové instance, pro které bude chtít vytvářet datovou sadu. Tuto problematiku si tak představíme ve této části textu. Pro jakoukoliv manipulaci s instancemi je důležitý soubor `application.properties`. Nejdeme jej ve složce `server/src/main/resources`. Pro předchozí ukázku vypadal soubor takto:

```
spring.application.name=collector
spring.task.scheduling.pool.size=4

wrappers[0].systemType=Neo4j
wrappers[0].instanceName=neo4j
wrappers[0].hostName=neo4j
wrappers[0].port=7687
wrappers[0].databaseName=sales
wrappers[0].userName=neo4j
wrappers[0].password=password

wrappers[1].systemType=PostgreSQL
wrappers[1].instanceName=postgres
wrappers[1].hostName=postgres
wrappers[1].port=5432
wrappers[1].databaseName=sales
wrappers[1].userName=postgres
wrappers[1].password=password

wrappers[2].systemType=MongoDB
wrappers[2].instanceName=mongo
wrappers[2].hostName=mongodb
wrappers[2].port=27017
wrappers[2].databaseName=sales
wrappers[2].userName=mongo
wrappers[2].password=password

persistor.systemType=MongoDB
persistor.hostName=mongodb
persistor.port=27017
persistor.databaseName=queries
persistor.userName=mongo
persistor.password=password
```

První dva řádky se týkají především nastavení aplikace. Můžeme kromě názvu aplikace například změnit počet vláken pro obsluhu plánovače. Další řádky jsou pak věnovány nastavení jednotlivých wrapperů a persistoru.

Představme si, že bychom chtěli přidat do souboru novou instance pro již existující databázi, ale chceme upravit nad jakou databází se budou dotazy spouštět. I když se jedná o stejnou databázi, přidáme pro tento účel novou instanci. Budeme si tak moci během používání vybrat jakou instanci a tudíž i databázi si pro dotaz zvolíme. Přidání instance funguje tak, že přidáme nové řádky s novým indexem pro list `wrapppers` a nastavíme potřebné parametry. Výsledek bu mohl vypadat například takto.

```
wrappers[3].systemType=PostgreSQL
wrappers[3].instanceName=postgresNewInstance
wrappers[3].hostName=postgres
wrappers[3].port=5432
wrappers[3].databaseName=newDatabase
```

```
wrappers[3].userName=postgres
wrappers[3].password=password
```

Přidali jsme tak novou instanci s názvem `postgresNewInstance` se změnou databáze na název *newDatabase*. Aby byly změny platné, je potřeba uložit pozměněný soubor a pozastavit a následně opět spustit Sběratele. Název nově přidané instance musí být unikátní, jinak při startu skončí Sběratel chybou.

Analogickým způsobem je pak možné změnit parametry další, pokud se bude jednat o jiný databázový server či jiného uživatele. Parametr `systemType` se mapuje na výčtový typ `SystemType` a tudíž název systému musí odpovídat hodnotám výčtového typu. Povolené hodnoty jsou `PostgreSQL`, `MongoDB` a `Neo4j`.

Pokud nastavujeme instance v rámci docker compose, musí hodnota `hostName` odpovídat názvu služby v souboru `docker-compose.yaml`. Pokud běží databáze na stejném stroji, je možné nastavit `hostName` na hodnotu *localhost*.

5.5.4 Nastavení persistoru

Obdobně, jako v případě instancí lze v souboru nastavit i parametry potřebné pro připojení k persistoru. Systém ovšem podporuje existenci pouze jediného persistoru, takže při změnách je vhodné pouze upravit stávající hodnoty a nepřidávat další. V případě persistoru pro databázi MongoDB pak Sběratel předpokládá existenci databáze s návem *queries* a kolekce `executions`. Je podle toho tedy nutné přizpůsobit nastavení databáze. Systém aktuálně podporuje persistory pouze pro systém `MongoDB`. Pro jiné bude končit chybou, protože je neumí inicializovat a ani nejsou na-implementované. Pro nový persistor je tak potřeba v souboru `application.properties` nastavit požadované údaje. Především `hostName`, `port` a údaje pro přihlášení.

5.6 Chybová hlášení a jejich řešení

V této sekci se podíváme na chybové hlášky sběratele. Budeme se zabývat dále netradičním chování programu, a jejich možnými příčinami. Jde o problémy, na které jsem během práce narazil a nejsou ani tak problémem Sběratele jako spíš důsledkem nešťastného nastavení databází a prostředí.

5.6.1 Formát hlášek

Nejprve se budeme zabývat chybovými hláškami. Především si představím jejich strukturu. Jde především o chyby, které mohou nastat během exekuce dotazu a uživatel se k nim dostane při zobrazení výsledku. Chyby, které se logují na serveru pak mají formát velmi podobný, ale klasický uživatel již tyto chyby nevidí. O jejich výskytu se dozví pomocí `http` kódu a automatické hlášky serveru pro daný kód, která nebývá povětšinou příliš sdělná.

Většina chyb, které se objeví během exekuce mají nějakou příčinu. Kolikrát příčinou může být i chyba, která nepochází z kódu sběratele, ale byla vyhozena výjimkou při komunikaci se řadičem databáze. Výjimky se vzájemně obalují, když se šíří ven. Formát hlášek tak tuto skutečnost reflektuje. Každá výjimka má

svou hlášku a nějakou příčinu, pokud není sama původní chybou. Každá chybová hláška má tedy následující strukturu, která se může rekurzivně aplikovat na hlášku příčiny.

```
"<errorMessageOfTopException>, cause { <errorMessageOfCause> }"
```

Pro vizuální oddělení jednotlivých hlášek a lepší čitelnost hierarchie, jsou hlášky příčin obaleny ve složených závorkách. Hlášky jsou následně konstruovány tak, aby pro problém poskytli co možné nejvíce informací o kontextu. V jednoduchých apostrofech jsou vždy uvedeny názvy entit, kterých se chyba týká. Jedná se o tabulky, sloupce, identifikátory, dokumenty, názvy indexů apod.

Příklad hlášky, kdy selhala exekuce, protože v modelu nešlo najít název atributu s daným typem.

```
Running execution 'f5d9413c-bc14-45a3-b68e-609af87f6342'
    with query 'SELECT * FROM customers'
    on instance 'postgres' failed, cause {
        No byte size for column 'fname'
        of type 'varchar' was found in data model
    }
```

Hláška je rozdělena na několik řádek, aby byla čitelná.

5.6.2 Rady pro řešení chyb

Původně jsem zamýšlel, že bychom se v této části věnovali jednotlivým chybovým hláškám. Přijde mi ovšem, že hlášky jsou celkem výstižné a není potřeba se jim věnovat takto podrobně. V předchozí části jsme si popsali strukturu jednotlivých hlášek a uvedli jsme si jednoduchý příklad jak takové hláška může vypadat. Nyní se zaměříme na netradiční chování Sběratele a co v takových případech dělat.

Chyba vstupního dotazu. Ať už nastane jakákoliv chyba, pokud si uživatel není jistý v kramflecích, kde by mohl být problém, je dobré nejprve zkontrolovat uživatelem zadaný vstupní dotaz. Syntaktické chyby, či dotazování se na neexistující entity může způsobit nepříjemné problémy. Sběratel by měl uživateli být schopen podle hlášky poradit, že se v dotazu vyskytuje nějaký problém. Jeho schopnosti takovou chybu popsat jsou však značně omezené, jelikož je závislý na hláškách získaných z řadičů databáze.

Opakované spouštění jedné exekuce. Pokud je exekuce spuštěna, ale během procesu nastane nějaký problém a Sběratel se snaží exekuci znovu spustit, což opět končí chybou, bývá většinou na vině problém s připojením k instanci. Opakované spouštění se děje kvůli tomu, že server se opakovaně snaží připojit k instanci. Z pohledu uživatele se taková exekuce jeví jako stále běžící. Vypadá to tedy tak, že vyhodnocení běží neúměrné dlouho, pokud šlo předtím o jednoduchý dotaz. Pokud selže, tak to pak zkouší znovu, což vypadá jako opakované spouštění exekuce. Po několika neúspěšných iteracích pak dotaz doběhne a je uložena chyba do persistoru. Pokud není problém s připojením, nebo pokud běží

všechny systémy na jednom stroji, pak doporučuji zkontrolovat si údaje v souboru `application.properties`. Alternativně zapátrat v nastavení komunikace mezi jednotlivými kontejnery, pokud používáte docker pro spouštění aplikace. Obsahem chybové hlášky většinou bývá věta `Connection refused`.

Chybí statistiky pro nová data v PostgrSQL. Pokud během práce s databází nastane problém, kdy skončí vyhodnocení výjimkou s hláškou říkající, že nebylo možné získat nějaká data z datového modelu, mohou být na vině zastaralé statistiky. Prakticky jistotu, že problém nastane, máte, pokud vytvoříte v databázi novou tabulku a přidáte do ní malé množství dat. V takovém případě se statistiky sami neaktualizují a při hledání parametrů, pak Sběratel není schopen dohledat klíčové hodnoty parametrů pro tabulku, protože ve statistikách neexistují. Silně doporučuji toto kontrolovat a po libovolných změnách podobného typu spouštět příkaz `ANALYZE`, který statistiky v databázi aktualizuje. Určitě se vyplatí spouštět tento příkaz pravidelně i bez větších změn, pokud chceme mít přesnější statistiky.

Chybné escapování apostrofů. Pokud využíváte pro testování nástroj curl, pak si dejte pozor na správné escapování apostrofů, hlavně pokud se v dotazu nachází řetězcový literál. Doporučuji pak použít python klienta, který se dle mého názoru v této věci chová předvídatelněji. U něj je ovšem potřeba apostrofy u literálů také escapovat. Zpětná lomítka se pak ale nedostanou do řetězce dotazu, což se v případě curl někdy stává. Pak dostáváme z databáze zprávu o špatné syntaxi.

Chybějící podpora aggregate u MongoDB. V případě, že jste zrovna v situaci, kdy se Sběratel nechová podle vašich představ a zároveň neevidujete žádný problém v zadaném dotazu v jazyce mongosh, vezměte v úvahu, že program nepodporuje práci z dotazy využívající `aggregate`. Podrobnější důvody jsou uvedeny v sekci 4.4. Je důležité si uvědomit, že systém nebude dobře fungovat, ani pokud spouštíte dotaz nad nematerializovaným pohledem, který byl v

Neúplná podpora mongosh. V případě, že pracujete s dotazy pro databázi MongoDB a potýkáte se s problémy, pak je vhodné se zamyslet, zda nepoužíváte některé Sběratelem nepodporované funkce v dotazovacím jazyce mongosh. Když je v dotazu přítomna funkce, jež není v kontextu Sběratele podporována, pak by měl sám Sběratel její přítomnost detekovat a ohlásit skutečnost uživateli. Alternativně může nastat problém, pokud využíváte dané funkce pro konstrukci pohledů. Pokud jde o materializované pohledy, pak by to nemělo působit žádné problémy. Problematická situace nastane, pokud pomocí nepodporovaných funkcí vytváříte klasické pohledy, neboť pak se to promítne do fungování Sběratele. U takových pohledů pak může být problém volat i klasickou funkci `find`. V takovém případě není schopen Sběratel uživateli oznámit, že používá nepodporovanou funkčnost, neboť takovou skutečnost neumí detekovat. Následně se Sběratel snaží zpracovat výsledek, ale velmi pravděpodobně se mu nepovede dohledat všechny informace. Některé parametry tak zůstanou prázdné, v horším případě hrozí, že někde kvůli nenalezení parametru skončí výjimkou. Konkrétní důvody pro nepodporování daných metod jsou uvedeny v sekci 4.4 Dále je ovšem dobré zkontrolovat

správnost zadaných dotazů a zároveň i nastavení parametrů pro připojení, jak již bylo zmíněno u problémů popsaných výše. Mezi Sběratelem nepodporované funkce v jazyce mongosh patří `aggregate`, `count` a `distinct`.

Závěr

Nadešel čas pro shrnutí celé práce. Vyjádříme se k míře naplnění jednotlivých cílů, během čehož se pokusíme shrnout i splnění funkčních a nefunkčních požadavků ze sekce 2.1 Nakonec se budeme věnovat možnostem budoucího rozvoje tohoto softwarového díla.

Prvním z probíraných cílů, ke kterým se vyjádříme, bude C1. Pod tímto cílem bylo očekáváno, že provedeme analýzu databázových metrik a následně vybereme vhodné parametry pro sledování výkonnosti dotazů. Výběr parametrů byl inspirován rešerší existujících monitorovacích systémů a následně doprovázen analýzou o tom, co je v jednotlivých systémech možné získat. Nerad bych zde tvrdil, že výčet vybraných parametrů, které systém aktuálně sbírá, je takto kompletní. Určitě se jedná ovšem o solidní základ pro tvorbu kvalitní datové sady i navazujících prací. Vnímám tedy tento cíl jako splněný.

Dále bylo stěžejním úkolem navrhnout a implementovat kvalitní architekturu pro řešení problému se sběrem parametrů z různých databázových systémů. V úvodu je tento požadavek shrnut pod cílem C2. O architektuře modulu, která je založená na principech OOP, pojednává velká část práce, která je převážně technického charakteru. Navržená architektura pak plní nejen problém potřebný pro splnění zadaného cíle, ale i vybrané funkční a nefunkční požadavky. Program tak parsuje exekuční plány dotazů F5 pro získání potřebných informací. Je schopen získat statistické parametry F2, parametry důležité pro výkonnost dotazů F1 a názvy důležitých databázových entit souvisejících s dotazem F4. Všechny tyto vlastnosti splňuje v kontextu čtecích dotazů F3 a sesbírané logy ukládá do databáze MongoDB N5. Vyjmenované požadavky vnímám jako bližší specifikaci původně vytyčeného cíle, který je tak možné považovat za splněný.

Jeden z důležitých nároků na výslednou architekturu bylo umožnit modulární rozšířitelnost pro podporu dalších databázových systémů, což je zahrnuto pod cílem C3 a nefunkčním požadavkem N1. Pokud bude potřeba přidat podporu pro nový databázový systém, pak je možné v aktuálním návrhu architektury toto udělat prakticky bez zásahu do již napsaného kódu. Zjednodušeně by se dalo tvrdit, že stačí pouze implementovat požadované rozhraní, které server od wrapperů předpokládá. Úprava vyžadující zásah do stávajícího kódu je potřebná pouze v případě rozšiřování podpory pro sběr nových parametrů. U stávajících wrapperů je v takovém případě nutné do-implementovat, jakým způsobem získat daný parametr, pokud nechceme jeho přidání ignorovat, což může být také jedna z možností, jak se k problému postavit. Obecně ale předpokládáme, že takové úpravy nebudou tak časté a bude se jednat o výjimečné případy.

Posledním cílem, kterému odpovídá nefunkční požadavek N2, ke kterému jsme se zatím nevyjádřili, je C4. Aby bylo možné Collector integrovat do stávajícího projektu, byl vyžadován návrh a implementace vhodného REST API pro komunikaci se serverem. Vytvořené rozhraní je podrobněji popsáno v kapitole uživatelské dokumentace v sekci 5.4. Požadované rozhraní pro komunikaci je implementované pomocí frameworku SpringBoot, čímž byl splněn další z nefunkčních požadavků práce N4.

Ačkoliv jsme prošli všechny v úvodu vytyčené cíle práce, zbývá jeden nefunkční požadavek N3, o kterém jsme se doposud nebavili. Nebyla tedy zatím

možnost vyjádřit se k míře jeho splnění, což napravíme v tomto odstavci. Požadavek specifikuje, že je nutné implementovat pro modul wrappery pro databáze PostgreSQL, Neo4j a MongoDB. Pokud se bavíme o představitelích relační a grafové databáze, pak se nebojím tvrdit že byl požadavek splněný na slušné úrovni jejich podpory. Bohužel to samé nemůžeme tvrdit o představiteli dokumentové databáze MongoDB. Sběratel nepodporuje práci s agregační funkcí nad databázovými kolekcemi. Podrobnosti o daném problému a důvody, proč jsem se prozatím rozhodl omezit podporu dotazovacího jazyka *mongosh* jsou popsány v sekci 4.4. V kontextu MongoDB tedy nemůžeme říci, že bychom požadavek plnohodnotně splnili. Wrapper pro systém MongoDB ale je součástí aplikace a měl by spolehlivě fungovat pro všechny dotazy používající *mongosh* metodu `find`. Podporu pro základní dotazy v systému MongoDB tak Collector podporuje. Toto by mělo posloužit i jako solidní základ pro další možnost rozšíření podpory pro práci s databází MongoDB, kterému se budeme podrobněji věnovat v odstavcích níže.

Budoucí práce

Jak již bylo zmíněno na začátku, budeme se věnovat způsobům, jak je dále možné rozvíjet vytvořený modul. Nejprve se pozastavíme nad využitím datové sady a následně se pak budeme věnovat jednotlivým směrům, kterými by bylo možné modul posunout dál.

Využití datové sady. Kromě rozvoje samotného modulu je možné dále pracovat na nástrojích, které budou využívat data z Collectorem vytvořené datové sady. Jedná se především o nástroje představené a probrané v úvodu práce. Například zmíněný MM-evocat, do kterého bude potřebné Collector vhodně integrovat. Dále pak nástroje napomáhající s optimalizací dotazů napříč různými datovými modely založené na strojovém učení a podobných metodách.

MongoDB podpora aggregate. Téma, které bylo již několikrát zmíněno v kontextu celé práce. Databáze MongoDB si do budoucna bezesporu zaslouží plnohodnotnou podporu dotazovacího jazyka *mongosh*. Pevně věřím, že problémy, s nimiž jsem se během práce setkal, se podaří překonat a umožní se tak plnohodnotná podpora funkcí `aggregate`, `count` a `distinct`. Úkol, ačkoliv se může zdát celkem triviální, tak bohužel musíme říci, že v tomto případě je opak pravdou. Nástroj `aggregate` je ovšem v kontextu databáze velmi silný a určitě by byla škoda, kdyby se Collector v tomto směru neposunul dál.

Podpora dalších databázových systémů. Systém je navržen tak, aby bylo možné jej snadno rozšiřovat o podporu dalších databázových systémů. Jsem přesvědčen, že kvalitě výsledných dat ve tvořené datové sadě jistě pomůže větší rozmanitost systémů. Je možné, že existují oblasti, kde aktuálně podporovaná databáze z rodiny systémů pro daný datový model se nechová úplně nejefektivněji, ale jiný reprezentant z rodiny by se pro daný problém choval lépe. K odstranění pochybnostem tohoto rázu by pak větší rozmanitost systémů jistě pomohla. Jistě je možné rozšiřovat podporu pro nové systémy i dalším směrem a to podporou pro datové modely, které zde zatím nemají zastoupení. Jedná se například o sloupcové databáze, vyhledávací enginy apod.

Implementace grafického rozhraní. Modul Collector sbírá data, které následně ukládá pro další použití. Bohužel kromě REST-ového rozhraní pro komunikaci se serverem nedisponuje žádným nástrojem pro vizualizaci nasbíraných dat. Přijde mi, že pro účely testování či analýzy sbíraných parametrů, by grafické rozhraní pro práci s modulem s možností vizualizace nasbíraných dat, dávalo dobrý smysl. Nástroj by mohl využívat stávající rozhraní poskytované modulem, alternativně by se toto rozhraní v případě potřeby dalo rozšířit pro lepší možnosti agregace dat. Dávalo by pak smysl se zamyslet nad tím, zda by nebylo vhodné implementovat tento nástroj i pomocí vlastního backendu, který by se rovnou připojil k databázi s nasbíranými logy a měl tak k dispozici větší spektrum dotazů. Zároveň by se tímto způsobem ulevilo i samotnému Collectoru, jelikož by na něj následně bylo adresováno menší množství požadavků. Otázku bych rád nechal otevřenou a nechal tak prostor pro zamyšlení či detailnější analýzu, která by mohla najít pro tento problém uspokojivou odpověď.

Seznam použité literatury

[1] Irena Holubová, Pavel Koupil, and Jiaheng Lu. Self-adapting design and maintenance of multi-model databases. In Bipin C. Desai and Peter Z. Revesz, editors, *IDEAS'22: International Database Engineered Applications Symposium, Budapest, Hungary, August 22 - 24, 2022*, pages 9–15. ACM, 2022.

[2] Filip Mihál', Jáchym Bártík, and Pavel Koupil. Refining storage strategy through index selection methods in multi-model database systems: A survey. *Available at SSRN 4536677*, 2023.

[3] Pavel Koupil, Jáchym Bártík, and Irena Holubová. Model-agnostic evolution management. In Janis Grabis, Tanja E. J. Vos, María José Escalona, and Oscar Pastor, editors, *Research Challenges in Information Science - 19th International Conference, RCIS 2025, Seville, Spain, May 20-23, 2025, Proceedings, Part I*, volume 547 of *Lecture Notes in Business Information Processing*, pages 245–261. Springer, 2025.

[4] Zabbix SIA. Zabbix Documentation. https://www.zabbix.com/documentation/6.0/en/manual, [Online, Accessed 16 Jul 2025].

[5] Prometheus Authors. Prometheus Documentation. https://prometheus.io/docs/introduction/overview/, [Online, Accessed 16 Jul 2025].

[6] The PostgreSQL Global Development Group. PostgreSQL Documentation. https://www.postgresql.org/docs/14/index.html, [Online, Accessed 16 Jul 2025].

[7] Neo4j, Inc. Neo4j Documentation. https://neo4j.com/docs/operations-manual/5/, [Online, Accessed 16 Jul 2025].

[8] MongoDB, Inc. MongoDB Documentation. https://www.mongodb.com/docs/v6.0/, [Online, Accessed 16 Jul 2025].

Seznam obrázků

2.1 Případy užití 14

3.1 Architektura modulů 30

4.1 Server diagram tříd 38
4.2 Modul datového modelu 44
4.3 Wrappers diagram tříd 48
4.4 Persistor diagram tříd 51
4.5 Sekvenční diagram exekuce dotazu 52
4.6 Sekvenční diagram sběru parametru 53

Seznam tabulek

1.1 Srovnání monitorovacích technologií 10

2.1 Srovnání parametrů pro databázi. 16
2.2 Srovnání parametrů pro kind. 18
2.3 Srovnání parametrů pro index. 22
2.4 Srovnání parametrů pro atribut. 24
2.5 Srovnání parametrů pro datový typ atributu. 26

Seznam použitých zkratek

API Application Programming Interface. 4, 34, 35, 37, 85

APOC Awesome Procedures On Cypher. 63, 65, 67

CSV Comma-Separated Values. 72

GPLv2 General Public License version 2. 5, 9, 10

HA High Availability. 6

HTTP Hypertext Transfer Protocol. 7, 35, 73

ILP Integer Linear Programming. 3

JDBC Java Database Connectivity. 34, 35, 56

JDK Java Development Kit. 67, 69

JMX Java Management Extensions. 9

JRE Java Runtime Environment. 67

JSON JavaScript Object Notation. 31, 36, 65, 73, 74

NoSQL Not only SQL. 34, 62

OOP Object-Oriented Programming. 85

RAM Random-Access Memory. 35

REST Representational State Transfer. 4, 12, 13, 31, 36, 39, 73, 75, 85, 87

SQL Structured Query Language. 34, 35, 71

XML eXtensible Markup Language. 36