Posudek bakalářské práce

Matematicko-fyzikální fakulta Univerzity Karlovy

Autor práce	Josef Holubec
Název práce	Sběratel informací o výkonu dotazování v multimodelových databázových systémech
Rok odevzdání	2025
Studijní program	Informatika
Specializace	Databáze a web

Autor posudku	Jáchym Bártík	**Oponent**	
Pracoviště	Katedra softwarového inženýrství		

Cílem bakalářské práce byla implementace nástroje pro získávání metadat a statistik o výkonu dotazů v různých databázových systémech (s odlišnými datovými modely) – konkrétně PostgreSQL, MongoDB a Neo4j (relační, dokumentový a grafový model). Výsledkem je webová aplikace, na kterou je možné posílat dotazy. Ta na jejich základě sestrojí vlastní dotazy, které vykoná nad příslušnými databázemi a výsledky vrátí uživateli.

K celé práci	**lepší**	**obvyklé**	**horší**	**nevyhovuje**
Obtížnost zadání	☐	■	☐	☐
Splnění zadání	☐	☐	■	☐
Rozsah práce	☐	☐	■	☐

V první kapitole autor provádí rešerši dvou existujících systémů (Zabbix a Prometheus), které mají sice trochu jiný účel, ale jsou alespoň vzáleně podobné.

Ve druhé kapitole definuje funkční a nefunkční požadavky, uživatele a use casy. Dále rozebírá parametry, které má nástroj z databází získávat.

Ve třetí kapitole krátce představí architekturu aplikace a dále se věnuje použitým návrhovým vzorům a technologiím.

Ve čtvrté kapitole několikrát popisuje třídy – nejdřív obecně, potom jak dají rozšířit a nakonec z hlediska implementace.

V poslední, páté kapitole ukazuje instalaci, použití a řešení chybových stavů.

Práce technicky splňuje zadání, avšak kvalita textu i kódu je na velmi nízké úrovni. Společným jmenovatelem nedostatků textové části je, že autor pouze popisuje *co* udělal, ale už nijak nezdůvodňuje *proč* to tak udělal. Nadprůměrný rozsah textu je způsoben zdlouhavými vyjádřeními a popisem věcí, které by tam ani neměly být.

Implementační část zase sráží přílišná míra abstrakce, která je v mnoha případech spíše na škodu, a některá kontroverzní rozhodnutí (viz níže).

Z těchto důvodů musím práci hodnotit jako **neprospěl(a)**. Věřím ale, že po důkladném přepracování by mohla být bez problémů obhájena.

Textová část práce	lepší	obvyklé	horší	nevyhovuje
Formální úprava	☐	☐	☐	■
Struktura textu	☐	☐	☐	■
Analýza	☐	☐	☐	■
Vývojová dokumentace	☐	☐	■	☐
Uživatelská dokumentace	☐	■	☐	☐

Práce obsahuje značné množství gramatických, typografických a stylistických chyb. Časté jsou chyby v čárkách a větné skladbě. Autor pravidelně přechází z autorského plurálu do jednotného čísla a zpět. Také používá hovorové a pro odbornou práci zcela nevhodné výrazy. Text působí velmi zdlouhavě, mnoho částí je redundantních či úplně zbytečných. Předposlední odstavec na stránce 83 měl být zřejmě nahrazen tím posledním, ale autor jej pravděpodobně zapomněl smazat. Ve výsledku se práce velmi špatně čte a je těžké se v ní orientovat.

Rešerše existujících nástrojů je v kontextu práce zcela zbytečná, autor na ni ve zbytku textu nijak nenavazuje. Navíc se z ní ani nedozvím, co dané systémy vlastně dělají. Zato se dozvím kdo je založil, v jakém městě sídlí jejich společnosti a podobné nerelavantní informace. Např. u Zabbixu je popsáno, jaké všechny varianty SQL databází může použít pro ukládání vlastních dat – ale to hlavní, tj. *jaké databázové systémy umí monitorovat*, už zmíněno není.

Práce obsahuje výsledky analýzy, např. funkční požadavky či které parametry chceme sbírat, ale nijak se nezabývá tím, jak k nim autor došel, proč si vybral zrovna tyto požadavky a parametry, jaké jsou případné alternativy atd. Funkční požadavky by měly být konkrétní a měly by obsahovat slovo „musí“ (či jiné s podobným významem). Pro ilustraci, toto je funkční požadavek **F5**:

„Analýza a zisk parametrů z exekučního plánu dotazu.“

Většina nefunkčních požadavků naopak spadá do funkčních, např. **N3**:

„Implementace wrapperů pro databázové systémy PostgreSQL, MongoDB a Neo4j.“

Očekával bych podrobnější rozbor měřených parametrů, zejména v kontextu různorodosti podporovaných systémů. K čemu je např. parametr `CacheSize` u databáze MongoDB, která může být distribuována na více serverech? Má každá instance vlastní cache? Měří tento parametr celkovou velikost? Neměli bychom potom měřit i počet běžících instancí? Distribuovanost je jednou z klíčových vlastností MongoDB a dalších NoSQL databází, přitom v práci není nijak zmíněna a v implementaci se s ní nepočítá. Dále, v MongoDB se povinné atributy hledají podle validačního JSON schématu. Proč nelze tento přístup použít i pro zjištění názvů a typů atributů (místo procházení všech objektů v do hloubky)?

Kromě analýzy v práci chybí i návrh řešení. Třetí kapitola, nazvaná „Návrhová dokumentace“, je tvořena popisem modulů a jejich rozhraní, návrhových vzorů, a technologií. Pouze část o návrhových vzorech obsahuje zdůvodnění rozhodnutí autora. Ostatní části jsou čistě popisné.

Programátorská dokumentace je nevhodně strukturována. Mnohé třídy jsou vysvětlovány vícekrát. Například o `ExecutionContext` se mluví v sekcích **4.1.3** a **4.3.2**. První sekce by měla být obecný popis třídy a druhá návod, jak ji potenciálně rozšířit. Nicméně obě mají mnoho společného a v jejich záměru prakticky nevidím rozdíl. Dále, podstatná část textu je jen výčet metod s triviálním komentářem, např.:

„`setPageSize` – nastaví velikost jedné stránky v bytech.“
Toto patří do javadocu, ne do textu práce. (Odbočka k implementaci: v kódu naopak příslušný javadoc chybí.)

V závěru dokumentace jsou zmíněny testy, resp. to, že žádné nejsou. Souhlasím s autorem, že v tomto případě nedávají unit testy příliš smysl. Není na škodu, že aplikace byla testována ručně, ale nejsou popsány žádné testovací scénáře. Píše se o existenci testovací datové sady, ale už ne o tom, kde se dá najít.

Uživatelská dokumentace je podrobná a srozumitelná. Nicméně při popisu instalace by stačilo uvést příkazy, které je třeba spustit. Není nutné uvádět i jejich výstup.

Chválím použití dockeru. Bohužel, instalace vyžaduje manuální kompilaci, až poté je možné spustit docker. To ji výrazně komplikuje, musím si např. ručně nainstalovat konkrétní verzi Javy. Přitom hlavní kontejner obsahuje vše potřebné – proč tedy neprobíhá kompilace v něm? Dále je vhodné dát kontejnerům unikátní názvy, aby nekolidovaly s jinými kontejnery běžícími na stejném stroji. Např. `neo4j` není moc unikátní.

Není nutné popisovat každý příkaz z dockerfile, docker má vlastní dokumentaci. Obsah celého konfiguračního souboru pro docker-compose rozhodně nepatří do textu práce.

Implementační část práce	**lepší**	**obvyklé**	**horší**	**nevyhovuje**
Kvalita návrhu	☐	☐	☐	■
Kvalita zpracování	☐	☐	■	☐
Stabilita implementace	☐	☐	■	☐

Kód je na úrovni jednotlivých tříd celkem čitelný a rozumně strukturovaný. Zásadním problémem je ale příliš velká míra abstrakce na úrovni jednotlivých modulů, která výrazně snižuje srozumitelnost a rozšiřitelnost. Konkrétní příklady jsou popsány níže.

Nechápu architekturu serveru. V použitém frameworku (Spring Boot) existuje jasná hierarchie `Controller-Service-Repository`. Tady je jeden `Controller`, několik různých `Component` a `Persistor`, který ale nijak nevyužívá standardní Spring Boot `Repository`. Autor tak musí spoustu věcí implementovat sám, místo aby využil existující (a pravděpodobně lepší) řešení.

Model pro získané statistiky je zbytečně složitý. Datová struktura `QueryData` je jednoduchý objekt, jehož properties jsou buď základní typy či další jednoduché objekty. Neobsahuje žádnou logiku, slouží pouze pro přenos dat. Nicméně všechny properties jsou `private` a zároveň mají `public` setter (většina i getter). Tj. na nich kdokoli může cokoli měnit, potom ale nemusí být `private`.

Celá tato struktura je pak skryta za fasádou `QueryDataModel`, která implementuje rozhraní `DataModel`, se kterým pracuje zbytek aplikace. Jinak řečeno, v jednoduchém DTO je vše skryto za třemi vrstvami abstrakce a ještě settery. Pokud bych chtěl cokoli změnit, např. přidat další parametr do modelu, musel bych to udělat na minimálně čtyřech různých místech.

Jádrem business logiky aplikace je abstraktní třída `AbstractWrapper`, implementující rozhraní `Wrapper` (skutečně to není naopak). Ta má prakticky jediný účel – volá se na ní metoda `executeQuery`. Ta má 20 řádků (připojí se na databázi, naparsuje dotaz, provede dotaz, atd. –

vše implementováno pomocí abstraktních metod). Ze třídy dědí wrappery pro konkrétní databáze, které tyto metody implementují. Všechny tyto metody, všechny třídy wrapperů, a všechny komponenty, ze kterých se skládají, potom musí používat jeden až tři generické parametry (`<TResult, TQuery, TPlan>`). A stejně se na několika místech nevyhneme explicitnímu přetypování. Mnohé metody ani nejsou potřeba v každém databázovém systému. Je toto nutné? Nestačilo by, kdyby si každý wrapper sám implementoval metodu `executeQuery`? Ušetření těch 20 společných řádků opravdu nestojí za ohromnou složitost současného řešení.

Dále, wrappery žijí po celou dobu běhu aplikace. Výše popsané metody si ale musí předávat informace – takže si všechny předávají `ExecutionContext` (přičemž některé jej potom ani nevyužívají). Proč se prostě nevytvoří nová instance wrapperu pokaždé, když se volá `executeQuery`? Každý by si pak mohl sám pamatovat vše, co potřebuje. Zároveň bychom se nemuseli bát, že když si wrapper něco uloží, tak to omylem ovlivní další dotaz.

Tento problém je dobře vidět na sekvenčních diagramech v textu práce. První ukazuje, jak `AbstractWrapper` pouze po řadě provolává všechny abstraktní metody v průběhu `executeQuery`. Ve druhém diagramu vidíme `AbstractDataCollector`, který se vytvoří v průběhu `executeQuery` a dostane `ExecutionContext` jako argument. Následně na něm v sekvenčním diagramu něco volá, potom volá další třídy, kterým opět předává `ExecutionContext`, takže i ty na něm něco volají. Přitom volá metody na komponentách `AbstractConnection` a `DataModel`, které jsou, jako prakticky všechno, celou dobu součástí `ExecutionContext`.

Vstupní dotazy jsou do programu předány přes webové rozhraní, tedy se jedná o `String`. Nicméně MongoDB driver potřebuje dotaz jako `Document`, takže je nutné jej parsovat. Nejlepší řešením by bylo použít existující přístup, který to umí. V tomto případě to není tak přímočaré, protože dotaz je vlastně kód v JavaScriptu, na který se v programu mongosh (MongoDB shell) zavolá `eval`. Pořád je ale možné zavolat externí proces (např. právě mongosh, či vlastní JavaScriptiový program), který dotaz převede na některou serializaci BSONu, se kterým už java dokáže pracovat.

Dalším řešením by bylo omezit uživatelský vstup na serializace BSONu, které java podporuje. Pokud by ani to nešlo, sestavil bych gramatiku a použil některý z volně dostupných nástrojů, např. ANTLR, na vygenerování parseru.

Určitě bych se vyhnul manuálnímu parsování dotazu pomocí spousty vnořených if podmínek (viz třída `ArgumentsArray`). To je extrémně nepřehledné, nerozšiřitelné a náchylné na chyby. Dalším důsledkem je podpora pouze omezené podmnožiny jazyka. Autor zmiňuje i další důvod omezené podpory, a sice nevhodně navržené rozhraní pro exekuci dotazů. Bohužel, z textu práce mi není jasné, včem přesně je problém.

Aplikaci se mi podařilo zkompilovat a spustit. Přitom ale vypisovala chyby (chybová hláška začínala na `com.mongodb.MongoSocketException: mongodb` – nejspíš tedy nefunguje připojení k MongoDB). Podařilo se mi vytvořit a spustit exekuci, nicméně už nedoběhla (zůstala ve stavu `Running`).

Po chvíli jsem zjistil, že kontejner `mongodb` se neustále restartuje. Důvodem jsou pravděpodobně špatně nastavená práva na volume kam si MongoDB ukládá data (řádek 24), protože po jeho zakomentování už všechno funguje správně. Nicméně potom už data nejsou perzistentní.

Celkové hodnocení	Neprospěl(a)
Práci navrhuji na zvláštní ocenění	Ne

Datum 25.8.2025 Podpis