Paralelní programování a jeho dopad na databázové systémy

Bc. Lukáš Juřina

Diplomová práce
2017

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2016/2017

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Lukáš Juřina**
Osobní číslo:	**A15164**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**prezenční**
Téma práce:	**Paralelní programování a jeho dopad na databázové systémy**
Téma anglicky:	**Parallel Programming and its Impact on Database Systems**

Zásady pro vypracování:

1. **Proveďte analýzu způsobů, jakými lze zpracovat více požadavků na databázi.**
2. **Popište metody detekce deadlocků a jejich řešení z pohledu databáze.**
3. **Proveďte analýzu návrhových metod a modelů databáze pro paralelní přístup.**
4. **Navrhněte databázi pro paralelní přístup.**
5. **Vytvořte zátěžový test na navrhnutou databázi a prezentujte dosažené výsledky.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **BEN-GAN, Itzik.T-SQL Fundamentals. 3rd. USA: Microsoft Press, 2016. ISBN 978-1509302000.**
2. **CELKO, Joe.Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Non-Relational Databases. USA: Elsevier, 2014. ISBN 978-0124071926.**
3. **DAVIDSON, Louis.Exam Ref 70-762 Developing SQL Databases. USA: Pearson Education, 2017. ISBN 978-1509304912.**
4. **Parallel Programming in the .NET Framework.Microsoft Developer Network [online]. [cit. 2017-01-30]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx**
5. **SQL Server Technical Documentation.Microsoft TechNet [online]. [cit. 2017-01-30]. Dostupné z: https://technet.microsoft.com/en-us/library/ms130214.aspx**
6. **Microsoft SQL Server 2016: A Beginner's Guide. 6th edition. USA: McGraw-Hill Education, 2017. ISBN 978-1259641794.**

Vedoucí diplomové práce:	**Ing. Milan Navrátil, Ph.D.** Ústav elektroniky a měření
Konzultant:	**Ing. Michal Hájek** Ústav informatiky a umělé inteligence
Datum zadání diplomové práce:	**3. února 2017**
Termín odevzdání diplomové práce:	**16. května 2017**

Ve Zlíně dne 3. února 2017

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Jméno, příjmení: Lukáš Juřina

Název bakalářské/diplomové práce: Paralelní Programování a jeho dopad na databázové systémy

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 11.5.2077

Juřina
.....
podpis diplomanta

ABSTRAKT

Práce se zabývá návrhem databáze přizpůsobené k paralelnímu přístupu aplikací. V teoretické části jsou popsány možné modely návrhu takové databáze a jejich rozdíly. Dále se práce zaobírá možnostmi pro zajištění požadovaného chování transakcí prováděných nad databází a výhody a nevýhody těchto chování. Dále je proveden popis možných situací ústících v deadlock. Jak zjistit že k takové situaci došlo a jak deadlocku předcházet či jej řešit. V praktické části je navržena testovací databáze a vytvořena aplikace zátěžového testu. Za využití této aplikace jsou poté nasbírána data a následně interpretovány získané výsledky.

Klíčová slova: MS SQL, izolační úrovně, deadlock, ASP .NET, databáze

ABSTRACT

This master thesis deals with database design suited for parallel access of applications. In theory part design patters for such database and their differences are described. Afterwards work is concerned with options for guarantying needed behaviour of transactions executed on database and advantages and disadvantages of such behaviour. Afterwards possible situations resulting in deadlock is described. How to find out that such situation has arisen and how to prevent deadlock or how to solve it. In practical part test database is designed and benchmark application is developed. With the use of developed application data are collected and followed by interpreted results.

Keywords: MS SQL, isolation levels, deadlock, ASP .NET

PODĚKOVÁNÍ

Chtěl bych poděkovat mojí rodině, za to, že mi umožnila studovat a rozvíjet moje znalosti, přátelům, jež mi pomáhali udržet chladnou hlavu a dobrou náladu a v neposlední řadě bych chtěl poděkovat Ing. Michalu Hájkovi za odborné vedení práce.

OBSAH

ÚVOD

Moje diplomová práce se zabývá dopadem paralelního programování na databázi, jinak řečeno jak navrhnout databázi pro velký počet přístupů v krátkém okamžiku a přitom neztratit výsledek jedné z operací nebo uživateli vrátit data, která by mohla způsobit špatné rozhodnutí.

V současné době je rostoucí poptávka po webových aplikacích dostupných přímo z prohlížeče osobního počítače nebo mobilního zařízení. S tím je nutné počítat již při návrhu aplikace a přizpůsobit také uložiště dat.

Při návrhu databáze podporující webovou aplikaci je potřeba vzít v úvahu s jak velkým objemem dat se bude běžně operovat. Jaký počet záznamů v jednotlivých tabulkách databáze může být a jak často se budou přidávat další záznamy, jak často data budou modifikována a jak často budou čtena.

Jaké dotazy budou na databázi posílány a v jakém měřítku ji budou zasahovat. Bude se jednat o jednoduché vytažení jednoho záznamu podle jeho indexu nebo se bude jednat o složitý dotaz, který pojí na jednu tabulku několik dalších tabulek a v závislosti na výsledném záznamu se rozhodne, jestli provede aktualizaci existujícího záznamu či budou do několik tabulek vloženy nové záznamy.

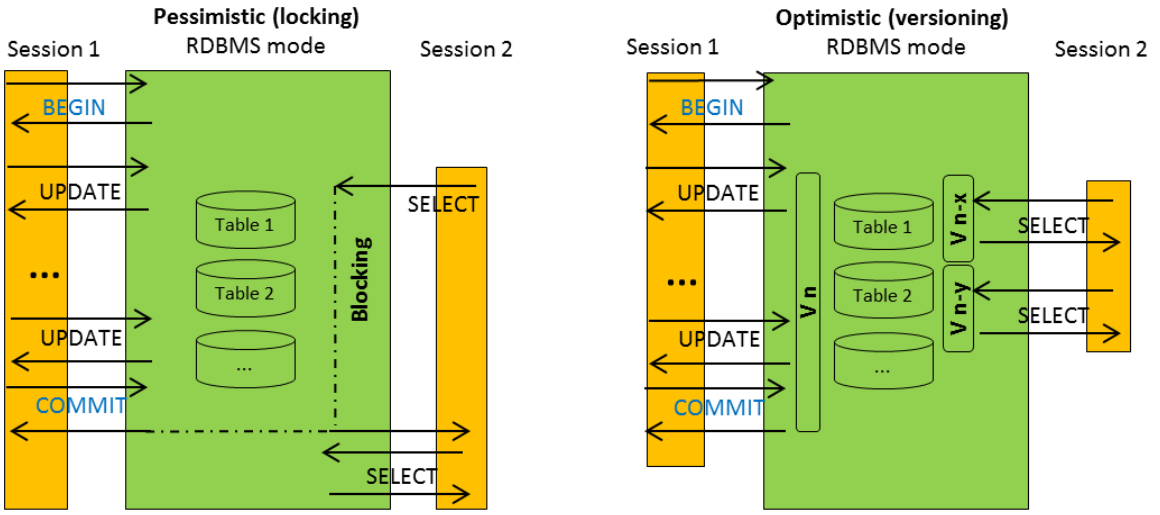
Také je nutné vzít v potaz, jak zajistit konzistenci a validitu dat neboli jak kontrolovat tok dat v databázi. Za jakých okolností je nepřípustné aby dotaz mohl číst zastaralá nebo nepravdivá data a kdy naopak je nezbytné, aby dotaz pracoval s aktuálními daty a za žádných okolností nebyl ovlivněn dalšími dotazy, i za cenu odložení ostatních dotazů či jejich opakovaného vykonání.

Při práci s velkým objemem dat nebo velkým počtem dotazů nad databází mohou vznikat konfliktní situace, dotazy mohou vracet nepravdivé informace anebo dokonce neproběhnout vůbec, protože jsou jiným dotazem zablokovány na dobu delší, než jim dává aplikace pro vykonání a také se dotazy mohou dostat do situace zvané „uváznutí“ (z anglického „deadlock“).

I. TEORETICKÁ ČÁST

1 NÁVRHOVÉ MODELY

Stupeň souběžnosti přístupu k datům, který je schopna jednotlivá instance SQL Serveru podporovat závisí na zvoleném návrhovém modelu SQL Serveru a do určité míry také na aplikační logice zapracované programátorem. [7]



Obr. 1. Pesimistický a optimistický návrhový model [9]

1.1 Pesimistická souběžnost

Je usouzeno, že existuje dostatek souběžných operací modifikující data v systému a mohou probíhat v jeden okamžik, tedy vzniknou problémy, které povedou na ztrátu integrity dat. Ztrátě integrity dat se zabraňuje přidělením zámků operacím čtení a zápisu. Při čtení dat je čtenáři přidělen sdílený zámek, proto žádné jiné sezení nemůže data modifikovat, ale může požádat o další sdílený zámek a data číst. Pro operace modifikace je vyžadován exkluzivní zámek, takže v průběhu modifikace dat s exkluzivním zámkem, není možné, aby na stejné data dostala operace čtení sdílený zámek a data si mohla přečíst, dokud nebude dokončena modifikace dat.

Jinými slovy v pesimistickém modelu souběžnosti čtenáři blokují zapisovatele a zapisovatelé blokují čtenáře.

V pesimistické souběžnosti se využívá izolačních úrovní READ COMMITTED, REPEATABLE READ a SERIALIZABLE. [7]

1.2 Optimistická souběžnost

Je opakem pesimistické souběžnosti. Je usouzeno, že je dostatečně málo konfliktních data modifikujících operací v systému, takže je nepravděpodobné že jedna transakce bude měnit data ve chvíli, kdy jsou měněna jinou transakcí. Výchozí chování optimistické souběžnosti je využít technologie verzování řádků, která je dostupná přes izolační úroveň založenou na takzvaných SNAPSHOT.

Když je využívána jedna z izolačních úrovní založených na verzování řádků, SQL Server si uchovává časově oražené verze v tempdb databázi, které obsahuje všechny předtím spáchané verze od začátku nejstarší otevřené transakce.

Pokud transakce narazí na exkluzivní zámek nad daty, které potřebuje přečíst, raději než by čekala na uvolnění exkluzivního zámku, tak rychleji a jednodušeji si požádá o poslední validní stav těchto dat zaznamenaný v tempdb.

Při využívání izolačních úrovní, založených na verzování řádků, operace SELECT nežádají o sdílené zámky, místo toho čtou přímo data, podle toho kdy začala transakce nebo dotaz, z verzí v tempdb a tím se vyhne zablokování operací modifikujících data.

Zapisovatelé blokují a nadále budou blokovat zapisovatele, což může vést k dalším konfliktům. [7]

2 ZÁMKY

Zamykání je aktivita, ke které dochází, když sezení SQL Serveru přebírá „vlastnictví“ nějakého zdroje předcházející nějaké práci s tímto zdrojem, například čtení nebo aktualizace.

Zamykání je pouze logický koncept, vytvořen pro podporu ACID vlastností transakcí, aby data zůstala konzistentní. Ovšem proto, že zamykání je pouze logický koncept, který není založen na fyzických vnitřních potřebách databázového systému, tvůrci databázového systému mají velké pole možností jak přesně implementovat zamykání v jejich systému, a tím také jaký dopad to bude mít na využití zdrojů uvnitř systému. [7]

2.1 Módy zámků

SQL Server používá několik typů zámků, které se označují jako módy zámku. Tyto módy zahrnují sdílené zámky, exkluzivní zámky, aktualizační zámky a zámky záměru. Tyto čtyři ANSI módy jsou nutné pro izolaci transakcí. Každý mód zámku specifikuje jak je omezující pro ostatní akce a jaké akce jsou možné, dokud je zámek s takovým módem držen. [7]

2.1.1 Sdílený zámek

Ve výchozím nastavení SQL Server dostane sdílený zámek („S“ zámek) automaticky při čtení dat. Tabulka, stránka, individuální řádek v tabulce nebo index mohou držet S zámek. K podpoře izolační úrovně SERIALIZABLE, SQL Server může S zámek použít na řadu indexovaných řádků.

Jak název napovídá, více než jeden proces může držet S zámek nad stejnými daty, ale žádný proces nemůže získat exkluzivní zámek nad daty zamčenými S zámkem. Výjimkou je pokud o exkluzivní zámek žádá proces, který nad stejnými daty drží S zámek a žádný jiný proces nad těmito daty nedrží S zámek.

Vlastnosti S zámků se liší podle izolačních úrovní. Většinou se mění doba držení zámku nad daty, kdy u úrovní REPEATABLE READ a SERIALIZABLE je doba držení zámku prodloužena po dobu trvání celé transakce. [7]

2.1.2 Exkluzivní zámek

SQL Server automaticky dostane exkluzivní zámek („X“ zámek), pokud se jedná o data modifikující operaci, jako je INSERT, UPDATE nebo DELETE operace. Pouze jedna transakce v daný čas může vlastnit X zámek nad určitými datovým zdrojem a tento zámek jí zůstává po dobu trvání celé transakce. Tyto data jsou obvykle nedostupná pro ostatní procesy, dokud transakce držící X zámek není dokončena a spáchána, nebo nastane takzvaný „rollback“ neboli zpětné vrácení všech změn.

Výjimku tvoří izolační úroveň READ UNCOMMITTED, která umožňuje transakci číst data uzamčena X zámkem jiné transakce. [7]

2.1.3 Aktualizační zámek

Aktualizační zámky („U“ zámek) nejsou příliš separátním druhem zámku, ale spíše hybridním typem zámku mezi S a X zámky. Transakce dostane U zámek, pokud SQL Server hodlá provést data modifikující operaci, ale předtím potřebuje provést hledání, aby našel zdroj k modifikaci (například řádek dat).

SQL Server nepotřebuje použít X zámek na řádek, dokud není připraven k provedení jeho modifikace, ale zároveň SQL Server potřebuje použít nějaký zámek pro interval prohledávání dat, aby ochránil již nalezené data před modifikací od jiné transakce. Proto SQL Server použije nad nalezenými řádky U zámek a v případě, že řádek odpovídá podmínkám, je U zámek konvertován na X zámek. [7]

2.1.4 Zámek záměru

Zámky záměru nereprezentují rozdílný způsob zamykání. Termín „záměr“ je kvalifikátor k předchozím módům. Jinak řečeno může být použit záměr sdíleného zámku („IS“ zámek), záměr exkluzivního zámku („IX“ zámek) nebo záměr aktualizačního zámku („IU“ zámek).

SQL Server může získat zámek na různých úrovních zrnitosti (řádková, stránková, tabulková úroveň), proto SQL Server potřebuje nějaký mechanizmus, který signalizuje zdali je část zdroje už uzamknutá nějakou transakcí.

Například pokusí-li se jedna transakce uzamknout tabulku, SQL Server musí být schopen určit, jestli je už uzamčena stránka nebo nějaký řádek této tabulky. Zámky záměru slouží právě k tomuto účelu. Vždy když transakce získá zámek na nižší úrovni zrnitosti, také dostává zámky záměru na vyšší úroveň granulity toho samého objektu. Například transakce

drží zámek X nad řádkem v tabulce zákazníků, ale také drží zámek záměru IX zámek na stránku tabulky, která obsahuje řádek daného zákazníka a také IX zámek na celou tabulku zákazníků. Takto použité zámky záměru zabrání jiné transakci v získání X zámku na celou tabulku. [7]

2.2 Zámek speciálního záměru

K zámkům záměru má SQL Server k dispozici další tři zámky, které mají spíše povahu konvertujících zámků. SQL Server získá tyto zámky, když zámek bez záměru je vyžádán na zdroj, který již drží IX nebo IU zámek.

V případě rozhodování SQL Serveru, který zámek bude udělen, vyhrává ten silnější a nahrazuje slabší. Například pokud stránka drží IU zámek a poté je zažádáno o IX zámek, IX zámek jednoduše nahradí IU zámek. [7]

2.2.1 Sdílený záměr exkluzivní (SIX)

Pokud SQL Server má aspoň jeden řádek uzamčen X zámkem, stránky a tabulka, která obsahuje tento řádek, dostanou IX zámek.

Pokud stejná transakce provede operaci, která vyžaduje S zámek, SQL Server dostane SIX zámek na tabulku. [7]

2.2.2 Aktualizační záměr exkluzivní (UIX)

SQL Server nikdy nezíská U zámek na úrovni tabulek, takže jediná možnost jak dostat U zámek a IX zámek dohromady je na úrovni stránek.

Například když spustíme UPDATE výraz první, a protože je aktualizován řádek, řádek dostane X zámek, a stránka i tabulka dostanou IX zámek. Když je poté spuštěn SELECT výraz, s příznaky nutící U zámky na stránce k přístupu, U zámky na stránce se skombinují s předešlým IX zámkem na stránce a vznikne UIX zámek. [7]

2.2.3 Sdílený záměr aktualizační (SIU)

SQL Server drží IU zámky pouze na úrovni stránek a odpovídající tabulka má IX zámek. SIU zámek získáme například spuštěním dotazu, který získá U zámek nad nějakým řádkem, a tím také získáme UI zámek na stránku tabulky. Pokud v té samé transakci získáme S zámek na tu samou stránku tabulky, výsledkem bude SIU zámek. [7]

2.3 Zámky klíčového rozsahu

Zámek klíčového rozsahu je spojen se specifickým indexovým klíčem, ale zahrnuje rozsah možných hodnot menších nebo stejných jako je klíč, se kterým je zámek vázán, a hodnotami většími než je klíč v listové úrovni. Jinak řečeno je zámek klíčového rozsahu množina napříč rozsahem mezi dvěma klíči, obsahující konečný klíč, ale vynechává počáteční klíč.

Například rozsahový zámek nad jmény „Aneta“ a „Gabriela“ by obsahoval jména větší než „Aneta“ a jména menší než a rovna jménu „Gabriela“.

SQL Server může držet devět různých módů zámku klíčového rozsahu a může je získat pouze, pokud transakce využívá izolační úroveň SERIALIZABLE. Z toho pouze čtyři módy jsou nejčastější módy zámku klíčového rozsahu. Zbylých pět módů zámku klíčového rozsahu získá SQL Server pouze, když přechází z jiného zámkového módu. Těchto pět přechodových stavů je velice těžké vůbec zachytit s nástroji, které jsou k dispozici. [7]

2.3.1 RozsahS-S (sdílený zámek rozsahu a sdílený zámek zdroje)

Když transakce běží v izolační úrovni SERIALIZABLE, SQL Server drží individuální sdílené klíče nad zvolenými daty, a pokud je použit index k přístupu k datům, SQL Server bude držet sdílené zámky rozsahu nad intervaly mezi indexovými klíči.

K pochopení, které rozsahy budou muset být uzamknuty, aby bylo zabráněno vložení fantomových řádků, je potřeba přemýšlet jak se bude SQL Server snažit uložit jakékoliv nově vložené hodnoty, a přitom nezapomenout na zámek rozsahu, který zabraňuje SQL Serveru vložit nové řádky do takto uzamknutého rozsahu.

Například v tabulce Objednávek v sloupci ObjednávkaID je index na tomto sloupci, takže řádky budou uloženy v pořadí ObjednávkaID. Zámky rozsahu musí zahrnout rozsah od klíče právě před prvním zvoleným klíčem rozsahu a právě první klíč, aby první klíč samotný nemohl být modifikován. Zámky rozsahu musí také zahrnout rozsah začínající právě po druhém klíči a klíč následující právě po tomto klíči, aby nebylo možné přidat hodnotu stejnou s největším klíčem.

Například vytvoření zámku nad objednávkami s ObjednávkaID sto dvacet tři až sto dvacet šest by znamenalo, že bude uzamčen rozsah sto dvacet dva až sto dvacet sedm, aby nebylo možné přidat zámek před první objednávku ani za poslední.

Pokud není použit index k získání řádků, kde tabulka je halda, tak není možné v takové situaci vytvořit zámek rozsahu, protože zámky rozsahu jsou vždy rozsah klíčů. Pokud se pracuje v izolační úrovni SERIALIZABLE a není nalezen žádný použitelný index, pro rozsah specifikovaný ve vyhledávací klauzuli (například, klauzule vyhledávání je WHERE ObjednávkaID je mezi sto dvacet tři a sto dvacet šest), SQL Server se většinou uchýlí k uzamknutí celé tabulky. [7]

2.3.2 Rozsah S-U (sdílený zámek rozsahu a aktualizační zámek zdroje)

Pokud je použit neclusterovaný index k lokalizaci a aktualizaci řádků na haldě, když se pracuje v izolační úrovni SERIALIZABLE, a aktualizovaný sloupec není indexovaným sloupcem použitým pro přístup, SQL Server získá zámek typu Rozsah S-U. To znamená, že na rozsahu mezi indexovacími klíči je použit S zámek, ale na samotním index klíči je použit U zámek. Řádky na haldě budou mít očekávaný X zámek na řádkovém ID. [7]

2.3.3 Rozsah X-X (exklusivní zámek rozsahu a exkluzivní zámek zdroje)

Pokud se aktualizují řádky v indexu a pracuje se pod izolační úrovní SERIALIZABLE, sezení dostane exkluzivní rozsahový zámek. Ke sledování zámku rozsah X-X je zapotřebí, aby aktualizované řádky byly indexovými klíči, nezávisle na tom jestli se jedná o aktualizaci indexu v clusteru či nikoliv. [7]

2.3.4 Rozsah I-N (vložení klíčového rozsahu a žádný zámek zdroje)

Tento typ zámku indikuje exkluzivní zámek, který zabraňuje vložení na rozsahu mezi klíči, ale žádný typ zámku není nad jednotlivými klíči. Typ zámku nad rozsahem klíčů je speciální typ I, který se objevuje pouze jako část zámku klíčového rozsahu, a jelikož zde není žádný existující zdroj k uzamknutí, tak druhá část je N (z anglického „NULL“ neboli prázdný).

SQL Server získá zámek rozsahu I-N, když se pokusí o vložení hodnot do rozsahu mezi klíče v izolační úrovni SERIALIZABLE. Tento typ zámku není jednoduché vidět, protože je přechodový, a je držen pouze do doby, než je nalezena správná lokace pro vložení dat, a poté je eskalován na zámek X. Ovšem pokud jedna transakce prohledává rozsah dat v SERIALIZABLE izolační úrovni a další transakce se pokusí o INSERT do stejného rozsahu, tak je druhá transakce pozastavena zámkem v stavu čekající (z anglického „WAIT“) a je jí přiřazen zámek rozsahu I-N.

Tento typ zámku lze sledovat, když spustíme transakci s operací SELECT, která vybírá z rozsahu tabulky záznamy, ale není dokončena nebo vrácena zpět. Poté otevřením nového připojení, se transakce pokusí operací INSERT vložit data do stejného rozsahu, jako prohledává první transakce. Poté v prvním připojení lze provést dotaz na Dblocks pohled, kde je vidět že druhé připojení má žádost o zámek ve stavu čekající s módem zámku rozsahu I-N. [7]

2.3.5 Konverzní zámky rozsahu

K výše popsaným čtyřem zámkům rozsahu patří ještě několik typů, které se nazývají konverzní zámky rozsahu, které stačí pouze zmínit. SQL Server získává tyto zámky, když zámek klíčového rozsahu překryje jiný zámek, jak je ukázáno v tabulce níže (Tab. 1.).

Pokud jedno sezení na začátku získá zámek v sloupci Zámek 1 a poté, když stále drží tento první zámek, získá zámek ve sloupci Zámek 2, jsou oba zámky konvertovány na odpovídající zámek v sloupci Konverzní zámek. [7]

Tab. 1. Typy konverzních zámků [7]

Zámek 1	Zámek 2	Konverzní zámek
S	Rozsah I-N	Rozsah I-S
U	Rozsah I-N	Rozsah I-U
X	Rozsah I-N	Rozsah I-X
Rozsah I-N	Rozsah S-S	Rozsah X-S
Rozsah I-N	Rozsah S-U	Rozsah X-U

2.4 Eskalace zámků

Ve výchozím nastavení SQL Server získá zámek odpovídající nejlepší zrnitosti možné, aby udržel co největší možnou souběžnost. Ve většině případů tohle znamená, že SQL Server získá řádkové (řádkové ID nebo klíčové) zámky. SQL Server může získat stovky až tisíce jednotlivých zámků nad daty v jedné tabulce, aniž by byly způsobeny nějaké problémy. V některých případech ovšem, pokud SQL Server rozhodne, že dotaz bude přistupovat k rozsahu řádků v clusterovaném indexu, může místo zámku na úrovni řádků získat zámek na úrovni stránek. Nakonec, pokud bude přistoupeno ke každému řádku na stránce tabulky, je jednodušší spravovat jeden stránkový zámek, než desítky až stovky zámků nad jednotlivými řádky. V jiném případě, primárně pokud nad tabulkou není nějaký použitelný index pro

oporu zpracování dotazu, SQL Server může uzamknout celou tabulku hned na začátku procesu.

SQL Server může eskalovat zámky na základě celkového využití zdrojů SQL Server instance, nebo podle celkového počtu zámků získaných jedním výrazem. [7]

2.4.1 Eskalace podle využití zdrojů SQL Server instance

V některých případech může získání velkého počtu jednotlivých zámků nad řádky vyústit ve velkou spotřebu paměti SQL Serveru. Ačkoliv paměť potřebná pro každý zámek je nepatrná (okolo 96 bytů na zámek), tak počet zámků se může pohybovat až v tisících. Pokud SQL Server dosáhne hranice dvaceti čtyř procentního využití jeho nárazníkového fondu (z anglického „buffer pool“), kromě AWE paměti, aby si udržel informace o všech zámcích a čekajících žádostech o zámek, SQL Server si vybere nějaké sezení držící zámky a eskaluje „jemně“ zrnité (řádkové nebo stránkové) zámky sezení na zámky tabulkové úrovně.

Alternativně je možné specifikovat také, že je možné dosáhnout i na celo-serverovou eskalaci zámků na základě celkového počtu zámků, držených všemi sezeními na instanci. Pokud se změní hodnota LOCKS konfigurace v nastavení na jinou hodnotu než nula, SQL Server začne vybírat sezení, která budou mít eskalovány zámky, jakmile získají počet zámků roven čtyřiceti procentům čísla nastaveného v LOCKS. Například pokud je nastaveno v LOCKS číslo deset tisíc, tak eskalace nastane ihned po získání čtyř tisíc zámků získaných nebo požadavků na zámky.

Kdykoliv je spuštěna eskalace napříč celou instancí, protože byla překročena hranice využité paměti nebo je drženo přílišné množství zámků, není možné jednoznačně určit sezení, které bude vybráno k eskalaci zámků na tabulkovou úroveň, protože výběr sezení je spíše náhodný.

Navíc dokud setrvává využití paměti nad celo-instanční hranicí, tak pro každých tisíc dvě stě padesát nových zámků SQL Server provede opět eskalaci zámků nižší úrovně než tabulkových. [7]

2.4.2 Eskalace podle počtu zámků držených jedním výrazem

K eskalaci zámků při překročení celo-instančního limitu, SQL Server může také eskalovat zámky, pokud nějaké individuální sezení získá více než pět tisíc zámků v jednom výrazu. V tomto případě je ovšem jasné, které sezení bude mít své zámky eskalovány, je ním to sezení, které tyto zámky získalo. [7]

2.5 Další typy zámků

Typy zámků popsány výše jsou zámky, se kterými se lze setkat nejčastěji při prošetřování blokování a zamykání v SQL serveru. Existují ale ještě další dva typy, které se mohou vyskytnout a je dobré je zmínit. [7]

2.5.1 Petlice („Latch“)

Petlice jsou podobné zámkům, ale jsou aplikovány na fyzické úrovni a nejsou tak „nákladné“ na udržení a správu jako zámky. Petlice používají méně systémových zdrojů a jejich trvání je většinou velice krátké. Petlice a zámky vypadají velice podobně, protože oboje se může zobrazovat jako *last_wait_type* sloupec v *sys.dm_exec_requests* pohledu. SQL Trace a Windows Správce Úloh mají desítky čítačů na monitorování petlic, které velice podobně jako čítače na monitorování zámků. Jako zámky, petlice mohou být sdílené nebo exkluzivní, a mohou být uděleny nebo ve stavu čekající.

Ovšem petlice se nezobrazují v *sys.dm_tran_locks* pohledu. Petlice jsou použity k ochraně vnitřních struktur na krátké periody, mezitím co jsou čteny nebo modifikovány, nikoliv aby zajistily správné chování transakcí. Jak datová stránka samotná, tak zásobník ve kterém se data nacházejí, jsou chráněny petlicemi.

Další možnou cestou jak chápat rozdíl mezi zámkem a petlicí je, že zámek je něco, co potřebujeme k zajištění integrity dat, jako je například nutnost zajistit, aby jiná transakce nezměnila data ve chvíli, kdy jedna transakce s daty právě pracuje, protože v SQL Serveru není nic, co by zabránilo takové změně. SQL Server „nezajímá“, jestli další transakce změní data, které právě probíhající transakce zpracovává, takže je nutné použít adekvátní izolační úroveň a mechanizmus kontroly transakcí, aby bylo zajištěno adekvátní zamykání dat. Na druhou stranu, petlice jsou něco, co SQL Server potřebuje, aby ochránil fyzickou strukturu dat. Pokud by se sezení pokusilo aktualizovat stránku tabulky, když ji SQL Server čte nebo zapisuje

na disk, stránka by mohla být poškozena. Petlice předcházejí takovým druhům porušení dat. Petlice chrání fyzickou integritu dat, zámky chrání logickou integritu dat.

Velice těžce je možné ovlivnit, jak a kdy SQL Server získá nebo jak dlouho drží petlice. Petlice jsou navíc tak rychle přecházejícím jevem, že jsou těžce postřehnutelné. Většinou je petlice získána, když SQL Server načítá stránku tabulky do cache paměti, ale jakmile je stránka načtena, zámky jsou přiřazeny a petlice je uvolněna. Petlice nejsou založeny na transakcích, pro tento aspekt jsou tak přechodné.

Petlice jsou velmi vzácně důvod k obavám, ale může se objevit chyba číslo osm set čtyřicet čtyři a chyba osm set čtyřicet pět, které indikují, že vypršel čas při čekání na petlici na zásobníku. Tyto chyby jsou víceméně vždy způsobeny problémem na úrovni hardwaru, zahrnující nedostatečný vstupně/výstupní systém, který není schopen zvládnout vyvinutou zátěž, špatnou konfiguraci SQL Server systému, nebo špatný indexovací návrh, vedoucí k zbytečné zátěži SQL Serveru, protože musí zbytečně provádět více čtení než je potřeba. [7]

2.5.2 Kompilační zámky

V cache paměti SQL Serveru je většinou pouze jedna kopie plánu kompilované uložené procedury v jeden okamžik. Pro zajištění, že se v paměti nenachází více jak jedna kopie, některé části kompilačního procesu SQL Serveru musí být serializovány, aby pouze jedno sezení v daný okamžik mohlo kompilovat vybranou rutinu.

Kompilační zámky jsou získávány v průběhu částí kompilace, které musí být serializovány. Většinou SQL Server drží kompilační zámky po velmi krátkou dobu, ale v některých případech, když se hodně sezení pokouší ke spuštění jedné procedury najednou a procedura není kvalifikované pro schéma, když ej zavolána, může nastat problém se souběžností kvůli kompilačním zámkům.

Ačkoliv SQL Server potřebuje vzít kompilační zámky kdykoliv, kdy potřebuje rekompilovat nějakou proceduru, pokud potřebuje překompilovat proceduru pokaždé, když ji pouští, je situace daleko horší a tato situace nastává, kdykoliv uživatel spouští proceduru bez kvalifikování procedury se jménem schématu.

Například uložená procedura nazvaná MyProc, která je uložena jako dbo schéma, ale spuštěna uživatelem Uživatel, který má výchozí schéma uživatel_schema. Pokud Uživatel zavolá proceduru pomocí příkazu EXEC MyProc, SQL Server se při prvním spuštění podívá do

paměti cache, kde objekt nenalezne. Dokonce pokud existuje plán pro MyProc v cache paměti, SQL Server neví, že toto je ten pravý plán k použití, dokud nemůže ověřit, jestli v paměti není jiná rutina pojmenovaná MyProc v uživatel_schema. SQL Server získá exkluzivní kompilační zámek nad touto procedurou a připraví se ke kompilaci procedury, který by zahrnoval řešení jména objektu na objekt ID. Jakmile SQL Server má tento obejct ID, tak může konečně zjistit, jestli je validní plán pro dotazovanou proceduru. Pokud existuje použitelný plán v cache paměti, SQL Server může použít tento plán a nemusí kompilovat požadovanou proceduru. Ačkoliv kvůli chybějící kvalifikaci vůči schématu, SQL Server musel provést druhé prohledání cache paměti a získat exkluzivní kompilační zámek, než mohl s jistotou usoudit, že může použít existující spouštěcí plán v cache paměti.

Získání zámku a provedení nezbytných prohledání systémových tabulek může přinést zpoždění, které je dostatečné aby vedlo na blokování kvůli kompilačním zámkům. Ačkoliv délka blokování není nějak dlouhá pro jednotlivá sezení, pokud je připojeno hodně sezení, která vyvolávají tu samou proceduru, bez kvalifikace proti uživateli, tak jakmile skončí jedna kompilace, další sezení přebere roli hlavního blokujícího na několik sekund či méně a tato situace se opakuje dále a vzniká takzvané „valící“ blokování (z anglického spojení „rolling blocking“).

Tento problém má jednoduché řešení, pokud uživatel vždy kvalifikuje své uložené názvy procedur proti schématu, je tento problém razantně redukován. [7]

3 TRANSAKCE

Transakce je jedna jednotka práce, úkol nebo množina úkolů, která tvoří „vše nebo nic“ operaci. Pokud nějaká událost naruší transakci a způsobí, že transakce není z nějakého důvodu dokončena, systém by měl nedokončenou transakci brát, jako by nikdy ani nebyla spuštěna.

Transakce může obsahovat pouze pár úkonů, jako například změna ceny jedné knihy v knihovně, ale také může být rozsáhlá a komplexní, jako například aktualizace prodaného zboží ve všech prodejnách na začátku účetního období. [7]

3.1 ACID vlastnosti

Transakce má čtyři základní vlastnosti, takzvané ACID vlastnosti, které garantují validitu dat po dokončení jakékoliv transakce. [7]

3.1.1 Atomičnost (Atomicity)

Transakce je atomická jednotka práce. Buď jsou provedeny všechny změny v transakci, nebo nejsou provedeny žádné změny. Pokud systém selže před dokončením transakce (před zaznamenáním spáchání instrukce v transakčním logu), po restartu SQL server vrátí zpět změny, které byly provedeny.

Pokud se vyskytne při vykonávání transakce chyba, může dojít k vrácení provedených změn. Chyba musí být vyhodnocena jako závažná, aby bylo provedeno automatické vrácení změn. Taková chyba může být například snaha o vložení dat do souborové skupiny, která je již plná. Chyby jako porušení vlastností primárního klíče či vypršení času u zámků nejsou brány jako natolik závažné, aby se provedlo automatické vrácení změn transakce.

Samozřejmě toto chování lze nastavit. Pomocí příznaku XACT_ABORT můžeme nastavit přerušení a vrácení změn při jakékoliv vzniklé chybě. [4]

3.1.2 Konzistence (Consistency)

Výraz konzistence odpovídá stavu dat, ke kterým nám dá manažerský systém relační databáze (RDBMS) povolení přistupovat, když je transakce modifikují a dotazují se na ně.

Výraz konzistence je velice subjektivní výraz, který se odvíjí od samotné aplikace.

Konzistence také odpovídá faktu, že databáze musí dodržovat všechna integritní pravidla, která byla definována uvnitř ní pomocí omezení (primární klíče, unikátní omezení a cizí klíče). To v důsledku znamená, že přechody transakcí převádí databázi z jednoho konzistentního stavu do druhého.

Konzistence se kontroluje pomocí izolačních vrstev. [4]

3.1.3 Izolace (Isolation)

Izolace zajišťuje, že transakce má dostupná pouze data, která jsou konzistentní. Která data jsou konzistentní pro transakci je definováno nastavením izolačních vrstev.

S diskovými tabulkami SQL Server podporuje dva různé modely pro ovládání izolací.

1. Uzamykání
2. Kombinace uzamykání a verzování řádků

Oběma modelům a jejich izolačním vrstvám bude věnována následující kapitola (*4 Izolační Vrstvy*) o jednotlivých izolačních vrstvách a jejich vlastnostech.

3.1.4 Výdrž (Durability)

Změny v datech jsou vždy zapsány do transakčního logu databáze na disku, než jsou zapsány do datové části databáze na disku. Až po instrukci spáchání transakce, která je zapsána do transakčního logu na disku, je transakce považována za trvanlivou a to i v případě, že změny ještě nebyli provedeny v datové části na disku.

Ve chvíli kdy systém nastartuje, ať už normálně či po selhání, SQL Server prověří log transakcí každé databáze a spustí zotavovací proces, který má dvě fáze.

První fáze se nazývá předělat (z anglického „redo“). Tato fáze zahrnuje přehrání všech změn z jakékoliv transakce, která má zaznamenanou provedenou instrukci spáchání v logu transakcí, ale změny se neprovedly v datové části databáze na disku.

Druhá fáze se nazývá vrátit (z anglického „undo“). V této fázi jsou odstraněny všechny změny provedené jakoukoliv transakcí, která nemá v logu transakcí zaznamenánu provedenou instrukci spáchání. [4]

4 IZOLAČNÍ ÚROVNĚ

Izolační úrovně určují úroveň konzistence dat, když se s nimi pracuje.

Ve výchozím nastavení SQL Serveru, čtenáři využívají sdílené zámky na cílový zdroj a zapisovatelé používají exkluzivní zámky.

Při takovém nastavení není možné kontrolovat, jak se chovají zapisovatelé v rámci přidělování zámků ani jakou mají přidělené zámky časovou trvanlivost.

Je ale možnost kontrolovat chování čtenářů. Důsledkem kontrolování chování čtenářů, je možné nepřímo ovlivnit chování zapisovatelů. Tohoto způsobu ovlivnění chování zapisovatelů lze dosáhnout, když je nastavena izolační úroveň buď na úrovni sezení v nastavení sezení, nebo na úrovni dotazovací pomocí tabulkové rady.

SQL Server nabízí čtyři izolační úrovně založené na zámkovém modelu:

1. READ UNCOMMITTED
2. READ COMMITTED
3. REPEATABLE READ
4. SERIAZABLE

SQL Server nabízí dvě izolační úrovně založené na modelu kombinace zámků a verzování řádků:

1. SNAPSHOT
2. READ COMMITTED SNAPSHOT

Změnou izolační úrovně v databázi, se změní jak paralelní chování databázových uživatelů, tak i dosažené konzistence dat.

Izolační úrovně založené na zámkovém modelu nabízejí striktnější pravidla pro zámky a jejich delší časovou trvanlivost, čím vyšší izolační úroveň je. READ UNCOMMITTED je tedy nejméně striktní, proto se také používá pro ni alias NOLOCK. SERIAZABLE je naopak nejstriktnější a používá nejdelší časovou trvanlivost zámků, také pro ni je alias HOLDLOCK. Zvolením vyšší izolační úrovně je tedy dosaženo vyšší konzistence dat, ale jsou sníženy možnosti paralelní práce s nimi.

U izolačních úrovní založených na modelu verzování řádků si SQL Server uchovává poslední validní verzi řádků v tempdb. Při příchozí operaci pro čtení dat se zjistí, zda je validní

požadovaný řádek. Pokud tomu tak není, tak je poskytnuta poslední validní verze řádku z tempdb. [4]

4.1 Zabranitelné fenomény čtení

Nejjednodušším způsobem jak popsat rozdíl mezi různými ANSI izolačními vrstvami je popsat množinu chování, která jsou povolena nebo naopak zakázána. [7]

Tab. 2. Povolené chování fenoménů čtení v izolačních úrovních [7]

Izolační úroveň transakce	Povolené chování				Model souběžnosti
	Špinavé čtení	Neopakovatelné čtení	Ztráta updatu	Čtení fantomů	
READ UNCOMMITTED	Ano	Ano	Ano	Ano	Pesimistický
READ COMMITTED	Ne	Ano	Ano	Ano	Pesimistický
READ COMMITTED SNAPSHOT	Ne	Ano	Ano	Ano	Optimistický
REPEATABLE READ	Ne	Ne	Ne	Ano	Pesimistický
SNAPSHOT	Ne	Ne	Ne	Ne	Optimistický
SERIALIZABLE	Ne	Ne	Ne	Ne	Pesimistický

4.1.1 Špinavé čtení (Dirty read)

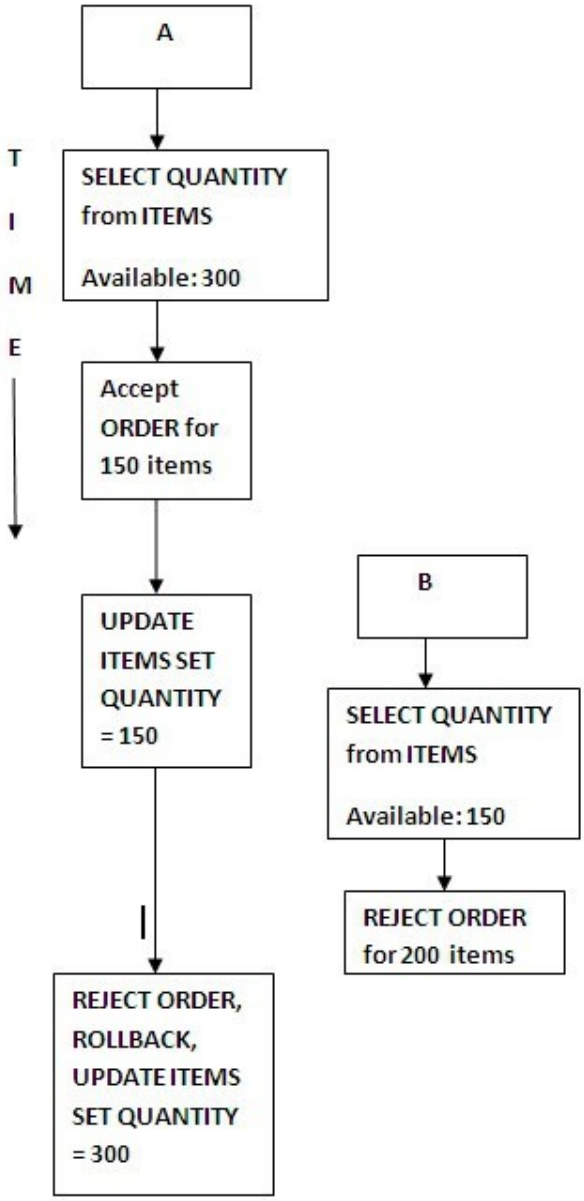
Toto chování se objeví, pokud je transakci umožněno číst data, nad kterými nebyly dokončeny všechny změny. Pokud jedna transakce změnila data, ale ještě nezapsala změny, a další transakce je umožněno číst tyto změněné data, tak je velice pravděpodobné, že data budou přečteny v nekonzistentním stavu.

Například aplikace managementu skladu pro výrobu, která přijímá a vydává nástroje. Prodavači logují dodávky a odeslané objednávky, aktualizující nástrojový inventář.

V jednu chvíli je v databázi inventáře skladu pouze dvaceti pěti nástrojů, ale nová dodávka padesáti nástrojů právě přišla, takže prodavač A začne transakci a vydá UPDATE, aby zvýšil počet nástrojů v databázi inventáře na sedmdesát pět. V tu chvíli prodavač B dostane objednávku na šedesát nástrojů a ověří dostupnost v databázi inventáře. Pokud je transakci prodavače B povoleno špinavé čtení, prodavač B uvidí sedmdesát pět nástrojů a může tak vyřídit objednávku na šedesát nástrojů, aby mohla být odeslána zákazníkovi. Mezitím se prodavač A připravuje k potvrzení aktualizační transakce databáze na sedmdesát pět nástrojů, ale dostane zprávu, že byla nalezena vada v dodávce nástrojů, a dodávka nástrojů by se měla vrátit

výrobci. Jako výsledek, prodavač A zruší transakci (vrácení změn transakce) a tím prodavač A autorizuje zakázku, kterou není společnost schopna pokrýt, protože na skladě už nemá dostatek nástrojů.

SQL Server ve výchozím nastavení nepovoluje špinavé čtení. Je potřeba držet v paměti, že transakce aktualizující data nemá kontrolu nad tím, jestli může jiná transakce číst stejná data, která jsou měněna, před tím než je dokončena celá transakce a zapsána v logu transakcí. Rozhodnutí, jestli bude nebo nebude číst špinavé data, je čistě na straně čtecí transakce. [7]

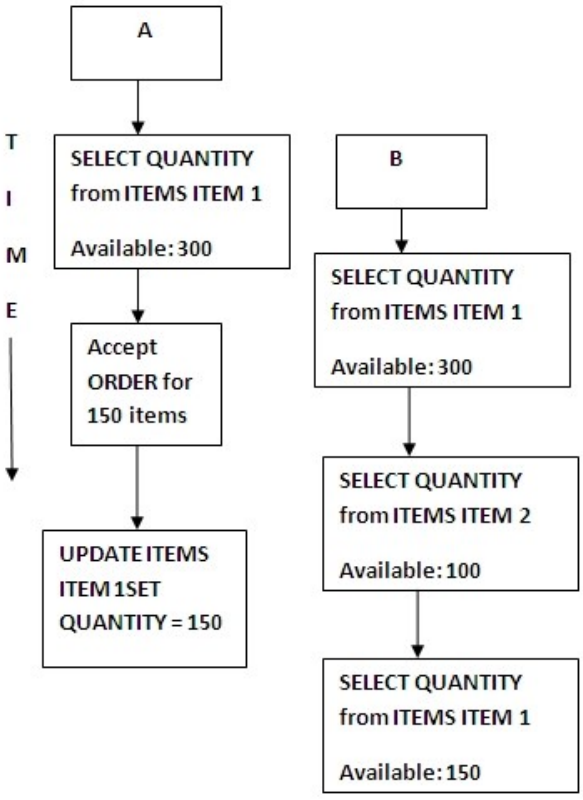


Obr. 2. Schéma špinavého čtení [8]

4.1.2 Neopakovatelné čtení (Non-repeatable read)

Toto chování je také nazývané nekonzistentní analýza. Čtení je neopakovatelné, pokud dotaz může vrátit jiné hodnoty výsledků, když stejná data čte ve dvou rozdělených čteních v rámci jedné transakce. Takové chování může nastat ve chvíli, kdy jedna transakce provede první čtení dat, poté jsou data aktualizována jinou transakcí na jiné hodnoty a následně jsou opět přečtena první transakcí.

V příkladu přijímací místnosti, manažer se rozhodne provést kontrolu aktuálního stavu inventáře. Manažer přijde ke každému skladníkovi, zeptá se na jeho celkový počet přijatých dílů za ten daný den a připočítává si je na kalkulačce. Když manažer dokončí kontrolu, rozhodne se udělat druhou kontrolu pro jistotu a jde opět k prvnímu skladníkovi. Problém nastává, pokud nějaký skladník přijal další díly mezi první a druhou návštěvou manažera, protože tím se změní koncový součet dílů všech skladníků při každém sčítání a tím se stává sčítání neopakovatelným. [7]

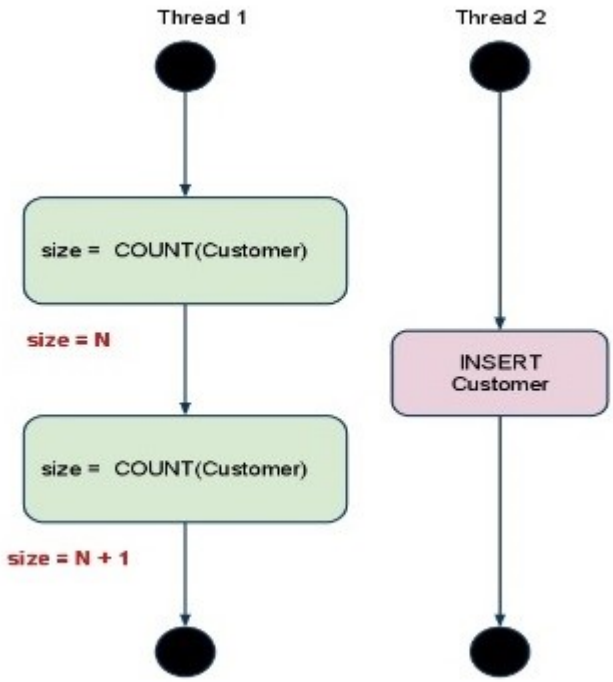


Obr. 3. Schéma neopakovatelného čtení [15]

4.1.3 Čtení fantomů (Phantom read)

Toto chování se objevuje, pokud je změněno členství v množině. Může nastat pouze pokud je použit dotaz s predikátem, jako například WHERE počet dílů je menší než deset. Fantom se objeví, pokud dvě SELECT operace používající ten samý predikát ve stejné transakci vrátí rozdílný počet řádků.

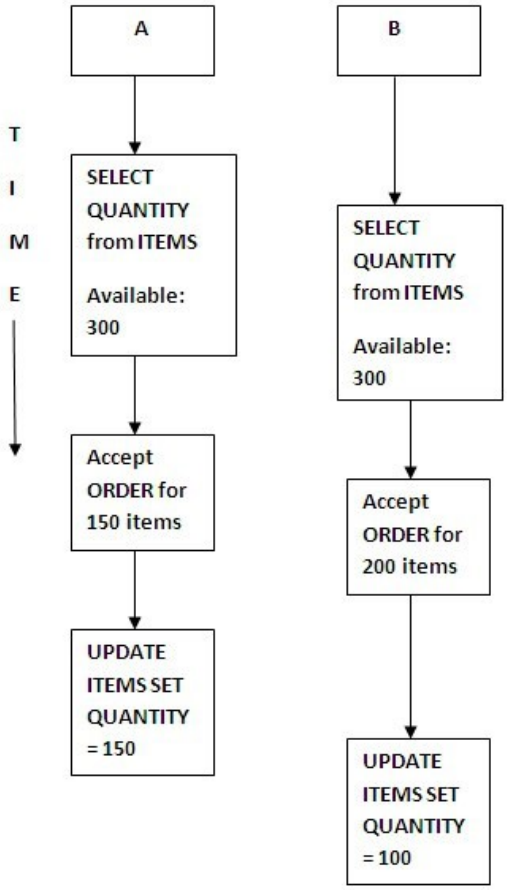
Například v situaci příjímací místnosti, kde manažer obchází skladníky a sčítá jejich celkový denní příjem dílů. Tentokrát místo sčítání dílů si zapisuje skladníky, kteří mají méně jak deset přijatých dílů. Po dokončení obchůzky všech skladníků, se manažer rozhodne jít od začátku a poskytnout nějakou radu všem skladníkům, které si poznamenal. Ale při první obchůzce manažer nezapočítal skladníka, který byl na obědě a měl méně jak deset dílů přijatých. Po první obchůzce se vrátil a manažer ho nemá v seznamu, takže skladník co přišel z oběda je fantomem. [7]



Obr. 4. Schéma čtení fantomů [16]

4.1.4 Ztráta updatu (Lost update)

Ztracený update nastane, pokud dvě transakce přečtou hodnotu, na základě této hodnoty provedou výpočet a poté tuto hodnotu updatují. Jelikož ji ale updatují obě transakce, zapsána zůstane pouze hodnota z poslední transakce, protože tato transakce změní hodnotu jako poslední a není přepsána žádnou další transakcí. [4]



Obr. 5. Schéma ztraceného updatu [8]

4.2 Izolační úroveň READ UNCOMMITTED

READ UNCOMMITTED je nejnižší nabízenou izolační vrstvou.

V této izolační vrstvě čtenáři nežádají o sdílený zámek. Čtenář, který nežádá o sdílený zámek, nikdy nemůže být v konfliktu se zapisovatelem, která drží exklusivní zámek. V důsledku se tedy může stát, že čtenáři je umožněno přečíst data, nad kterými nebyly dokončeny všechny změny. Tento problém se také nazývá špinavé čtení.

V této izolační vrstvě se tedy může stát, že zapisovatel přepíše část dat, která jsou právě čtena jiným čtenářem. [4]

4.3 Izolační úroveň READ COMMITTED

READ COMMITTED izolační vrstva odstraňuje problém READ UNCOMMITTED izolační vrstvy. READ COMMITTED je nejnižší izolační vrstva zabraňující problému špinavého čtení.

Podle názvu izolační vrstvy si lze odvodit, jaký je rozdíl mezi READ COMMITTED a READ UNCOMMITTED vrstvami. V READ COMMITTED vrstvě je čtecí operaci povoleno číst pouze data, nad kterými nejsou prováděny žádné změny. Čtenáři je přiřazen sdílený zámek a tím je zajištěno, že nad čtenými daty není prováděna operace zápisu. Sdílený zámek přidělený čtenáři při pokusu o čtení dat je v konfliktu s exkluzivním zámkem pro zapisovatele. Operace čtení musí tedy počkat, než je provedena celá transakce zapisovatele a uvolněn její exkluzivní zámek. Poté je čtenáři přiřazen sdílený zámek a on může přečíst požadovaná data. Čtenář po provedené transakci čte nutně pouze spáchané změny.

READ COMMITTED je výchozí izolační úrovní v SQL Serveru. [4]

Izolační úroveň READ COMMITTED se také vyznačuje několika problémy. Krátkodobé S zámky využívané izolační úrovní READ COMMITTED poskytují pramálo jistot. Výraz běžící pod READ COMMITTED izolační úrovní může:

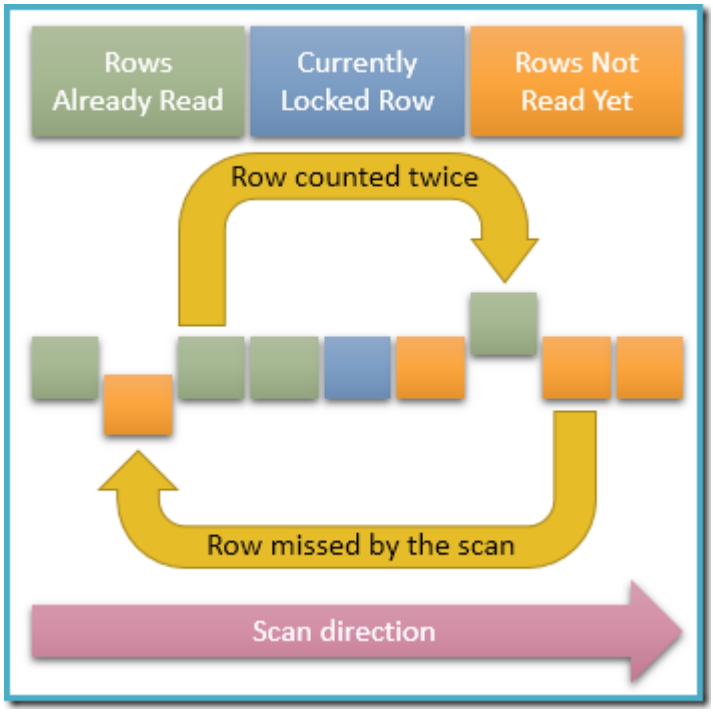
1. Přečíst stejný řádek několikrát
2. Vynechat řádky
3. Nedokáže poskytnout bod-v-čase pohled na data

Tento seznam může vypadat spíš jako popis podivného chování, které odpovídá příznaku NOLOCK, ale všechny tyto události se mohou stát při běhu transakce pod úrovní READ COMMITTED. [10]

4.3.1 Vícenásobné čtení a vynechání řádku

Například jednoduché spočítání všech řádků v tabulce, za použití jednoduchého dotazu o jednom výrazu. Tento dotaz je spuštěn pod READ COMMITTED izolační úrovní se zrnitostí na řádkové úrovni. Dotaz získá S zámek na první řádek tabulky, přečte jej, uvolní S zámek a přejde na další řádek. Tento postup opakuje, dokud nenarazí na konec struktury, kterou čte. Jelikož je ale zamknut pouze jeden řádek v daný čas, je ostatním transakcím umožněno modifikovat ostatní nezamčené řádky, které dotaz prochází. Pokud taková transakce změní hodnotu indexového klíče, způsobí pohyb řádků v indexovací struktuře.

Obrázek níže (Obr. 4.) ukazuje schéma takového případu:



Obr. 6. Schéma problému updatu při zablokovaném čtení [10]

Na schématu lze vidět, že horní šipka přečte jeden řádek vícekrát, protože po prvním přečtení byla hodnota řádku změněna jinou transakcí a řádek se posunul před aktuálně čtený řádek v pořadí indexů, takže řádek bude přečten dvakrát.

Spodní šipka ukazuje řádek, který ještě nebyl přečten, ale byl změněn jinou transakcí a posunut v seznamu indexů za aktuální pozici čtení, takže řádek nebude vůbec přečten ve výsledku. [10]

4.3.2 Pohled na data v určitém bodu v čase

Důvod za tímto tvrzením je po předchozím případu vcelku jasně viditelný. V příkladu z bodu 4.3.1, kde je popsán průběh dotazu sčítajícího řádky v tabulce, bylo možno jednoduše číst data vložené souběžnou transakcí poté, co byl spuštěn dotaz pro sčítání. Taktéž data, která viděl sčítací dotaz, mohla být modifikována souběžnou aktivitou poté, co byl sčítací dotaz spuštěn a mezitím než byl dokončen. A nakonec již přečtené a započítané řádky mohly být smazány souběžnou transakcí před dokončením dotazu.

Data které vidí výraz nebo transakce, běžící pod READ COMMITTED izolační úrovní, odpovídají nejednotnému stavu databáze v jakémkoliv bodu v čase. Takže data se kterými hodláme pracovat mohou být z různých bodů v čase, kde jediným společným faktorem je, že každá položka reprezentuje poslední vloženou hodnotu těchto dat v čase, kdy byla přečtena (s tím že od té doby mohla být změněna nebo úplně smazána). [10]

4.4 Izolační úroveň REPEATABLE READ

Izolační úroveň REPEATABLE READ zajišťuje konzistenci mezi jednotlivými čteními v průběhu jedné transakce. Neboli v průběhu jedné transakce je znemožněno jinému zapisovateli, aby změnila data, dokud transakce není kompletně dokončena a zaznamenána jako spáchaná.

V této izolační úrovni je po čtenáři vyžadován nejen sdílený zámek, aby mohl číst data, ale také si drží tento sdílený zámek po dobu trvání transakce. To znamená, že jakmile dostane čtenář sdílený zámek pro čtení nad datovým zdrojem, nikdo nemůže dostat exkluzivní zámek, aby tento datový zdroj mohl změnit, dokud čtenář nedokončí transakci. Tím je zajištěno konzistentní analýzu nebo opakované čtení.

Izolační úroveň REPEATABLE READ dále předchází ztraceným updatům, které se mohou objevit u nižších izolačních vrstev.

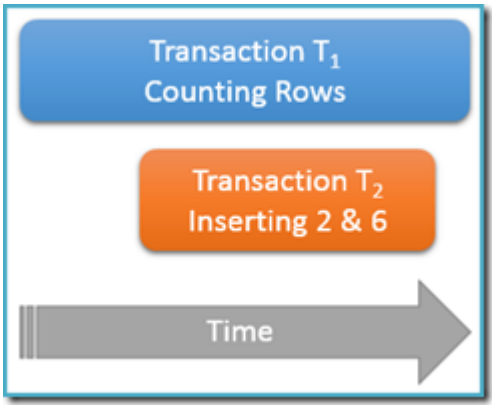
Ztracený update nastane, pokud dvě transakce přečtou hodnotu, na základě této hodnoty provedou výpočet a poté updatují hodnotu. Protože v nižší úrovni než je REPEATABLE READ není držen žádný zámek nad zdrojem dat po jeho přečtení, obě transakce mají možnost upravit hodnotu. V tu chvíli záleží na tom, která transakce změnila hodnotu jako poslední a její hodnota již nebude přepsána. V REPEATABLE READ úrovni si obě transakce drží své sdílené zámky po prvním čtení, takže ani jedna transakce nemůže získat exkluzivní zámek později, aby mohla upravit hodnotu. Tato situace vyústí v takzvaný deadlock. [4]

4.4.1 Neopakovatelné čtení a přeskočení záznamu

Například v databázi je tabulka s pěti hodnotami (jedna, tři, čtyři, pět a sedm). Nad touto tabulkou je nastavena izolační úroveň REPEATABLE READ a spuštěna transakce a v ní dotaz na zjištění počtu řádků, ale transakce není dokončena. Dotaz vrátí očekávaný počet pěti řádků.

Dále je spuštěn opět dotaz na celkový počet řádků v tabulce a zároveň je spuštěna další transakce na přidání dvou nových hodnot.

Celá situace je zobrazena na schématu:



Obr. 7. Schéma příkladu [12]

Izolační úroveň REPEATABLE READ garantuje, že druhé spuštění sčítacího dotazu uvidí všechny řádky přečtené v prvním čtení a budou ve stejném stavu. Ovšem REPEATABLE READ izolační úroveň už nic neříká o tom, jak řešit nově přidané řádky (fantomy).

Dotaz na zjištění celkového počtu řádků je na hodnotě tři, když jsou vloženy dvě nové hodnoty (dvě a šest). Takže pokračující dotaz uvidí již nově přibylou hodnotu šest, ale už neuvidí nově přibylou hodnotu dvě, protože prohledávání je již za její v době přidání do tabulky.

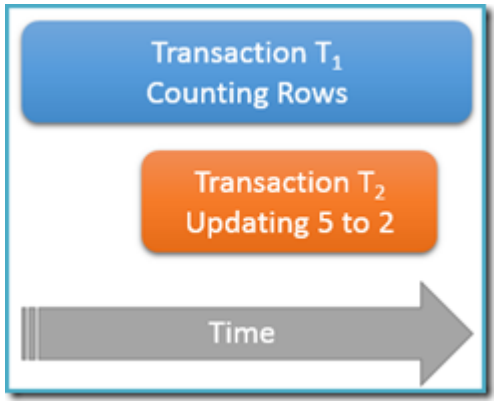
Výsledek dotazu pod izolační úrovní REPEATABLE READ napočítá celkově šest řádků v tabulce (hodnoty jedna, tři, čtyři, pět, šest a sedm). Tento výsledek je nekonzistentní s předešlým výsledkem dotazu, který spočítal pět řádků, v též transakci. Druhé čtení navíc započítalo pouze jednu ze dvou nových hodnot.

Stejný problém nastává i pro operaci UPDATE. V databázi je tabulka o pěti hodnotách stejných jako v příkladu výš. Je spuštěna transakce na zjištění celkového počtu řádků v tabulce a poté je spuštěna také transakce na aktualizaci hodnoty pět na hodnotu dvě.

V tomto případě opět sčítací transakce dojde na hodnotu tři, když je spuštěna transakce na aktualizaci hodnoty pět na hodnotu dvě a tabulka je poté seřazena vzestupně.

Transakce na zjištění počtu řádků pokračuje dále od hodnoty tři, až dojde po hodnotu sedm. Výsledný počet řádků je tedy čtyři řádky (jedna, tři, čtyři a sedm), protože hodnota dvě v době přechodu z hodnoty jedna na hodnotu tři v tabulce ještě nebyla a po aktualizaci z tabulky zmizela také hodnota pět. [12]

Tento případ je vidět na schématu:



Obr. 8. Schéma druhého příkladu [12]

4.5 Izolační úroveň SERIALIZABLE

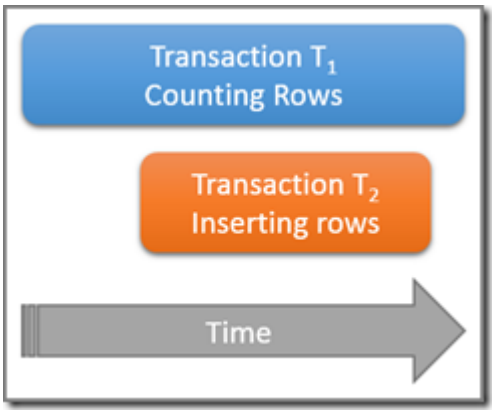
V izolační úrovni REPEATABLE READ si čtenář udržuje sdílený zámek až do konce transakce. Takže je garantováno konzistentní opakované čtení dat, která byla přečtena v prvním spuštění transakce, nikoliv data která neexistovala v době spuštění transakce. Druhé spuštění transakce může tedy vrátit i nově přidaná data, takto přečteným datům se říká fantomové čtení. Fantomové čtení nastane, pokud mezi čteními dat vloží jiná transakce nová data.

Izolační vrstva SERIAZABLE předchází fantomovému čtení na rozdíl od předešlé izolační vrstvy REPEATABLE READ. Na rozdíl od izolační vrstvy REPEATABLE READ, SERIAZABLE uzamyká čtenáři celý rozsah klíčů, které odpovídají čtenářově dotazovacímu filtru. Tím jsou čtenáři uzamknuty již existující řádky odpovídající jeho dotazovacímu filtru, ale také ty řádky, které tomuto filtru budou odpovídat i v budoucnu. Principiálně to funguje tak, že jsou zablokovány pokusy jiných transakcí o přidání řádků, které odpovídají čtenářově dotazovacímu filtru. [4]

4.5.1 Pohled na data v určitém bodu v čase

Důležité je si uvědomit, že je rozdíl mezi logickou funkčností SERIALIZABLE izolační úrovně a fyzickou funkčností, aby se při návrhu databáze či transakcí nepředpokládaly jistoty, které nemusí vůbec v dané implementaci existovat. Například pokud je předpokládáno, že serializovatelné transakce jsou skutečně spouštěny jedna po druhé, je možné se poté domnívat, že SERIALIZABLE transakce nutně uvidí databázi, jak existovala v době spuštění transakce a tím dávající pohled na data v určitém bodě v čase.

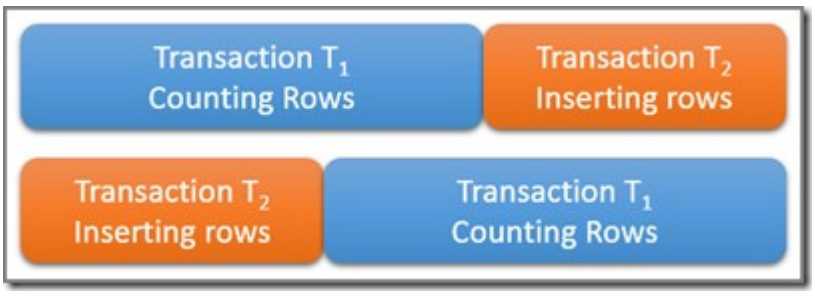
Ve skutečnosti je tento detail specifický pro implementaci. Například v databázi je tabulka s několika řádky, ze kterých pět řádků odpovídá dotazu s predikátem. Serializovatelná transakce T1 začne sčítat řádky v tabulce, které odpovídají danému dotazu a jeho predikátu. Nějaký čas po spuštění transakce T1, ale před jejím dokončením, druhá serializovatelná transakce T2 je také spuštěna. Transakce T2 přidává čtyři nové řádky, které také odpovídají dotazu a jeho predikátu, do tabulky a zapíše je. Na obrázku (*Obr.5.*) je časové schéma takového případu:



Obr. 9. Schéma příkladu [11]

Nastává tedy otázka: „Kolik řádků ve výsledku spočítá transakce T1?“.

Obě transakce se překrývají v čase. SERIALIZABLE izolační úroveň vyžaduje pouze, aby výsledek těchto dvou transakcí odpovídal nějakému sériovému spuštění. Pro logické sériové naplánování transakcí T1 a T2 jsou dvě možnosti, které jsou zobrazeny na obrázku (Obr.6.) níže:



Obr. 10. Možné řešení příkladu [11]

Při použití prvního možného sériového naplánování (T1 a poté T2), T1 napočítá pět řádků, protože T2 transakce nové čtyři řádky začne vkládat až po dokončení T1. Při použití druhého možného řešení, T1 by napočítala devět řádků, protože vložení čtyř nových řádků proběhlo již před spuštěním sčítací transakce T1.

Obě odpovědi jsou logicky správně pod SERIALIZABLE izolační úrovní. Navíc není přípustná žádná jiná odpověď (například napočítání sedmi řádků transakcí T1). Která z těchto dvou možností nastane, je ale závislé na načasování a specifické implementaci databázového systému.

Pokud je vrácený počet řádků devět, první transakce mohla vidět řádky které neexistovaly v době spuštění transakce T1. Tento výsledek je možné získat v SQL Server, ale nikoliv v PostgreSQL Serializable Snapshot Isolation, ačkoliv obě implementace odpovídají logickému chování specifikovanému pro SERIALIZABLE izolační úroveň.

V SQL Serveru serializované transakce nemusí vidět data, jak existovala na začátku transakce. Spíše detaily implementace SQL Serveru umožňují serializované transakci vidět poslední validní data v moment, kdy data byla poprvé zamknuta pro přístup. Navíc množina posledních validních dat, čtena nakonec, má garantované neměnné členství, dokud není transakce dokončena. [11]

4.6 Izolační úroveň SNAPSHOT

V izolační úrovni SNAPSHOT, pokusí-li se čtenář číst data, je mu garantováno, že dostane poslední validní verzi řádku, který byl dostupný, když transakce začala. To znamená, že je garantováno čtení konzistentních dat a opakované čtení těchto dat, přitom je také předcházeno fantomovému čtení. Tedy stejné výhody jaké nabízí izolační úroveň SERIALIZABLE úroveň. Ale SNAPSHOT úroveň na rozdíl od SERIALIZABLE úrovně nevyužívá sdílené zámky, ale využívá verzování řádků.

Verzování řádků přináší výhody i nevýhody. Výhodami je odstranění zámků a tedy zrychlení INSERT operací, protože neexistující záznam není potřeba verzovat. Také může dojít až k dramatickému zrychlení čtecích operací, jelikož po čtenáři není vyžadován sdílený zámek a není potřeba čekat, pokud je nad daty exkluzivní zámek nebo jsou data v neočekávané verzi.

Nevýhodou verzování řádků je zpomalení operací UPDATE a DELETE. Zpomalení je způsobeno nutností překopírovat stávající řádek před provedením změny nad řádkem v tempdb.

Tato izolační úroveň je výchozím nastavením pro Azure SQL. [4]

4.6.1 Zkřivený zápis (Write skew)

Izolační úroveň SNAPSHOT je náchylná na fenomén známý jako křivý zápis. Čtení stálých dat hraje svou roli v tomto problému.

Křivý zápis se objevuje, pokud dvě souběžné transakce čtou data, když je druhá transakce modifikuje. Žádný konflikt zápis se neobjeví, protože obě transakce upravují jiné řádky. Ani jedna transakce nevidí změny provedenou druhou transakcí, protože obě transakce čtou data z bodu v čase, který je před tím než byly změny provedeny.

Například vytvořené dvě tabulky s jedním sloupcem typu integer. Nejdříve se pokusí první transakce projít celou druhou tabulku a celkový počet řádků vloží do první tabulky jako jeden nový záznam, ale není dokončena. Spustí se tedy druhá transakce, která projde celou první tabulku a celkový počet řádků vloží do druhé tabulky jako nový záznam a je dokončena. Následně je dokončena i první transakce.

Výsledkem tohoto příkladu je jeden záznam s hodnotou nula v obou tabulkách. Tento výsledek je správný, ale není to výsledek žádného možného serializovaného řešení pro spuštění transakcí. V jakémkoliv skutečně serializovaném plánu, jedna transakce by musela být dokončena před začátkem další transakce. To by vedlo na výsledek, že jedna z tabulek by měla záznam o hodnotě jedna, protože jedna transakce byla dokončena a druhá transakce již přečetla záznam zapsaný první transakcí. Může to znít jako detail, ale je potřeba mít na paměti silné jistoty serializace, které ale fungují pouze při opravdu serializovatelných transakcích. [14]

4.7 Izolační úroveň READ COMMITTED SNAPSHOT

Izolační úroveň READ COMMITTED SNAPSHOT je úroveň založená na modelu verzování řádků. Rozdíl mezi touto izolační úrovní a izolační úrovní SNAPSHOT je, že izolační úroveň SNAPSHOT vrací pohled na data konzistentní na úrovni transakcí, kde READ COMMITTED SNAPSHOT vrací pohled na data konzistentní na úrovni prohlášení (z anglického „statements“).

Izolační úroveň READ COMMITTED SNAPSHOT nedetekuje konflikty updatů. Je to z důvodu logické podobnosti izolační úrovni READ COMMITTED. Rozdíl mezi izolační úrovní READ COMMITTED a READ COMMITTED SNAPSHOT je, že v READ COMMITTED SNAPSHOT čtenáři nedostanou sdílený zámek a nečekají tedy, když žádají o exkluzivně uzamčené datové zdroje.

Pro vynucení funkcionality, aby čtenář musel získat sdílený zámek, je nutné použít tabulkovou radu. [4]

4.7.1 Pohled na data v bodu v čase

Pohled na data v určitém bodu v čase odstraňuje problém s chybějícími řádky nebo přečtením jednoho řádku vícekrát. Další velkou výhodou READ COMMITTED SNAPSHOT izolační úrovně je čtení bez nutnosti získání S zámku, jelikož data jsou čteny z verzovaného uložiště tempdb. Odstranění nutnosti získávání S zámků může dramaticky zrychlit běh celé aplikace, jelikož jsou odstraněny konflikty se souběžnými transakcemi, které chtějí získat nekompatibilní zámky. Tato vlastnost se většinou označuje jako „čtenáři neblokují zapisovatele“ a naopak. Dalším dopadem nepoužití zámků je až dramatické snížení blokování a možnosti vzniku deadlocku pod izolační úrovní READ COMMITTED SNAPSHOT.

Ovšem tyto výhody jsou vyváženy zápory a úskalími. Jednou z těchto nevýhod je, že samotné udržování verzí aktualizovaných řádků si žádá větší vytížení systémových zdrojů a je nutné, aby fyzické možnosti systému byly v tomto směru adekvátně navrženy, hlavně výkon a paměťové zdroje pro tempdb.

Další úskalí je v samotném verzování dat. Transakce pracuje s daty, která odpovídají stavu či spíše verzi, aktuálních v čase spuštění transakce samotné. Žádný mechanizmus již nebrání modifikaci dat v průběhu transakce, protože databáze nevyžaduje k běhu transakce žádné S zámky pro čtení. Takže vzniklým problémem je možnost, že transakce pracuje nebo dokonce se rozhoduje na základě zastaralých dat, místo aktuálního stavu uložených dat v databázi.[13]

4.7.2 Neopakovatelné čtení a fantomové

Izolační úroveň READ COMMITTED SNAPSHOT používá pohled na data v čase na úrovni výrazů.

Čtení stejných dat několikrát v rámci jednoho výrazu pod izolační úrovní READ COMMITTED SNAPSHOT vždy vrátí stejný výsledek, žádné data se neztratí mezi těmito čteními ani žádná nová data nepřibydou. Jde tedy o výraz který přečte data více jak jedenkrát (například dotaz odkazující se na jednu tabulku dvakrát pomocí poddotazu).

Konzistence na úrovni výrazů je jasný důsledek skutečnosti, že čtení jsou uskutečněna proti fixní verzi dat. Důvod proč izolační úroveň READ COMMITTED SNAPSHOT neposkytuje

obranu proti neopakovatelným čtením a fantomům je takový, že tyto SQL standardní fenomény jsou definovány na úrovni transakční, nikoliv výrazové. Několik výrazů uvnitř transakce běžící pod izolační úrovní READ COMMITTED SNAPSHOT mohou vidět jiná data, protože každý výraz vidí data podle jeho individuálního času spuštění.

Jinak řečeno, každý výraz uvnitř READ COMMITTED SNAPSHOT transakce vidí statickou konzistentní množinu dat, ale tato množina se může lišit mezi výrazy uvnitř jedné transakce. [13]

4.7.3 Zastaralá data

Pravděpodobnost že T-SQL procedura bude provádět důležité rozhodnutí založené na zastaralých datech je poněkud zneklidňující. Například výraz může pracovat s verzí dat, která jsou svévolně stará.

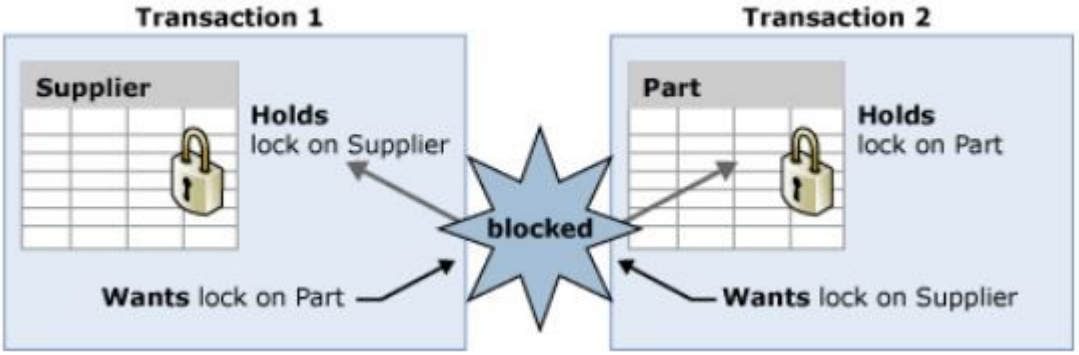
Výraz běžící po značně dlouhou dobu bude stále vidět verzi databáze, ve které se nacházela databáze, když byl výraz spuštěn. Mezitím výraz přichází o všechny provedené změny v databázi od jeho spuštění.

Tohle není problém spojený pouze s dlouho běžícími výrazy a jejich přístupu k zafixovaným datům, ale problém může v takovém případě být ještě horší. [13]

5 DEADLOCK

Deadlock neboli česky „uváznutí“ je situace, kdy dvě nebo více operací se blokují a ani jedna není schopna pokračovat. Například když operace A blokuje operaci B a operace B blokuje operaci A.

Deadlocky jsou náročná záležitost. Hlavním důvodem je nutnost vrátit zpět změny, které již byly provedeny a poté, za pomoci logiky zpracování chyb, opět provést tu stejnou práci. [4]



Obr. 11. Schéma deadlocku dvou transakcí [7]

5.1 Typy deadlocků

V databázi se lze setkat s několika typy deadlocků, které vznikají od konfliktu mezi sémanticky protichůdnými transakcemi, deadlockem mezi čtecí transakcí a zapisovací až po deadlock vyvolaný samotnou striktností zámků u izolační úrovně SERIALIZABLE.

5.1.1 Čtenář-čtenář deadlock

Dalším typickým deadlockem, který se nazývá „smrtící objetí“ (z anglického „deadly embrace“), je případ, kdy transakce přistoupí k datovým zdrojům v opačném pořadí. Například transakce 1 přistoupí k tabulce Produktů a poté k tabulce Prodejny a transakce 2 přistoupí nejdříve k tabulce Prodejny a poté k tabulce Produkty. Tomuto typu deadlocku lze předejít tak, že obě transakce budou přistupovat k tabulkám ve stejném pořadí. Takže přehozením pořadí přístupu v jedné z transakcí výše, by neměl takový typ deadlocku nikdy nastat.

Deadlock nemusí ale nastat z důvodu konfliktní logiky transakcí. Může nastat také z důvodu špatného indexování, které podporuje dotazovací filtry. Například když sezení jako v předchozím případě přistupují k tabulkám a v nich hledají určitý záznam, například sezení jedna hledá produkt číslo 1 a sezení dvě produkt s číslem 2. Z logického hlediska aby ani při další

práci s produkty neměl nastat žádný problém. Ale pokud v tabulce produktů nejsou definovány indexy nad sloupcem nesoucím id produktu, SQL Server musí projít všechny řádky v tabulce a zamknout je, což opět vede na deadlock. [4]

5.1.2 Čtenář-zapisovatel deadlock

Jedná se o typ deadlocku mezi operací pro čtení a operací pro zápis či modifikaci dat.

Při pohledu na použité zdroje, lze vidět charakteristickou známku tohoto typu deadlocku. Známkou tohoto typu jsou zámky, které jsou buď udělené sdílené zámky a vyžádané exkluzivní zámky nebo exkluzivní zámky udělené a sdílené zámky vyžádané.

Ve své výchozí izolační úrovni SQL Serveru (READ COMMITTED), SQL Server drží sdílený zámek, dokud není provedena celá operace. Proto čtenář-zapisovatel deadlocky jsou nejčastěji převládající v transakcích, které mají čtení dat následováno modifikací dat nebo v transakci, která si vyžaduje vyšší izolační úroveň, což vede k prodloužení sdíleného zámku až na délku transakce. [5]

5.1.3 Zapisovatel-zapisovatel deadlock

V deadlocku typu zapisovatel-zapisovatel jsou oba sdílené i exkluzivní zámky nad zdrojem dat buď aktualizace, nebo exkluzivní zámek. Jinak řečeno, obě operace se pokouší o modifikaci dat.

Při řešení deadlocku typu zapisovatel-zapisovatel je nutno si uvědomit, že exkluzivní zámky jsou drženy SQL Serverem až do spáchání transakce, na rozdíl od sdílených zámků v izolační úrovni READ COMMITTED, kde jsou sdílené zámky drženy pouze do dokončení operace (odemčení zámku může být provedeno ihned po přečtení řádku, tedy ještě před dokončením operace).

Také je dobré si uvědomit, že v tomto případě není řešením ani izolační úroveň SNAPSHOT, s typem deadlocku zapisovatel-zapisovatel. Důvodem je, že izolační úroveň SNAPSHOT ovlivňuje pouze sdílené zámky. Pro modifikaci dat jsou stále použity exkluzivní zámky. [5]

5.1.4 Deadlock při vyhledávání klíčů

Deadlocky vznikající při vyhledávání klíčů jsou technicky speciální případ deadlocku čtenář-zapisovatel. Tento typ deadlocku nepotřebuje procesy, které mají spuštěné transakce

s několika operacemi. Deadlock při vyhledávání klíčů je možné pro sezení dostat při spuštění jednoho dotazu SELECT, když jiné sezení má spuštěn jeden příkaz UPDATE.

Důvodem možnosti takového deadlocku je, že indexační architektura SQL Serveru a pořadí v jakém dotazovací procesor provádí operace. [5]

5.1.5 Deadlocky vzniklé paralelismem

Deadlocky vzniklé paralelismem se dělí na dva typy. Deadlockem vzniklý paralelním během dotazů anebo vlivem paralelismu uvnitř samotného dotazu.

Vnitro-dotazový deadlock je spíše chyba samotného SQL Serveru a neměla by nastat. [5]

5.1.6 Prohledávání rozsahu a SERIALIZABLE deadlocky

Izolační úroveň SERIALIZABLE je náchylná na deadlocky kvůli restriktivním zámkům a všechny zámky jsou drženy po dobu celé transakce. Izolační úroveň SERIALIZABLE předchází také čtení s výskytem fantomů ve výsledcích. Objevení fantomů je zabráněno tak, že izolační úroveň SERIALIZABLE zavede rozsahový zámek nad rozsahem, který je právě čten transakcí, a tím je zamezeno vložení nových řádků. Rozsahový zámek se týká existujících relevantních indexů anebo celé tabulky, pokud se v ní nenachází relevantní index. [5]

5.1.7 Deadlocky u oddílové eskalace

Tento typ deadlocku se vyskytuje na rozdělených tabulkách, kde tabulková zámková eskalace je nastavena na hodnotu AUTO. Nastavení AUTO způsobuje, že na rozdělené tabulce je umožněna zámková eskalace může jít do rozdělovací úrovně, místo aby šla do tabulkové úrovně. [5]

5.2 Detekce deadlocků v databázi

SQL Scheduler je sám schopen detekovat a vyřešit vzniklé deadlocky automaticky. K tomuto účelu se používá monitor zámků. Monitor zámků je proces běžící v pozadí od spuštění instance SQL Serveru a neustále sleduje systém a hledá možné deadlocky mezi sezeními. Pokud se začnou deadlocky objevovat, je už na správci databáze, aby se zabýval příčinou vzniku a jak je odstranit. [6]

5.2.1 Monitor zámků

Pokud je monitorem zámků nalezen deadlock, jedno ze sezení účastnící se deadlocku je označeno jako oběť deadlocku a jeho změny jsou vráceny zpět. Uvolněny jsou všechny jeho zámky nad zdroji a tím mohou ostatní sezení pokračovat dál ve své práci. Jakmile je dokončeno vracení změn oběti deadlocku, je vrácen chybové hlášení 1205 klientovi. [6]

SQL Server vybírá oběť deadlocku podle dvou následujících kritérií:

1. **Deadlocková priorita** – hodnota přiřazenému sezení určuje relativní důležitost dokončení jeho transakce, pokud dojde k deadlocku. Jako oběť deadlocku je vždy vybráno sezení s nejmenší prioritou.
2. **Cena vrácení změn** – pokud dvě či více sezení má stejnou prioritu a je detekován deadlock, tak SQL Server rozhodne, které sezení se stane obětí deadlocku podle odhadnutí „ceny“ za vrácení změn tohoto sezení. Zvoleno je sezení s odhadovanou nejnižší „cenou“ vrácení změn.

5.2.2 Příznak trasování 1204

Příznaky trasování v SQL Serveru povolují spuštění přídavného kódu, pokud je zapotřebí. Příznaky trasování mohou být povoleny pro jednotlivé sezení nebo pro všechny nad danou instancí.

Pokud je příznak trasování 1204 povolen pro všechny sezení nad SQL Server instancí, jakýkoliv deadlock detekovaný monitorem deadlocků způsobí zapsání diagramu deadlocku do SQL Server chybového logu.

Příznak trasování 1204 je jedinou možností k získání diagramu deadlocku v SQL Serveru 2000. V dalších verzích je nahrazen příznakem trasování 1222. [6]

5.2.3 Příznak trasování 1222

SQL Server 2005 přidal příznak trasování 1222, který změnil získané diagramy deadlocků při jejich výskytech. Principiálně funguje stejně jako příznak trasování 1204, ale podává diagramy o deadlocích, které jsou v ucelenějším formátu a tím se stávají jednodušším pro čtení. Informace jsou zachyceny a prezentovány způsobem, který je činí jednoduššími pro identifikování oběti deadlocku, zdroje a procesy které byly v deadlocku také zahrnuty.

Příznak trasování 1204 je stále dostupný kvůli zpětné kompatibilitě. Proto ve verzi SQL Server 2005 a novější by měl být pro zachycení deadlocku použit příznak trasování 1222. [6]

Podle oficiální dokumentace Microsoft [17] končí podpora příznaků trasování v SQL Server Database Engine 2016 verzi a v novějších se nebudou nacházet, proto by se mělo při návrhu využít spíše extend events.

5.2.4 SQL Profiler

Diagram deadlocku v SQL Trace je událost, která zachytí informace o deadlocku, bez toho aby byl diagram deadlocku zapsán do chybového logu SQL Serveru.

Výhodou použití SQL Profileru při prohlížení obsahu trasovacích souborů, zaznamenané deadlocky lze exportovat do jednotlivých XDL souborů, které mohou být otevřeny graficky v SQL Server Management Studiu. [6]

SQL Profiler je grafickou nástavbou SQL Trace a proto bude jeho podpora ukončena stejně jako SQL Trace, tedy poslední verzí ve které bude podporován je SQL Server Database Engine 2016. [18]

5.2.5 Notifikace událostí Service Broker

Od verze SQL Server 2005, notifikace událostí povolují zachycení informací diagramu deadlocku s využitím SQL Service Broker, vytvořením služby a dotazu pro trasování události diagramu deadlocku. Diagram deadlocku získaný tímto postupem se nijak neliší od diagramu deadlocku získaného pomocí příznaku trasování, liší se pouze mechanizmem získání.

K spuštění notifikace událostí k zachytávání diagramů deadlocků je potřeba tří Service Broker objektů:

1. **Dotaz** – k udržení zprávy o události DEADLOCK_GRAPH
2. **Služba** – nasměruje zprávu do fronty
3. **Notifikace události** – zachytí diagram deadlocku, zabalí jej do zprávy a tu pošle službě

[6]

5.2.6 WMI Provider pro serverové události

Jakýkoliv událost, která může být zachycena pomocí notifikací událostí, má odpovídající WMI Objekt Události a jakákoliv WMI manažerská aplikace může odebírat takovéto události.

SQL Server Agent byl updatován, aby mohl pracovat s WMI událostmi. Práce s těmito událostmi je zprostředkována WMI Dotazovacím Jazykem WQL, jazyk který je podobný T-SQL použitému s WMI a Agent Alerts pro WMI události. [6]

5.2.7 Extended Events

Před verzí SQL Server 2008, nebyla možnost zpětně dohledat informace o deadlocku. Pro získání diagramu deadlocku bylo nutné, aby SQL Trasování běželo aktivně, nebo byl pro instanci SQL Serveru zapnut příznak trasování 1222 nebo 1204. Jelikož trasování deadlocků jednou z metod může znamenat zbytečné navýšení zátěže a využití zdrojů serveru, se nejdříve počkalo na projevení několika deadlocků a poté bylo zapnuto trasování.

SQL Server 2008 obsahuje všechny výše popsané metody detekování deadlocků a přidává novou. Nová metoda detekce deadlocků umožňuje sběr informací skrze výchozí událostní sezení systém_health v Extended Events. Toto výchozí sezení událostí běží automaticky na všech instancích SQL Serveru a sbírá škálu užitečných informací pro řešení problémů, které se objeví v SQL Serveru, takže i deadlocků.

Diagram deadlocku zachycený pomocí Extended Events má jedinečnou vlastnost, kterou je zachycení informací o deadlocku, který vyžadoval více jak jednu oběť k jeho vyřešení. [6]

5.3 Řešení deadlocků

V takovém případě SQL Server detekuje deadlock a zasáhne. Zásahem SQL Serveru je jedna z transakcí ukončena. Pokud by SQL Server takto neprovedl, je možné, že transakce by takto byly blokovány navždy.

Pokud není nijak specifikováno, tak SQL Server vybírá k zániku transakci, která provedla nejméně práce (podle aktivity zapsané v logu transakcí). Důvodem je nejmenší vynaložené úsilí na navrcení do původního stavu. V případě deadlocku a zjištění, že transakce provedly stejnou práci, je využito hodu mincí pro rozhodnutí, která transakce bude ukončena. [4]

5.3.1 Deadlock vyhledávání klíčů

Pro zabránění deadlocku vznikajícího při vyhledávání klíčů je potřeba změnit definici neclusterizovaného indexu tak, aby obsahoval další sloupce potřebné pro pokrytí dotazu, jako klíčové sloupce nebo jako INCLUDE sloupce. [6]

5.3.2 Prohledávání rozsahu a SERIALIZABLE deadlocky

Deadlock tohoto typu se dá řešit několika způsoby a ten správný závisí na databázi a její aplikaci.

Pokud není nutné pro databázi, aby udržovala rozsahové zámky v průběhu SELECT operace, která prohledává a zjišťuje, zdali řádky existují. Tak je možné operaci SELECT přesunout mimo transakci, která provádí změnu dat, a tím zabránit deadlocku.

Další možností předcházení tohoto typu deadlocku je změna izolační úrovně. Pokud není opravdu nutné, aby transakce proběhla v izolační úrovni SERIALIZABLE, může pro odstranění deadlocku stačit přejít na nižší izolační úroveň READ COMMITTED, která odstraní deadlock a umožní větší stupeň paralelismu.

Pokud ani jedna z předchozích možností nevyřeší deadlock, je možné problém vyřešit pomocí donucení operace SELECT k použití kompatibilně nižšího zámku. Tohoto vynucení lze dosáhnout nastavením UPDLOCK nebo XLOCK v nápovědě tabulky. Zámek nižší kompatibility zabrání jakékoliv další transakci, která se pokusí získat zámek, v získání zámku vyšší kompatibility. Tento způsob opravy je specifický pro deadlock způsobený izolační úrovní SERIALIZABLE. Použití nastavení UPDLOCK při izolační úrovni READ COMMITTED může způsobit ještě větší množství deadlocků při určitých podmínkách. [6]

5.3.3 Kaskádové restrikce

Deadlock způsobený kaskádovým omezením je obecně velice podobný deadlocku rozsahového zámku izolační úrovně SERIALIZABLE, ačkoliv transakce, která se stala obětí deadlocku, samotná pod izolační úrovní SERIALIZABLE neběžela. K vynucení kaskádových omezení, SQL Server musí projít hierarchii cizích klíčů, aby zajistil zachování záznamů potomků bez rodičů, které jsou důsledkem aktualizační nebo odstraňovací operace nad tabulkou rodiče. K takovému zajištění je nutné, aby změny prováděné transakcí nad tabulku rodiče byly izolovány od efektů ostatních transakcí, z důvodu zabránění změny, která by porušila omezení dané cizím klíčem, až bude kaskádová operace následně dokončena.

V případě použití výchozí READ COMMITTED izolační úrovně, sezení získá a drží X zámky nad všemi řádky, které musí být změněny, po dobu trvání celé transakce. Tímto způsobem je zablokována uživateli možnost čtení nebo aktualizace jednoho z řádků určených ke změně původní transakcí. Tento přístup ale neznemožní jinému sezení v přidání nového řádku do tabulky potomka pro klíč rodiče, který je mazán. Aby se zabránilo takovému přidání fantomového řádku, databáze požádá a po získání drží zámek rozsahu, který zabrání v přidání nových řádků do rozsahu zasaženého kaskádovou operací. Tohle je prakticky skryté použití izolační úrovně SERIALIZABLE, pod dobu vynucení kaskádového omezení, ale izolační úroveň pro dávku není ve skutečnosti změněna, pouze typ zámků použitých pro kaskádové operace je změněn.

Při objevení deadlocku v průběhu kaskádové operace, první věc která by měla být ověřena je, zda existují clusterované nebo neclusterované indexy pro použité sloupce cizích klíčů. Pokud nad použitými sloupci cizích klíčů nejsou odpovídající indexy, získané zámky pro vynucení omezení budou drženy po delší čas a tím zvyšují šanci na vznik deadlocku mezi dvěma operacemi, pokud nastane konverze zámků. [6]

5.3.4 Paralelismus uvnitř dotazu

Tento typ deadlocku se objeví, když jedno sezení spustí dotaz běžící paralelně a sám sebe dostane do deadlocku. Na rozdíl od ostatních deadlocků v SQL Serveru, deadlocky tohoto typu mohou být způsobeny chybou v SQL Serverovém paralelním synchronizačním kódu, místo chyby v návrhu databáze nebo aplikace. Jelikož jsou rizika spojena s opravením některých chyb, může se stát, že chyba je známa, ale nebude opravena, protože je možno ji obejít zredukováním stupně paralelismu pro daný dotaz, použitím MAXDOP příznaku dotazu, nebo přidání či změněním indexů, aby se snížila nákladnost dotazu a tím se stal více efektivním.

Diagram deadlocku pro deadlock vzniklý při paralelním zpracování má stejné SPID pro všechny procesy a má aspoň dva procesy v seznamu procesů. Seznam procesů bude mít threadpool, exchangeEvent nebo obojí jako zdroj, ale nebude mít s nimi spojený zámek zdrojů. Dále také diagram deadlocku pro tento typ je znatelně větší, než diagram deadlocku pro jiný typ deadlocku, podle počtu použitého stupně paralelismu a počtu uzlů, které existovali ve spouštěcím plánu. [6]

5.3.5 Přístup k objektům v jiném pořadí

Jeden z nejjednodušších typů deadlocku na vytvoření a proto také jeden z nejjednodušších k předejití, je způsoben přístupem k objektům v databázi v jiném operačním pořadí uvnitř T-SQL zdrojovém kódu uvnitř transakce.

UPDATE operace transakce 1 nad tabulkou A vede na X zámek, který bude držet na tabulce až do dokončení transakce. Ve stejný čas transakce 2 spustí UPDATE nad tabulkou B, který také vede na X zámek, který bude držen až do dokončení transakce. Po dokončení aktualizační operace nad tabulkou A, transakce 1 se pokusí o přečtení tabulky B, ale je zablokována a její znemožněno získat potřebný S zámek, protože je nad tabulkou B držen X zámek transakcí 2. Po dokončení aktualizace tabulky B, transakce 2 začne číst tabulku A, kde je zablokována nemožností získat potřebný S zámek, protože nad tabulkou A je stále držen X zámek transakcí 1. Jelikož obě transakce blokují sami sebe, výsledkem je deadlock a monitor zámků zvolí jednu z transakcí jako oběť a ukončí ji, vrátí její provedené změny a dovolí druhé transakci pokračovat.

Při použití explicitních transakcí ve zdrojovém kódu je důležité, aby k objektům bylo přistupováno vždy ve stejném pořadí, aby se předešlo vzniku tomuto typu deadlocku. [6]

5.3.6 Zabránění chybám zpracováním deadlocků

Ve většině případů, stejný důvod způsobující časté blokování v databázi, jako je špatný návrh databáze, nedostatek indexů, špatně navrhnuté dotazy, nevhodná izolační úroveň a další, jsou také běžným důvodem pro vznik deadlocků. Ve většině případů vyřešením těchto problému, je možné předejít i vznikům deadlocků. Bohužel, v době kdy se deadlocky stanou problémem, už nemusí být možné provést změny k napravení takových návrhových chyb.

Proto je defensivní programování důležitou součástí aplikace a databázového návrhu. Defensivní programování je technika, která očekává a zpracovává výjimky jako část svého obecného kódu pro aplikaci nebo databázi. Zpracování deadlocků pomocí defenzivního programování může být implementováno dvěma způsoby:

1. Na straně databáze – je využito bloků T-SQL TRY…CATCH
2. Na straně aplikace – je využito TRY…CATCH bloků aplikace

V obou případech dobré zpracování výjimky 1205, která je vyvolána SQL Serverem pro oběť deadlocku, umožní vyhnutí se nezpracovaným chybám (z anglického „UnhadledException“) v aplikaci. [6]

5.3.7 T-SQL TRY…CATCH blok

V závislosti na tom, jak je aplikace navrhnuta, jestli je v aplikaci rozdělovač mezi zdrojovým kódem aplikace a zdrojovým kódem databáze, tak nejjednodušší implementací pro zpracování deadlocku může být použití a spuštění BEGIN TRY/CATCH bloku uvnitř T-SQL.

Tato technika je nejvíce aplikovaná v případech, kdy aplikace volá uložené procedury pro všechny její přístupy k datům. V takovém případě, změna zdrojového kódu v procedurách zajistí zpracování deadlocku a nemusí se nijak zasahovat do zdrojového kódu ve zdrojovém kódu aplikace, jejím překompilováním a opětovné distribuci. Tento postup velice zjednodušuje implementaci takových změn.

Nejlepší cesta k pracování s deadlockem, závisí na aplikaci a jejím očekávaném chování v případě objevení deadlocku ve zdrojovém kódu pro zpracování deadlocku. Jedna možná cesta ke zpracování deadlocku je spouštět transakci dokola, dokud nespustí výjimku a tím se spustí její zpracování v aplikaci. Situace křížového zamykání spojeného s deadlockem trvá většinou jen velmi krátkou dobu, v řádech milisekund, takže s velkou pravděpodobností opětovné spuštění transakce zvolené jako oběť deadlocku, bude úspěšně provedena a nebude potřeba vyvolávat žádnou výjimku pro aplikaci.

Nicméně je také možné, že deadlock se bude vyskytovat i nadále a bude potřeba se vyhnout uváznutí v nekonečné smyčce, která se bude snažit do nekonečna spustit stejný neúspěšný zdrojový kód. Tomu se předejde použitím proměnné, která je použita jako snižující se čítač možných pokusů, jakmile proměnná klesne na nulu, je výjimka delegována do aplikace. [6]

Zdrojový kód pro zpracování deadlocku v T-SQL:

```
DECLARE @retries INT ;
SET @retries = 4 ;

WHILE ( @retries > 0 )
      BEGIN
            BEGIN TRY
                  BEGIN TRANSACTION ;

             -- place sql code here
                  SET @retries = 0 ;

                  COMMIT TRANSACTION ;
            END TRY
            BEGIN CATCH
            -- Error is a deadlock
                  IF ( ERROR_NUMBER() = 1205 )
                        SET @retries = @retries – 1 ;

            -- Error is not a deadlock
                  ELSE
```

```
                    BEGIN
                        DECLARE @ErrorMessage NVARCHAR(4000) ;
                        DECLARE @ErrorSeverity INT ;
                        DECLARE @ErrorState INT ;

                        SELECT    @ErrorMessage = ERROR_MESSAGE() ,
                                  @ErrorSeverity = ERROR_SEVERITY() ,
                                  @ErrorState = ERROR_STATE() ;

                        -- Re-Raise the Error that caused the 53roblém
                        RAISERROR (@ErrorMessage, -- Message text.
                            @ErrorSeverity, -- Severity.
                            @ErrorState – State.
                            ) ;
                        SET @retries = 0 ;
                    END

                IF XACT_STATE() <> 0
                    ROLLBACK TRANSACTION ;
            END CATCH ;
        END ;
GO
```

5.3.8 Zachycení ADO.NET SQL výjimky v .NET aplikaci

Klientská aplikace by měla být navrhnuta ke zpracování výjimku vyvolaná SQL Serverem.

Rozdíl mezi zpracováním výjimky deadlocku není v .NET aplikaci nějak rozdílné od zpracování v T-SQL. TRY…CATCH blok je použit ke spuštění SQL volání z aplikace a k chycení jakékoliv vyvolané výjimky od SQL Serveru. Pokud se aplikace pokusí o znovu spuštění operace v případě vzniklého deadlocku, měl by počet spuštění být omezen na konečný počet, aby nedošlo k uváznutí aplikace v nekonečné smyčce neúspěšných pokusů o spuštění.

Zdrojový kód pro zpracování SQL výjimky v .NET aplikaci:

```
int retries = 4;
while (retries > 0)
{
    try
    {
        retries = 0;
    }
    catch (SqlException exception)
    {
        // exception is a deadlock
        if (exception.Number == 1205)
        {
                // Delay processing to allow retry.
                Thread.Sleep(500);
                retries --;
        }
        // exception is not a deadlock
        else
        {
          throw;
        }
    }}
```

Než spouštět operaci znovu, může být výjimka zapsána do logu Windows Application Event Log, nebo vyvoláno upozornění pro uživatele, a podle jeho vstupu pokračovat dále. Tyto dva způsoby jsou jedny z několika možných, pro řešení vzniklého deadlocku a jeho řešení na úrovni klientské aplikace, která umožňuje flexibilnější zpracování než zpracování v databázi. [6]

5.3.9 Výběr oběti deadlocku podle priority

Jsou případy, kdy je vhodné označit, který proces bude zvolen jako oběť deadlocku, pokud deadlock nastane, než nechat zvolení oběti na SQL Serveru, který zvolí transakci s nejnižšími zdrojovými náklady na vrácení všech změn. Takovým případem může být důležitá zpráva, která provádí dlouho běžící SELECT operaci, a musí být dokončena i v případě, že nastane deadlock a transakce provádějící zprávu je ideálním kandidátem na oběť deadlocku.

Nastavit prioritu procesu v případě vzniklého deadlocku lze v SQL následovně:

```
-- Set a Low deadlock priority
SET DEADLOCK_PRIORITY LOW ;
GO

-- Set a High deadlock priority
SET DEADLOCK_PRIORITY HIGH ;
GO

-- Set a numeric deadlock priority
SET DEADLOCK_PRIORITY 2 ;
```

Proces běžící v dávce nebo sezení s nejnižší prioritou pro deadlock, bude zvolen jako oběť deadlocku a zrušen místo ostatních procesů s vyšší prioritou. Jako všechny SET nastavení v SQL Serveru, DEADLOCK PRIORITY je efektivní pouze v rámci aktuálního spuštění. Jeli priorita pro deadlock nastavena uvnitř uložené procedury, tak je nastavená priorita platná po celou dobu běhu procedury a po dokončení procedury je vrácena zpět na výchozí nastavení.

Na výběr je mezi třemi jmenovitými možnostmi LOW, NORMAL, HIGH a také možnost nastavení číselné hodnoty z rozsahu mínus deset až plus deset. Pro větší možnost nastavení úrovně mezi různými procesy.

Priorita pro deadlock je nastavena v okamžik spuštění, a každý uživatel má možnost upravit prioritu pro deadlock. Taková možnost může být problém, pokud uživatelé používají ad hoc dotazový přístup k SQL Serveru a nastaví svou prioritu vyšší než ostatní procesy, z důvodu zajištění nevybrání jejich procesu jako oběti pro deadlock. [6]

II. PRAKTICKÁ ČÁST

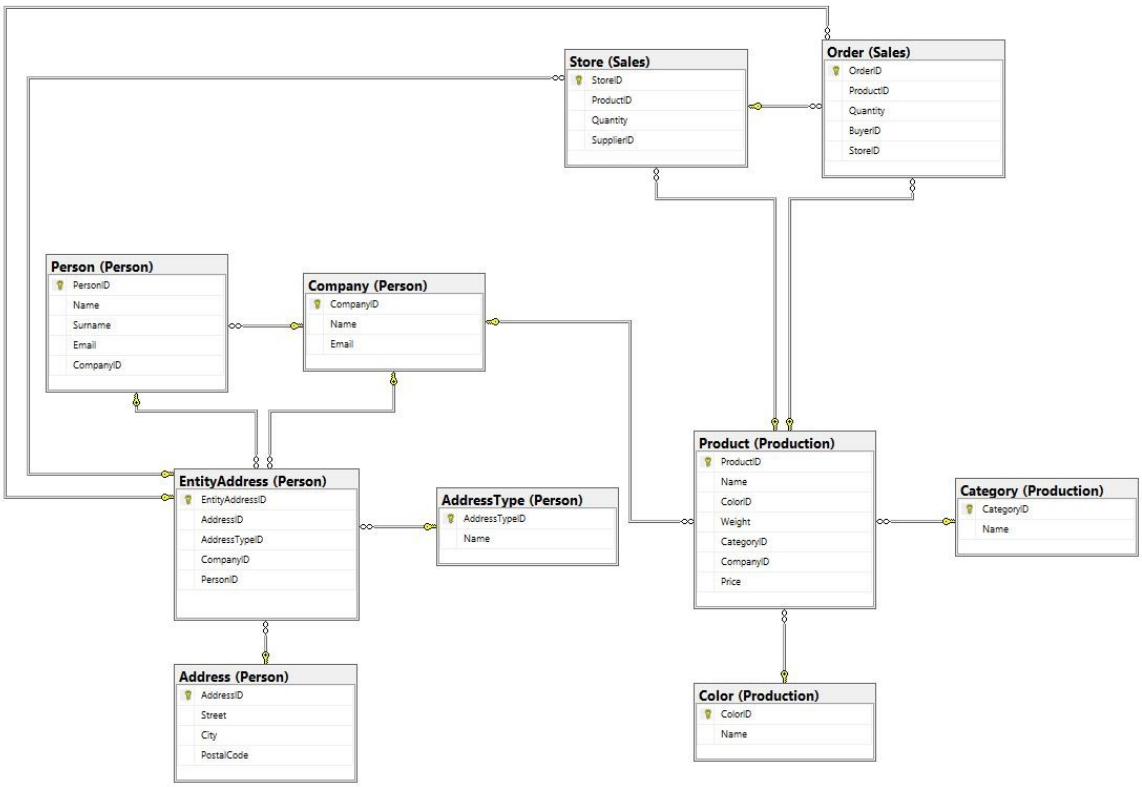
6 TESTOVACÍ DATABÁZE

Pro testování deadlocků jsem navrhnul jednoduchou databázi, která by mohla podporovat systém internetového obchodu. Jelikož se jedná o testovací a hlavně příkladovou databázi, tak jsem vynechal informace jako jedinečný identifikátor záznamu (ROWGUID) a také zaznamenání data poslední úpravy záznamu. Tyto informace jsou pro ukázku zanedbatelné, a tudíž byly vynechány, jelikož se nejedná o komerční využití databáze.

6.1 Struktura testovací databáze

Mnou navržená databázová struktura přebírá základní strukturu AdventureWorks pro internetový obchod, akorát je zjednodušena. Databáze AdventureWorks má ve své struktuře asi padesát tabulek a pro účely testování je tato struktura zbytečně komplexní a redundantní.

Proto jsem strukturu databáze ořezal na nejpotřebnější části. Tyto části jsem stejně jako v AdventureWorks rozdělil na menší celky pomocí schémat.



Obr. 12. Schéma testovací databáze

Použitá schémata jsou:

1. Person
2. Production
3. Sales

První schéma „Person“ neboli Osoba obsahuje údaje o všech fyzických, či právnických osobách v systému. Druhé schéma „Production“ neboli Produkce zahrnuje všechno, co se týká prodávaných produktů. Třetí schéma „Sales“ neboli Prodeje zahrnuje inforamce spojené s prodejem samotného produktu.

6.1.1 Schéma Person

Schéma Person obsahuje tabulky:

1. Person – informace o fyzické osobě
2. Company – informace o právnické osobě
3. Address – informace o adrese
4. EntityAddress – propojovací tabulka osoby a adresy
5. AddressType – tabulka o typu adresy

Každému záznamu v jednotlivých tabulkách je přiřazeno unikátní identifikační číslo, které je primárním klíčem a při odkazování do jiných tabulek se stává cizím klíčem.

Tabulka person obsahuje informace o fyzické osobě, jako je jméno, příjmení, email a s jakou společností je osoba svázána.

Tabulka company obsahuje záznamy všech firem a jejich údajů. V tomto případě se jedná o název firmy a její emailovou adresu pro řešení objednávek.

V tabulce Address jsou obsaženy informace o určité adrese, jako je ulice, město a směrovací číslo.

V tabulce AddressType jsou uvedeny názvy typu adresy, jako je například „shipping“ což označuje doručovací adresu apodobně.

V tabulce EntityAddress je spojena určitá adresa s jednotlivou právnickou či fyzickou osobou a také o jaký typ adresy se jedná.

6.1.2 Schéma Production

Schéma Production obsahuje tabulky:

1. Product – informace o produktu
2. Color – informace o barvě
3. Category – informace o kategorii

V tabulce produkt jsou zaznamenány informace o jednotlivých produktech, jako identifikační číslo produktu, jeho název, váha, cena, firma která produkt vyrábí, barva produktu a do jaké kategorie produkt spadá. Barva, kategorie a výrobce jsou cizí klíče do odpovídajících tabulek.

Tabulky Color a Category nesou pouze názvy a stejně jako tabulka AddressType slouží pouze jako číselníky typů.

6.1.3 Schéma Sales

Schéma Sales obsahuje tabulky:

1. Order
2. Store

Tabulka Order obsahuje informace o jednotlivých objednávkách. To jsou identifikační číslo objednávky, jaký produkt byl objednán a v jakém množství, kdo je objednavatelem a z jakého skladu si produkt objednal. Objednavatel je zde cizím klíčem do tabulky EntityAddress, protože je jednodušší a méně náročné najít osobu nebo firmu v odpovídající tabulce, kde je méně záznamů, než podle identifikačního čísla hledat osoby procházet celou tabulku EntityAddress a porovnávat, zda je daný záznam adresou osoby.

Tabulka Store obsahuje informace o skladech, kde jsem vynechal název a adresu a ponechal pouze informaci o identifikačním číslu skladu, produktu ve skladu a jeho množství a dodavatelem pro daný produkt. Dodavatel je opět cizí klíč z tabulky EntityAddress.

6.2 Rozdíly testovacích databází

Pro testování byly vytvořeny tři databáze, které jsou shodné ve struktuře databáze, uložených procedurách a hodnotách uložených v tabulkách AddressType, Color a Category.

Kde první databáze „*DiplomaDB_1*“ ukazuje navrženou databázi nejvíce nakloněnou pro vznik deadlocku. Jedná se o databázi, která nemá navržen žádný způsob obrany či odstranění deadlocků.

Druhá databáze „*DiplomaDB_2*“ využívá clusterovaných indexů, které by měly do určité míry odstranit deadlocky, jelikož je indexování jednodušší a rychlejší.

Třetí databáze „*DiplomaDB_3*“ využívá clusterovaných indexů v databázi a také je přidána optimalizace uložených procedur. Kde způsob jakým je procedura napsána, jestli přistupuje k tabulkám v opačném pořadí než jiná procedura, jestli je v proceduře více jak jeden druh operace (například čtení dat a jejich následná aktualizace), hraje významnou roli ve vzniku deadlocků v databázi.

6.3 Uložené procedury

K databázi jsou vytvořeny i uložené procedury (z anglického „Stored Procedures“). Tyto procedury jsou poté volány z aplikace zátěžového testu.

Každá tabulka v databázi, kromě typových tabulek, má vytvořenu proceduru pro vložení nového záznamu se všemi parametry záznamu a vrací index po vložení. Dále proceduru pro získání všech záznamů z tabulky a smazání všech záznamů z tabulky.

Poté jsou k tabulkám vytvořeny další procedury, například procedura pro získání všech záznamů z tabulky odpovídajícím danému parametru, procedura pro získání záznamů pro získání všech záznamů a záznamů s nimi spojenými v ostatních tabulkách.

K tabulkám Person, Order, Store, Product, Company a Address jsou vytvořeny procedury s názvem „SelectToUpdate“, které v rámci procedury, volají další dvě procedury. Nejdříve je zavolána procedura, která dotazem SELECT vybere veškeré záznamy z dané tabulky a další procedura způsobí aktualizaci všech záznamů v tabulce.

Procedury „SelectToUpdate“ byly vytvořeny pro ukázku optimalizace procedur, kde v databázi DiplomaDB_3 jsou upraveny tak, že procedury „SelectToUpdate“ provádí pouze aktualizační část.

7 ZÁTĚŽOVÝ TEST

Zátěžový test je vytvořen jako webová aplikace, která vytíží zvolenou databázi podle zvolených parametrů testu. Po dokončení testu jsou uživateli prezentovány výsledky testu.

7.1 Technologie použité v aplikaci

Aplikace je vyvinuta v ASP .NET, kde byl použit .NET Framework 4.6.2.

V aplikaci je použit návrhový model MVC („Model-View-Controller“), který rozděluje vzhled stránek prezentovaných uživateli, model nesoucí data a controller, který obsahuje logiku aplikace.

Vzhled stránek je definován ve značkovém jazyku Razor, který umožňuje použití programovacího jazyku C# přímo ve vzhledu webové stránky. Například dynamické vykreslení všech položek v seznamu apod.

Jedná se o aplikaci využívající paralelního programování. V aplikaci jsou požadavky na databázi spouštěny jako úkoly (tzv. „Tasks“). Úkoly jsou spuštěny pomocí funkce Run, která je okamžitě po vytvoření zařazuje do fronty pro spuštění.

V aplikaci jsou využity vláknově bezpečné kolekce („Thread Safe Collections“) pro uchování dat. Jedná se o kolekce do kterých může více vláken přidávat a odebírat položky, aniž by vyžadovaly další synchronizaci. Tyto kolekce jsou v aplikaci využity pro udržení aktuálních, fronty vzniklých výjimek a frontu vytvořených úkolů.

Pracování s úkoly bylo upřednostněno před využitím příznaku „parallel“ pro smyčky vytvářející požadavky na databázi, jelikož příznak „parallel“ může být plánovačem optimalizován a ve výsledku spuštěn pouze v jednom pracovním vlákně, čímž by došlo k potlačení paralelismu aplikace.

Pro připojení k databázi nebyl využit Entity Framework. Funkce jsou napsány pro větší kontrolu nad zdrojovým kódem aplikace a jeho lepší možnosti ladění.

Pro vývoj aplikace bylo využito vývojové prostředí Microsoft Visual Studio 2017 Community, SQL Server Management Studio verze 17.0 a SQL Server 2016 – Database Engine. Pro server webové aplikace bylo využito služby Internet Information Service, která je součástí operačního systému Windows. V této službě byl webové aplikaci přiřazen samostatný thread pool.

7.1.1 Nastavení testu

Pro spuštění zátěžového testu je nutné vyplnit parametry, které jsou od testu požadovány. V aplikaci je možné nastavit, která databáze bude použita pro zátěžový test. Na výběr je ze všech tří databází zmíněných výše. Dále je možné nastavit, pod jakou izolační úrovní budou transakce spouštěny.

Dále je možno nastavit parametr *Iterations*, který určuje počet opakování jednotlivého nastavení ostatních parametrů. Dalším parametrem k určení je počet nastavení, které má test vyzkoušet. Počet záznamů v databázi se v nastaveních testu zvyšuje podle rovnice popsané v kapitole *Výsledky testu* (*8 Výsledky Testu*). Počet opakování se zvyšuje o tři v každém následujícím nastavení.

Dále je možné určit, zda výsledky získané v průběhu testu mají být uloženy do souboru a poté vyhodnoceny v jiném programu například. Samozřejmostí je možnost nastavení názvu vytvořeného souboru. Soubor je vždy uložen s příponou *.csv* a je použito čárky pro oddělení hodnot a desetinné tečky pro čas.

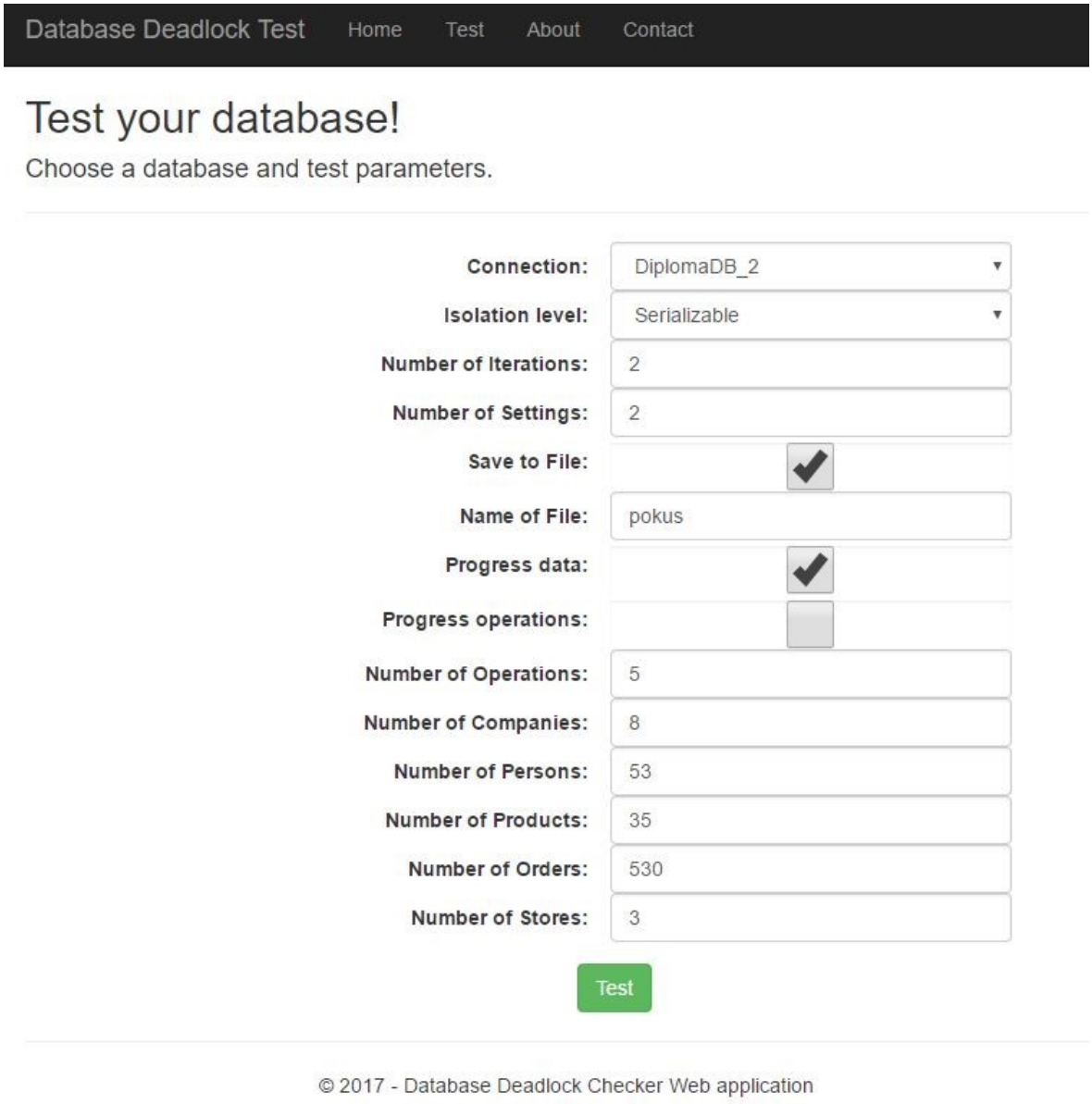
Dále jsou zaškrtávátka pro zvolení parametru, který bude zvyšován napříč nastaveními. Možné je postupně zvyšovat jen počet záznamů nebo jen počet operací nebo zvyšování jak záznamů, tak i počtu operací.

Poté je nutno nastavit kolik záznamů se má vytvořit v každé tabulce. Počet adres a počet záznamů v tabulce *EntityAddress* je dopočítán podle rovnic:

$$počet\ adres = (počet\ lidí + počet\ firem) * 1{,}2$$

$$počet\ EntityAddress = 1{,}3 * počet\ adres$$

Obě vypočítané hodnoty jsou poté zaokrouhleny na celá čísla a dále se zvyšují také podle odpovídající rovnice pro zvyšování počtu záznamů.



Obr. 13. Nastavení parametrů testu

7.1.2 Výsledek testu

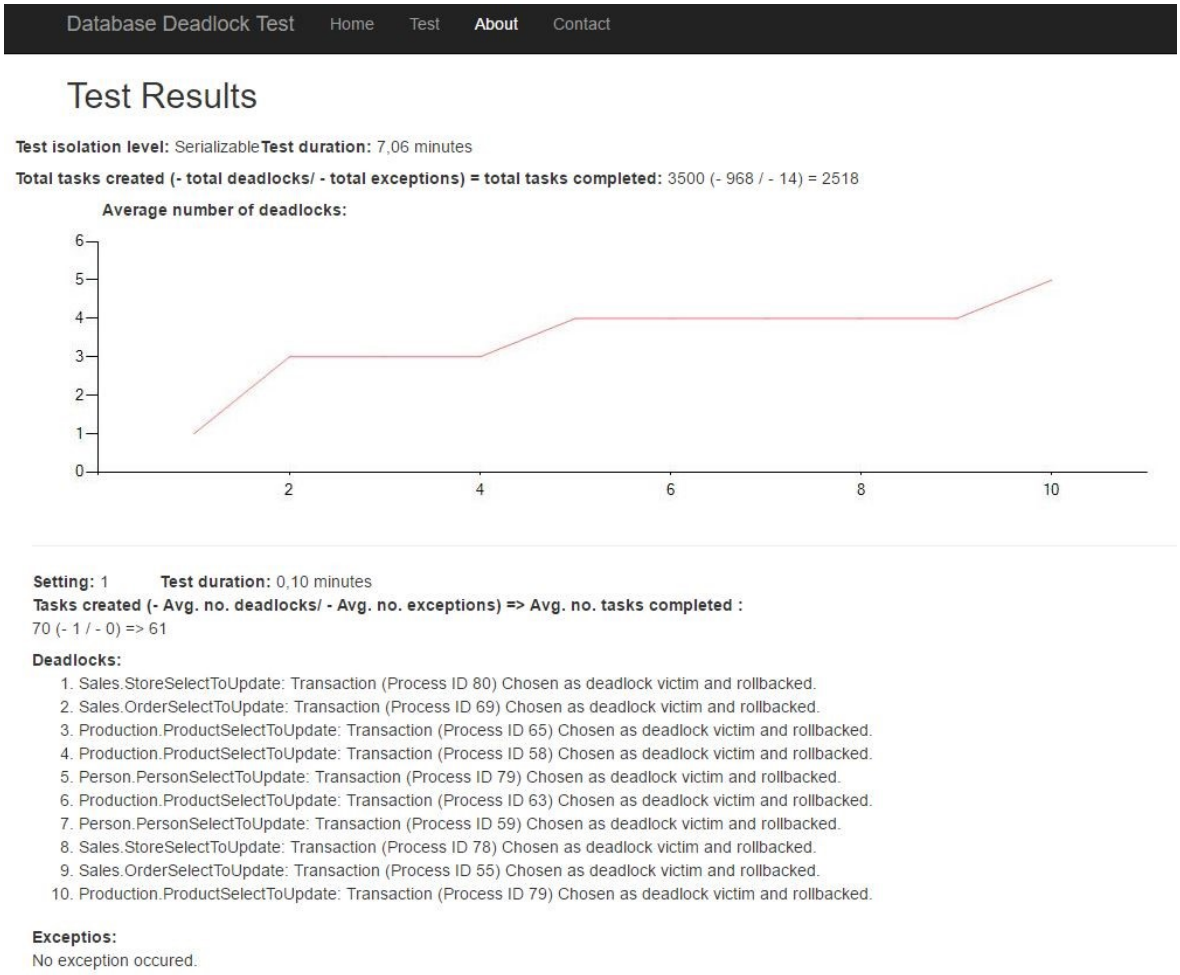
Po proběhnutí testu jsou zobrazeny informace o průběhu testu.

Je uživateli opět zobrazeno s jakou izolační úrovní byl test proveden a jak dlouho trvalo jeho proběhnutí.

Dále je uživateli zobrazena informace o celkovém počtu vytvořených požadavků na databázi, po jejím vyčištění a znovu vygenerování dat. Od tohoto počtu požadavků je odečten

celkový počet požadavků ukončených předčasně, protože byly zvoleny jako oběti deadlocku, a také odečten počet požadavků, které skončili v jiné výjimce. Výsledkem tedy je celkový počet provedených požadavků na databázi. Tyto počty neobsahují požadavky pro zápis nově vygenerovaných dat ani požadavky pro vyčištění databáze.

Dále je uživateli zobrazen graf vývoje průměrného počtu deadlocků napříč nastaveními.



Obr. 14. Zobrazení výsledku testu

Pod grafem jsou zobrazeny výsledky pro jednotlivé nastavení. Uživatel může vidět číslo nastavení a jak dlouho probíhal test nastavení. Dále jsou uživateli zobrazeny hodnoty průměrného počtu vytvořených požadavků, průměrného počtu deadlocků, průměrného počtu výjimek a průměrného počtu úspěšně dokončených požadavků.

8 VÝSLEDKY TESTU

Pro otestování a následné možné srovnání výsledků jednotlivých testů bylo zvoleno patnáct nastavení počtů záznamů pro jednotlivé tabulky. Tyto nastavení jsou získány inkrementálně z rovnice:

$$parametr_{iterace+1} = (parametr_{iterace} * (iterace + 1) * 0{,}0745) + parametr_{iterace}$$

Kde *parametr* je zvolený počet záznamů v tabulce a *iterace* je celé nezáporné číslo označující číslo nastavení. Tato rovnice byla zvolena pro její progresivní zvyšování záznamů. Výhodou je pozvolné zvyšování záznamů do desáté iterace a od vyššího čísla iterace je již zvyšování počtu záznamů okolo dvojnásobku. Takto zvoleným progresivním přírůstkem je možné prozkoumat, jaký dopad na výskyt deadlocků má malé zvýšení záznamů a jaký dopad má naopak velké zvýšení záznamů v jednotlivých tabulkách.

Tab. 3. Nastavení počtu záznamů

		Companies	Persons	Products	Orders	Stores	Addresses	Entities
Nastavení	0	8	53	35	530	3	73	95
	1	9	57	38	569	3	79	102
	2	10	65	43	654	4	90	117
	3	12	80	53	801	5	111	144
	4	16	104	69	1,039	6	144	187
	5	22	143	94	1,426	8	197	256
	6	31	206	136	2,064	12	285	371
	7	47	314	207	3,140	18	434	564
	8	76	501	331	5,011	28	692	900
	9	126	837	553	8,372	47	1,156	1,503
	10	221	1,461	965	14,609	83	2,018	2,623
	11	401	2,658	1,755	26,580	150	3,671	4,772
	12	760	5,034	3,325	50,343	285	6,953	9,039
	13	1,496	9,910	6,544	99,101	561	13,687	17,793
	14	3,056	20,246	13,370	202,462	1,146	27,963	36,352
	15	6,471	42,871	28,311	428,714	2,427	59,211	76,974

Pro testy zjišťující dopad počtu požadavků na vznik deadlocků nebyla využita rovnice jako pro výpočet nastavení parametrů, ale počet požadavků je pro každé nastavení zvýšen o tři. Počet záznamů v databázi je tedy stejný po celou dobu testu a počet záznamů odpovídá nastavení šest z tabulky nastavení počtu záznamů (*Tab. 3.*).

Tab. 4. Nastavení počtu požadavků

	Nastavení																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Operati-ons	5	8	11	14	17	20	23	26	29	32	35	38	41	44	47	50	53	56	59	62	65

Testy zjištění vlivu počtu záznamů v databázi na vznik deadlocků byly provedeny pro tři výše zmíněné databáze (bod *6.2 Testovací databáze*) a jejich izolační úrovně READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ a SERIALIZABLE. Pro každé z patnácti nastavení počtu záznamů bylo provedeno deset opakování.

Testy zjištění vlivu počtu požadavků na vznik deadlocků byly provedeny pro tři výše zmíněné databáze (bod *6.2 Testovací databáze*) a jejich izolační úrovně READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ a SERIALIZABLE. Pro každé z dvaceti nastavení počtu záznamů bylo provedeno dvacet opakování. V jednom cyklu je provedeno čtrnáct požadavků na databázi. Takže například první nastavení počtu operací znamená, že na databázi je vytvořeno sedmdesát požadavků na databázi apod. Jinak řečeno parametr Operations udává počet opakování čtrnácti požadavků na databázi.

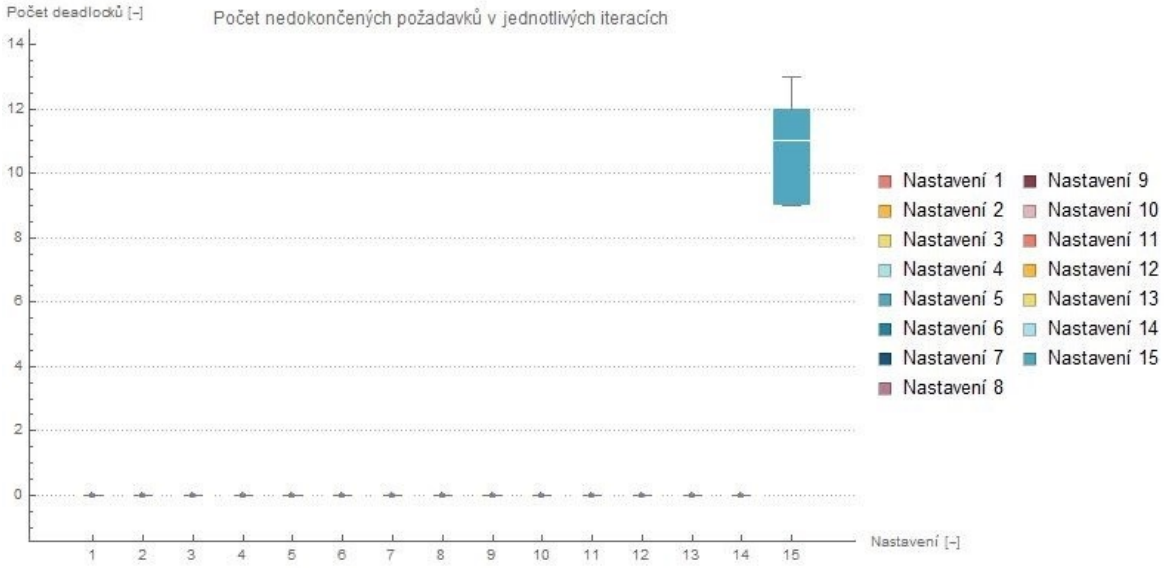
Nastavení patnácté pro počet záznamů v databázi je maximální možný, který bylo možno otestovat z důvodu paměťové složitosti. Při větším počtu záznamů docházelo k restartování služby Internet Information Services a tím také restartování celého testu.

V testu nejsou žádným způsobem přidávány nebo odebírány data. Jedním z důvodů je proměnlivý počet záznamů, který by se takto měnil. Druhým důvodem je také fakt, že operace vkládání nových záznamů a mazaní stávajících ovlivní pouze výsledky pro SERIALIZABLE izolační úroveň, protože využívá zámků rozsahu. Tudíž není možné do rozsahu přidat záznam či z něj smazat záznam, ale ostatní izolační úrovně by rozdíl nezaznamenaly, jelikož tyto operace nezahrnují práci nad více záznamy naráz, nejsou konfliktní s operacemi čtení či aktualizačními operacemi.

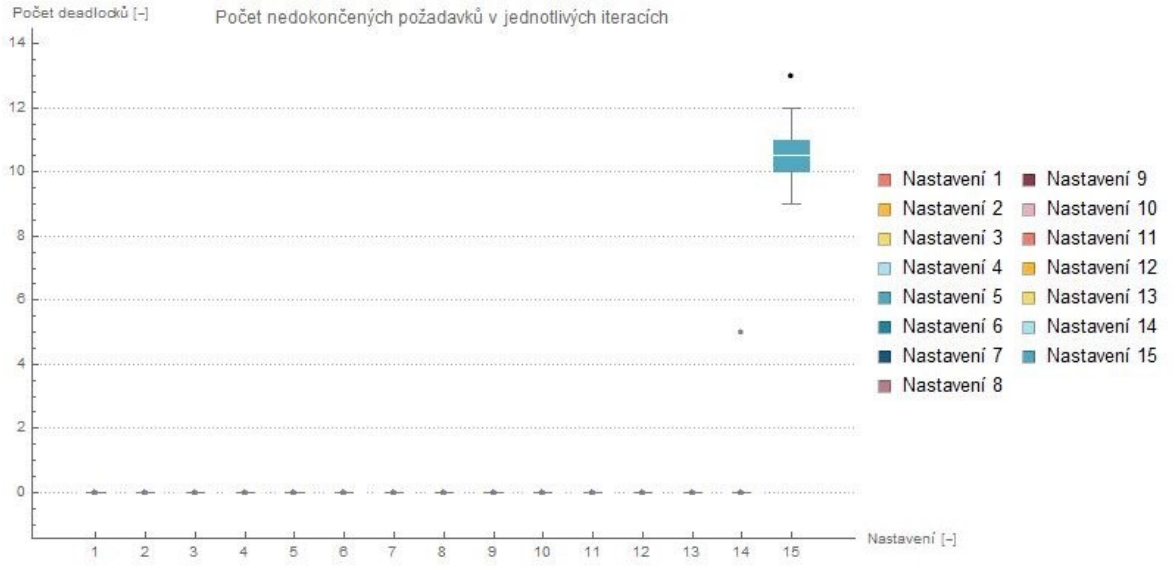
8.1 Výsledky testu pro izolační úroveň READ UNCOMMITTED

U izolační úrovně READ UNCOMMITTED se ve výsledcích testu projevila její vlastnost, že čtenář nežádá o S zámek a tím se neblokují aktualizační požadavky s čtecími požadavky.

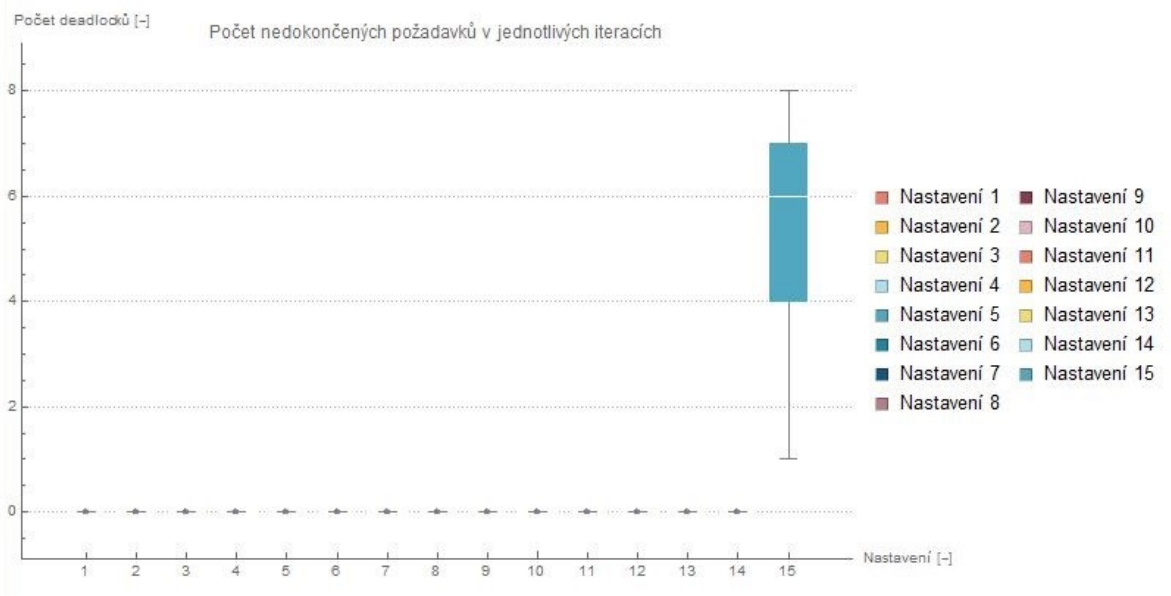
Jelikož je využito výchozího nastavení pro požadavky do databáze, tak je čas pro vykonání požadavku nastaven na třicet sekund. Pokud není požadavek do této doby proveden, je vrácena chyba o překročení tohoto limitu.



Obr 15. Krabicový graf výskytu deadlocků pro izolační úroveň READ UNCOMMITTED v závislosti na počtu záznamů pro databázi s neclusterovanými indexů

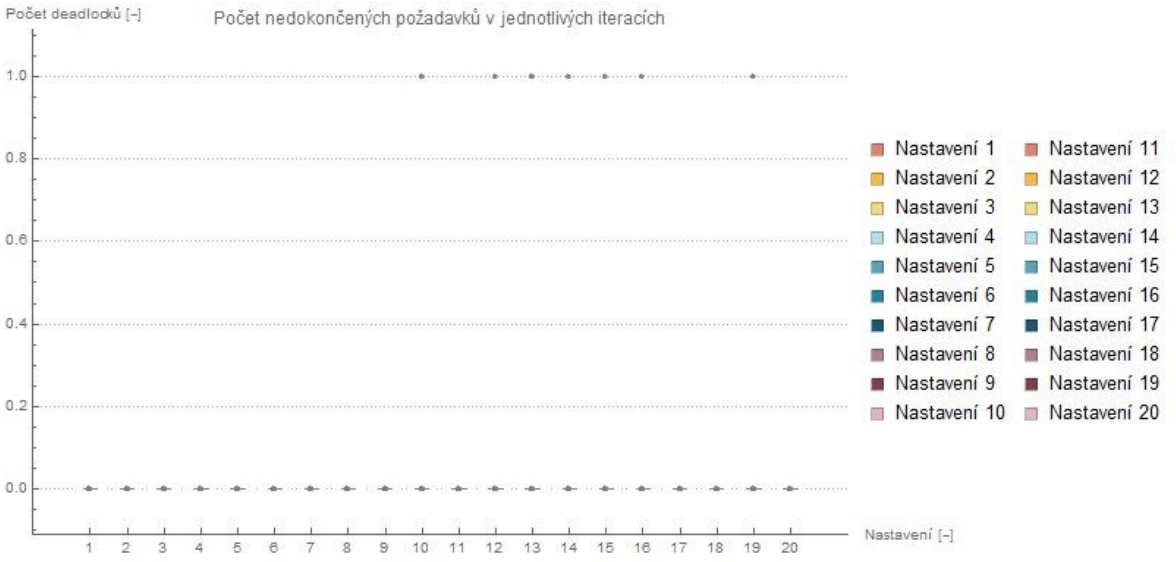


Obr. 16. Krabicový graf výskytu deadlocků pro izolační úroveň READ UNCOMMITTED v závislosti na počtu záznamů pro databázi s clusterovanými indexy

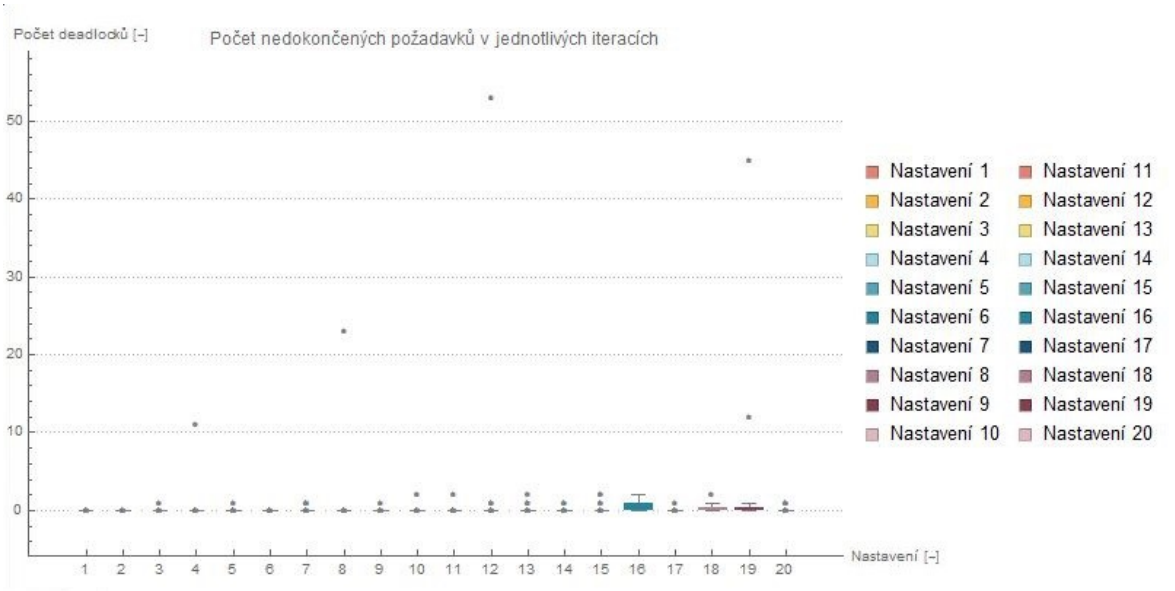


Obr. 17. Krabicový graf výskytu deadlocků pro izolační úroveň READ UNCOMMITTED v závislosti na počtu záznamů pro databázi s optimalizovanými procedurami

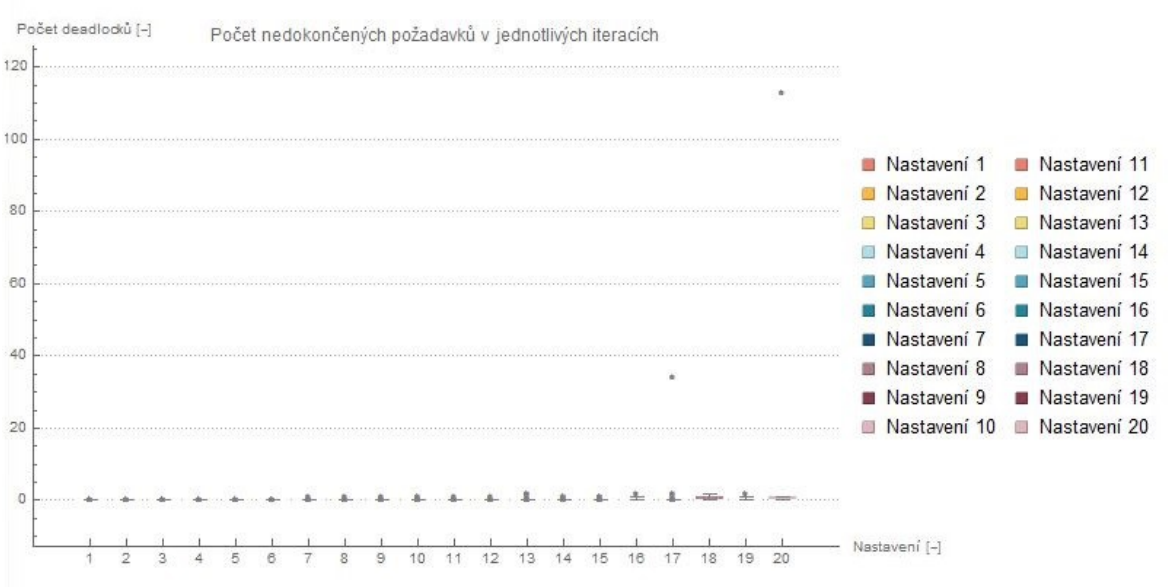
Z krabicových grafů (*Obr. 15., Obr. 16., Obr. 17.*) lze vidět nulový výskyt nedokončených požadavků napříč nastaveními. Až v případě patnáctého nastavení se objevují nedokončené požadavky, které jsou ukončeny z důvodu vypršení času pro provedení požadavku. Takže požadavek nekončí v deadlocku, ale je zrušen kvůli velké prodlevě ve spuštění, nebo získání výsledku.



Obr. 18. Krabicový graf výskytu deadlocků pro izolační úroveň READ UNCOMMITTED v závislosti na počtu požadavků na databázi s neclusterovanými indexů

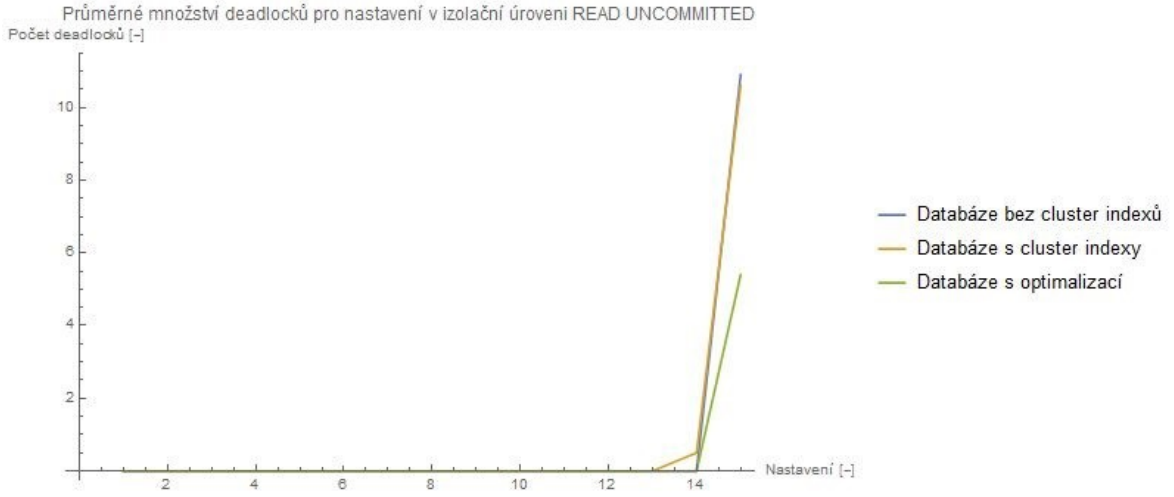


Obr. 19. Krabicový graf výskytu deadlocků pro izolační úroveň READ UNCOMMITTED v závislosti na počtu požadavků na databázi s clusterovanými indexy



Obr. 20. Krabicový graf výskytu deadlocků pro izolační úroveň READ UNCOMMITTED v závislosti na počtu požadavků na databázi s optimalizovanými procedurami

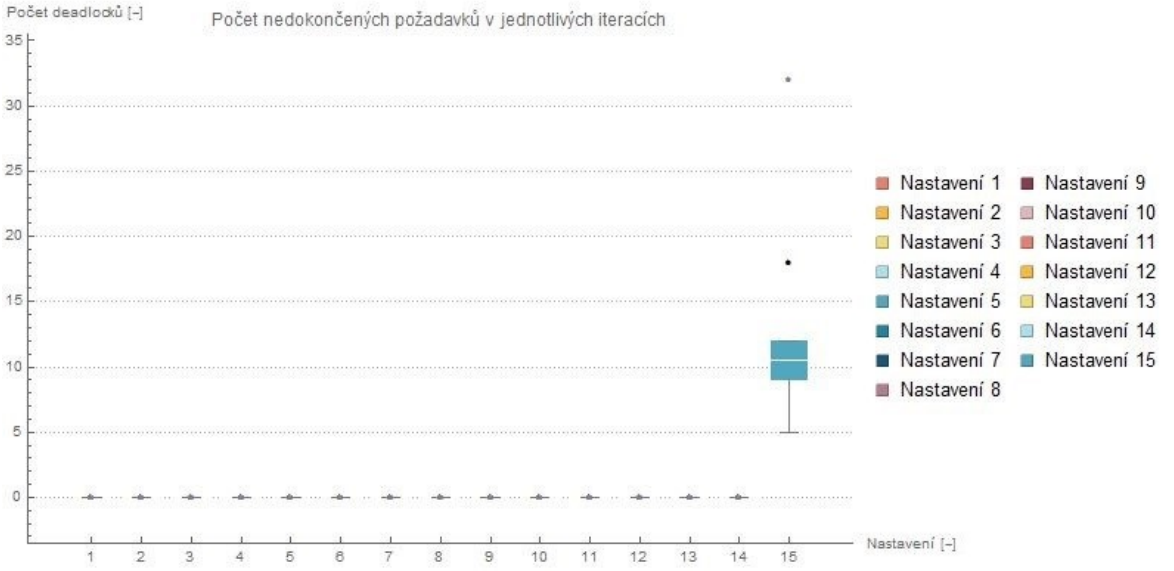
Z krabicového grafu (*Obr. 18*) lze vidět nulový výskyt nedokončených požadavků napříč nastaveními. Tentokráte ani v případě nejvyššího nastavení se neobjevují nedokončené požadavky pravidelně, pouze ojedinělé případy jednoho nedokončeného požadavku. Obdobného výsledku je dosaženo také u obou dalších databází, kde z grafů (*Obr. 19., Obr. 20.*) lze vidět spíš ojedinělý výskyt nedokončených požadavků dosahující řádu desítek.



Obr. 21. Srovnání vývoje průměrného počtu nedokončených požadavků pro izolační úroveň READ UNCOMMITTED pro všechny testovací databáze

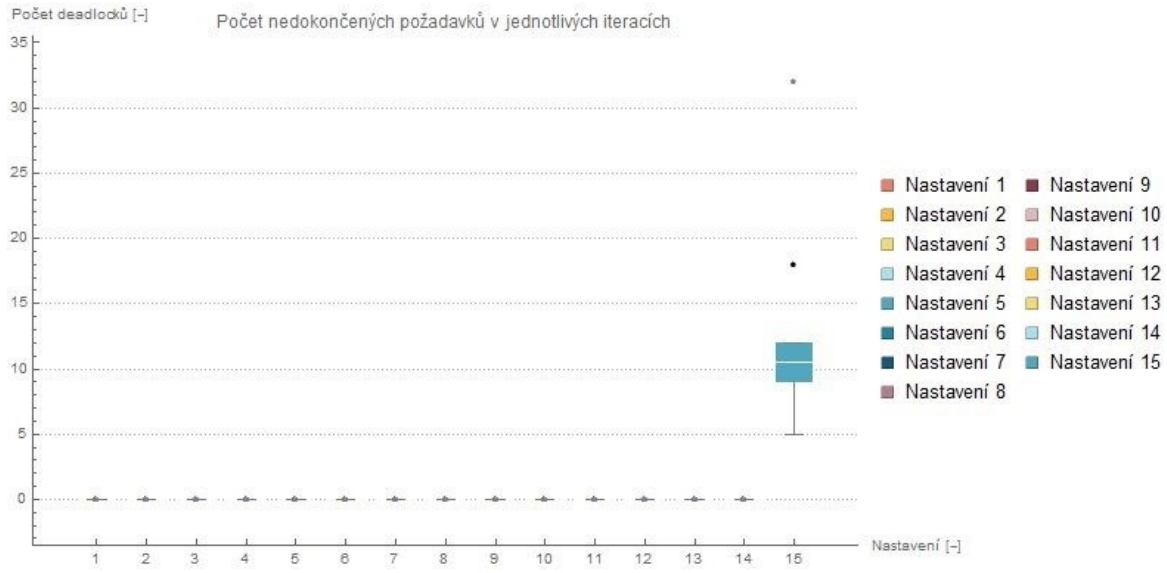
Podle grafu vývoje průměrného počtu nedokončených požadavků (*Obr. 21*) jde vidět, že všechny databáze jsou na tom obdobně a vykazují v průměru nula nedokončených požadavků. Až od čtrnáctého nastavení dojde ke vzniku nedokončených požadavků vlivem velkého počtu dat a tím prodloužení času nutného k dokončení požadavku.

8.2 Výsledky testu pro izolační úroveň READ COMMITTED

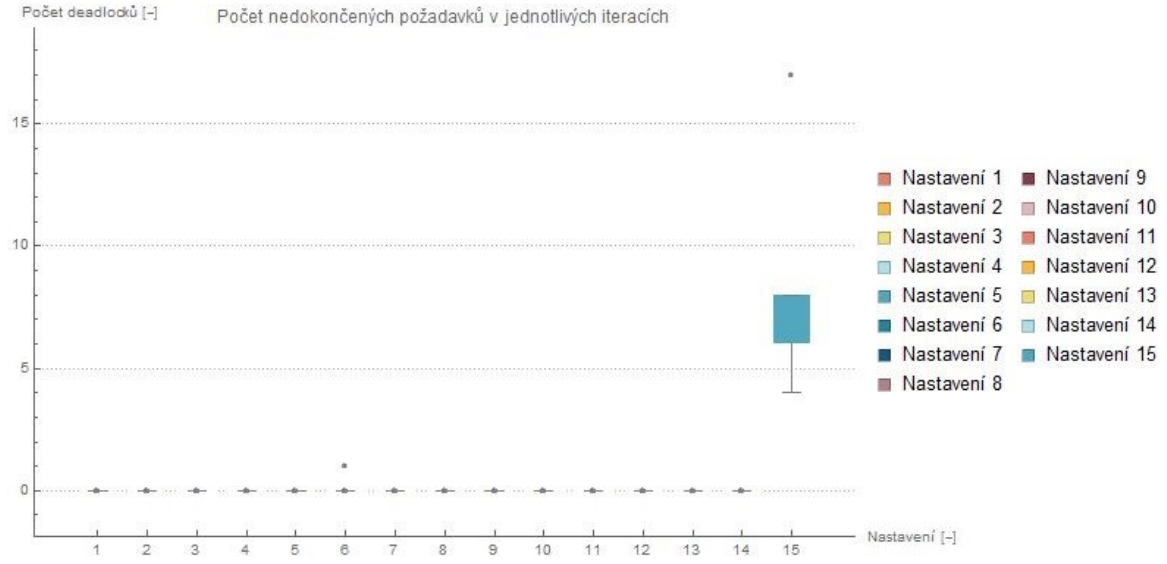


Obr. 22. Krabicový graf výskytu deadlocků pro izolační úroveň READ COMMITTED v závislosti na počtu záznamů pro databázi s neclusterovanými indexy

U izolační úrovně READ UNCOMMITTED se ve výsledcích testu neprojevila její vlastnost, že čtenář žádá o S zámek pro právě čtený záznam a tím se blokují aktualizační požadavky s čtecími požadavky.

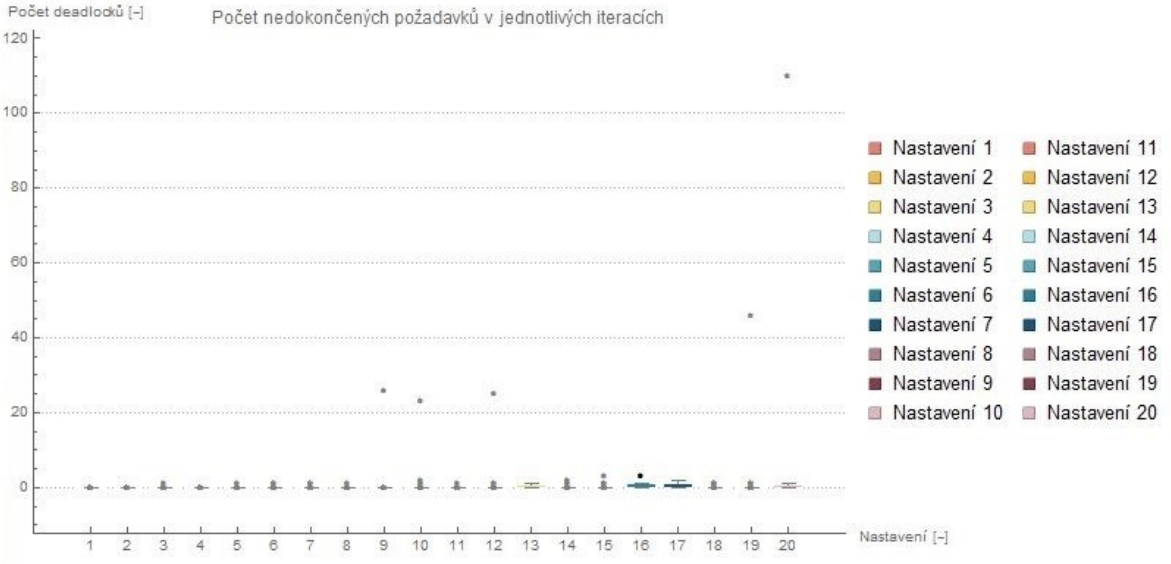


Obr. 23. Krabicový graf výskytu deadlocků pro izolační úroveň READ COMMITTED v závislosti na počtu záznamů pro databázi s clusterovanými indexy

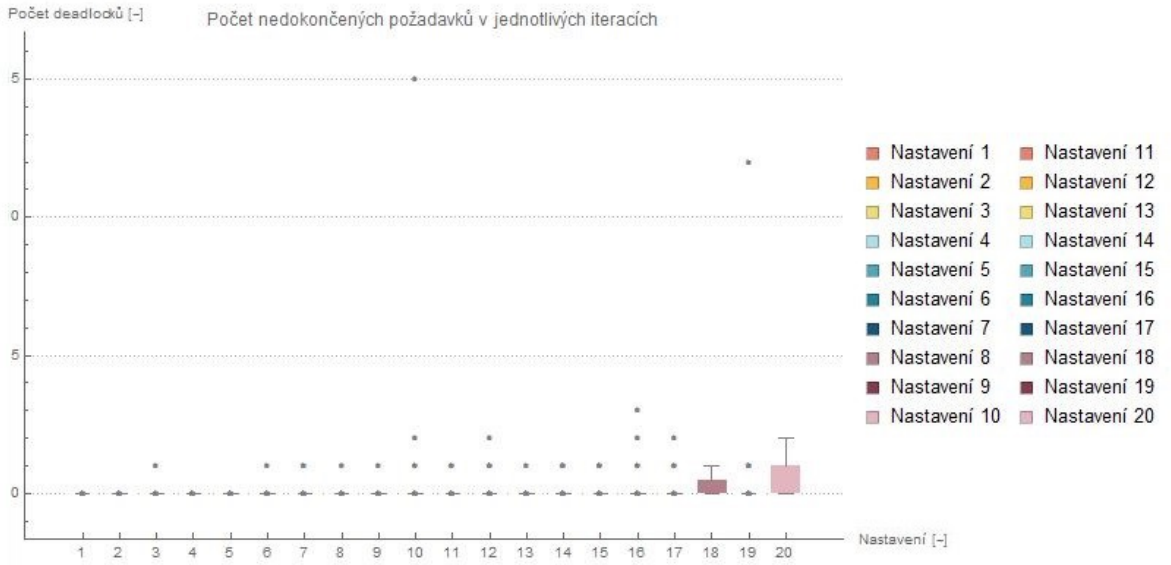


Obr. 24. Krabicový graf výskytu deadlocků pro izolační úroveň READ COMMITTED v závislosti na počtu záznamů pro databázi s optimalizovanými procedurami

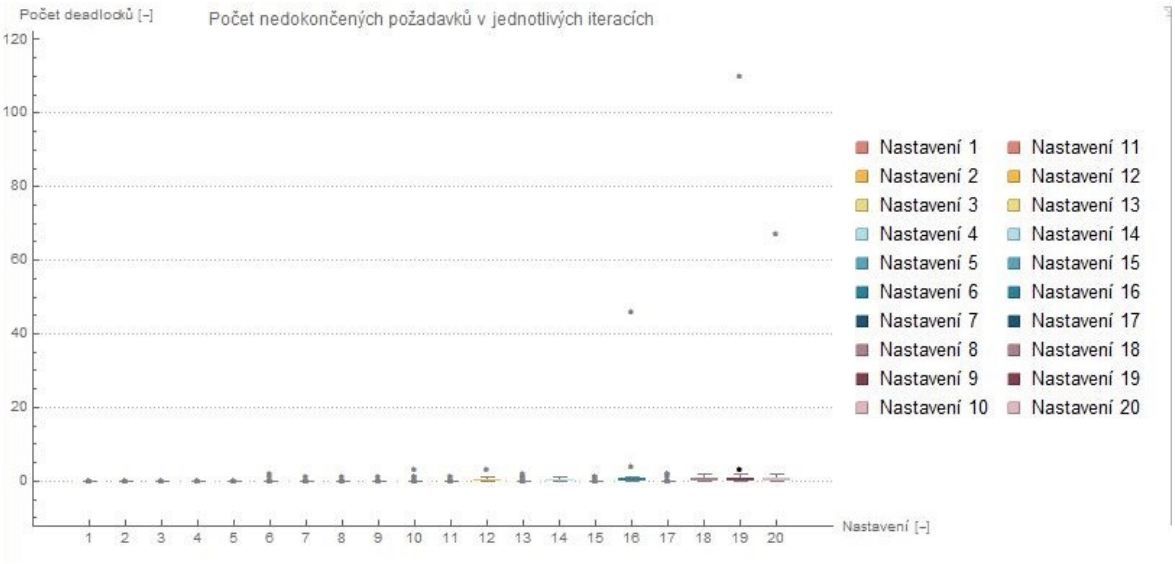
Z krabicových grafů (*Obr. 22., Obr. 23., Obr. 24.*) lze vidět nulový výskyt nedokončených požadavků napříč nastaveními. Až v případě patnáctého nastavení se objevují nedokončené požadavky, které jsou ukončeny z důvodu vypršení času pro provedení požadavku. Takže požadavek nekončí v deadlocku, ale je zrušen kvůli velké prodlevě ve spuštění, nebo získání výsledku.



Obr. 25. Krabicový graf výskytu deadlocků pro izolační úroveň READ COMMITTED v závislosti na počtu požadavků na databázi s neclusterovanými indexy



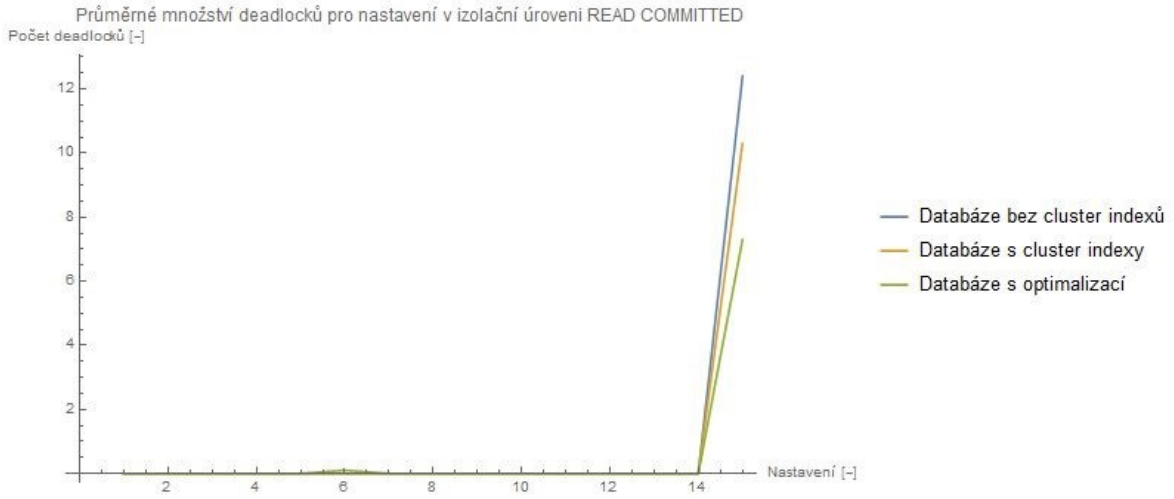
Obr. 26. Krabicový graf výskytu deadlocků pro izolační úroveň READ COMMITTED v závislosti na počtu požadavků na databázi s clusterovanými indexy



Obr. 27. Krabicový graf výskytu deadlocků pro izolační úroveň READ COMMITTED v závislosti na počtu požadavků na databázi s optimalizovanými procedurami

Z krabicových grafů (*Obr. 25., Obr. 26., Obr. 27.*) lze vidět spíše nepravidelný výskyt malého počtu nedokončených požadavků napříč nastaveními. V posledním nastavení jde vidět, že v jednom opakování nebylo dokončeno kolem sto deseti požadavků, opět kvůli překročení doby pro vykonání požadavku.

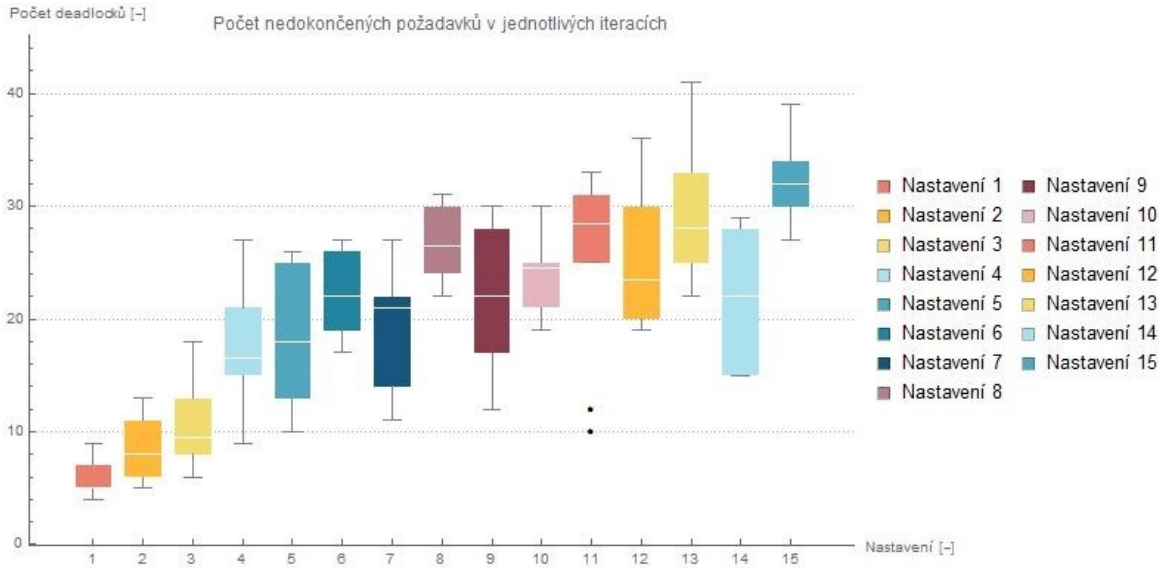
Důvodem velké prodlevy požadavků v READ COMMITTED vrstvě může být, že při probíhajícím aktualizačním požadavku a za ním probíhajícím čtecím požadavku dochází právě k chybnému chování, popsaném v kapitole o této izolační vrstvě. Jedná se o vícenásobné čtení, kde aktualizační požadavek neustále přidává pozměněné záznamy pro čtecí požadavek a ten čte daleko větší množství dat než by měl. Tím se stává požadavek značně delším a může překročit dobu vytyčenou pro jeho dokončení.



Obr. 28. Srovnání vývoje průměrného počtu nedokončených požadavků pro izolační úroveň READ COMMITTED pro všechny testovací databáze

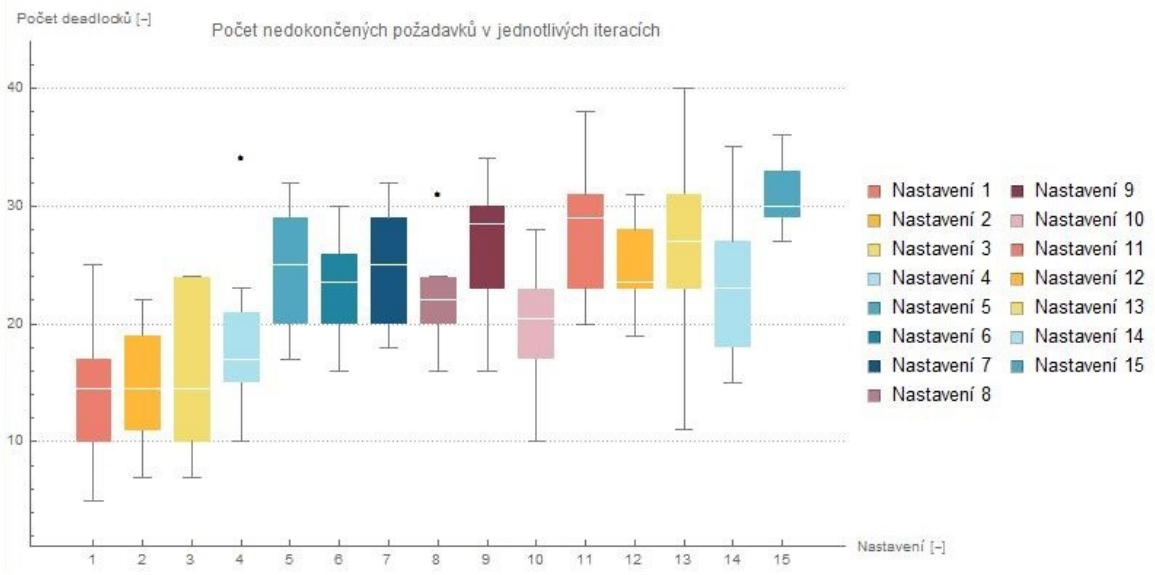
Podle grafu vývoje průměrného počtu nedokončených požadavků (*Obr. 28*) jde vidět, že všechny databáze jsou na tom obdobně a vykazují v průměru nula nedokončených požadavků. Až od čtrnáctého nastavení dojde ke vzniku nedokončených požadavků vlivem velkého počtu dat a tím prodloužení času nutného k dokončení požadavku. Stejně jako u předešlé izolační úrovně READ UNCOMMITTED.

8.3 Výsledky testu pro izolační úroveň REPEATABLE READ

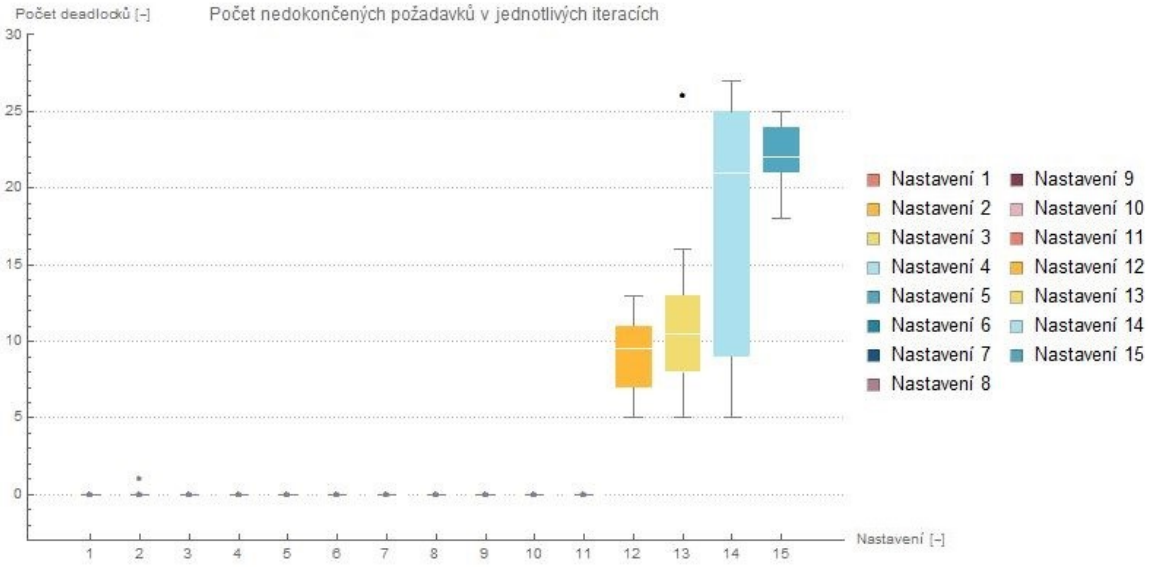


Obr. 29. Krabicový graf výskytu deadlocků pro izolační úroveň REPEATABLE READ v závislosti na počtu záznamů pro databázi s neclusterovanými indexy

V testech izolační úrovně REPEATABLE READ se projevila vlastnost, že transakce si ponechává získané S zámky po dobu celé transakce, nikoliv jen po dobu čtení záznamu.



Obr. 30. Krabicový graf výskytu deadlocků pro izolační úroveň REPEATABLE READ v závislosti na počtu záznamů pro databázi s clusterovanými indexy

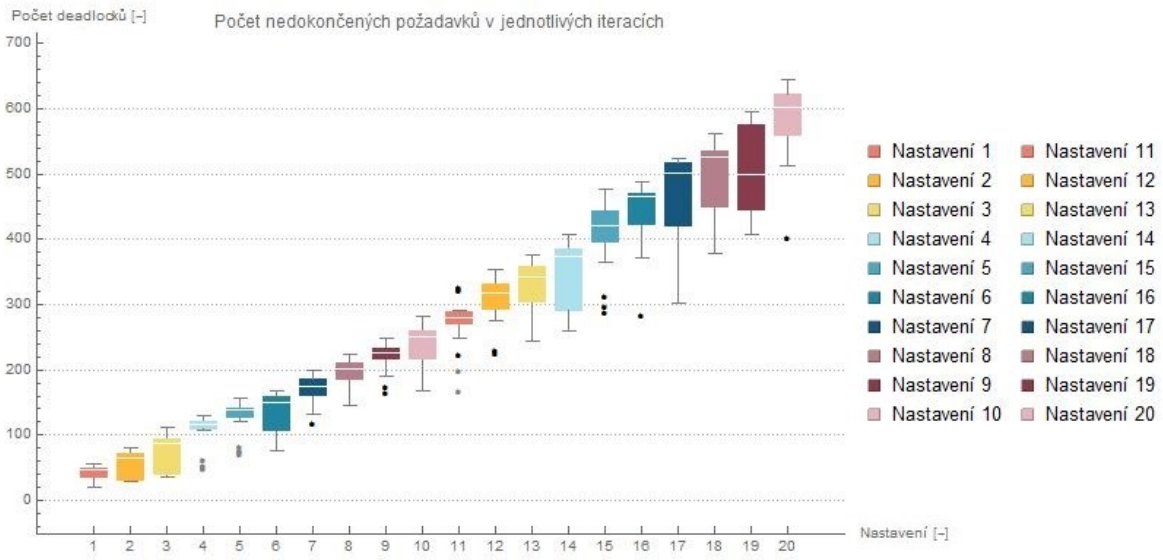


Obr. 31. Krabicový graf výskytu deadlocků pro izolační úroveň REPEATABLE READ v závislosti na počtu záznamů pro databázi s optimalizovanými procedurami

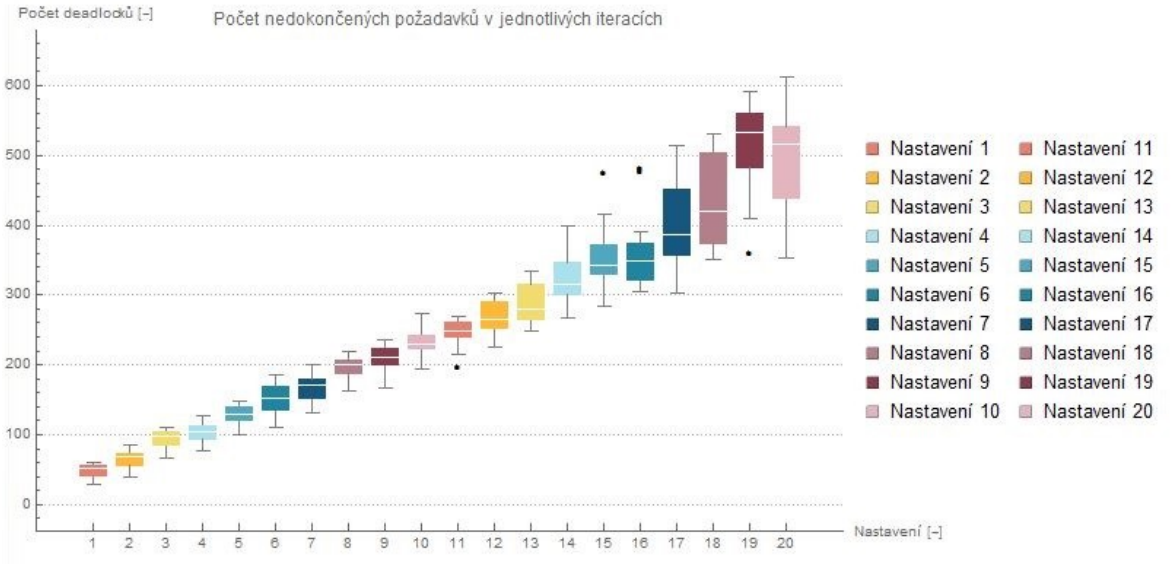
Z krabicových grafů (*Obr. 29., Obr. 30.*) lze vidět postupně se zvyšující výskyt nedokončených požadavků se zvyšujícím se počtem záznamů v databázi. Z grafu lze vidět, že deadlocky vznikaly už při prvním nastavení, které v databázi nastavuje záznamy v řádu stovek záznamů celkově. Dále se počet deadlocků zvyšuje, ale přes zvětšující se počet záznamů do řádu až deseti tisíců, se počet deadlocků zvětšuje v řádech jednotek. Při porovnání počtu záznamů v nastavení prvním s nastavením posledním, kde se jedná až tisíci násobek záznamů, dosahující až stovek tišíc záznamů v databázi, je nárůst deadlocků z průměrných pěti pro první nastavení na medián třiceti tří deadlocků pro patnácté nastavení.

Jinými slovy tendence růstu deadlocků není shodná s nárůstem dat, ale naopak s přibývajícími daty se počet deadlocků příliš nemění a ustaluje se na rozmezí třiceti až pětatřiceti nedokončených požadavků.

Z krabicového grafu (*Obr. 31.*) lze vidět, že odstraněním čtecí operace z transakce, která poté aktualizuje data, odstranilo víceméně veškeré deadlocky v průběhu testu. Důvodem je právě vlastnost izolační úrovně REPEATABLE READ, kde je transakci přiřazen S zámek po celou dobu jejího trvání. Tím pádem pokud dostala jedna transakce S zámek na záznamy v tabulce a poté byla spuštěna nad stejnou tabulkou aktualizační transakce, která nejdříve četla data, bylo jí znemožněno tyto data modifikovat.



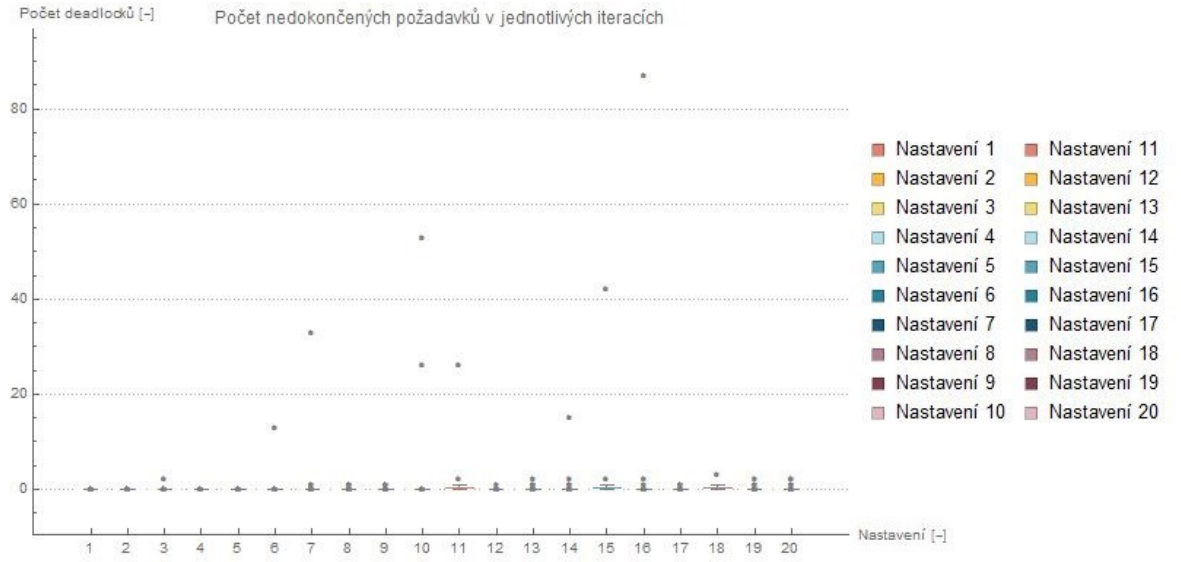
Obr. 32. Krabicový graf výskytu deadlocků pro izolační úroveň REPEATABLE READ v závislosti na počtu požadavků na databázi s neclusterozavanými indexy



Obr. 33. Krabicový graf výskytu deadlocků pro izolační úroveň REPEATABLE READ v závislosti na počtu požadavků na databázi s clusterozavanými indexy

Z krabicových grafů (*Obr. 32., Obr. 33.*) lze vidět počet nedokončených požadavků téměř lineárně rostoucí s počtem požadavků na databázi. Jelikož z čtrnácti požadavků v jednotlivém cyklu každý požadavek obsahuje čtecí operaci na nějakou tabulku, kde jeden požadavek čte všechny záznamy ze všech tabulek a tím si získá S zámek na celou databázi. Pokud je získán S zámek na všechny záznamy v databázi, je znemožněno aktualizačním operacím modifikovat záznamy a většinou jsou zvoleny jako oběti deadlocku.

Důkazem lineárního růstu je stálý poměr dvou třetin nedokončených požadavků z celkového počtu požadavků. Pro první nastavení je medián nedokončených požadavků čtyřicet požadavků z celkových sedmdesáti požadavků na databázi a pro poslední nastavení medián šest set nedokončených požadavků z celkových devíti set deseti požadavků na databázi.



Obr. 34. Krabicový graf výskytu deadlocků pro izolační úroveň REPEATABLE READ v závislosti na počtu požadavků na databázi s optimalizovanými procedurami

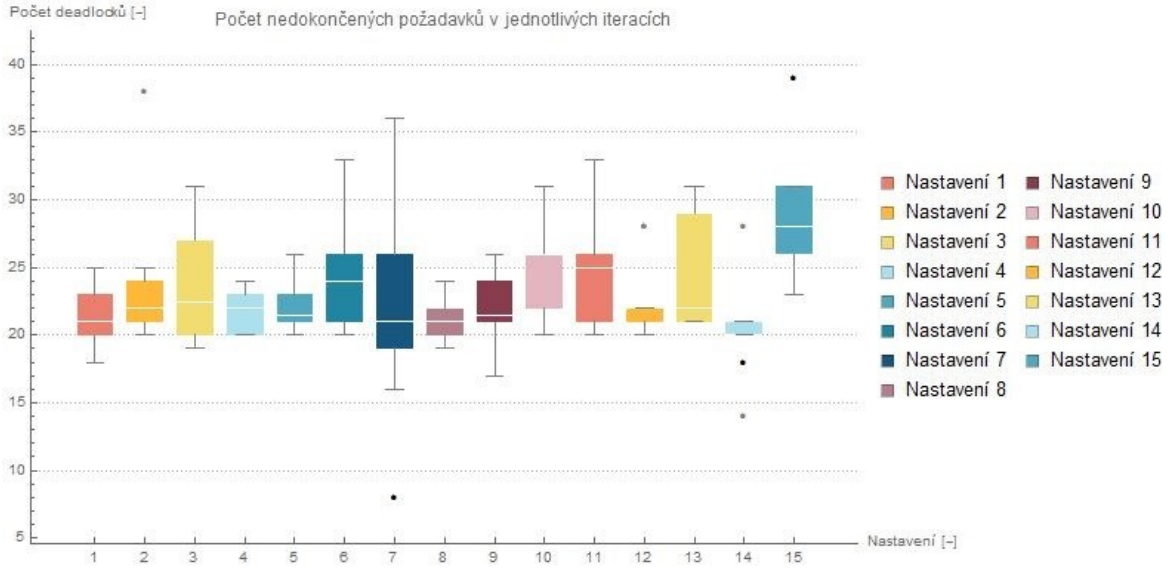
Z krabicového grafu (*Obr. 34.*) lze vidět, že odstraněním čtecí operace z transakce, která poté aktualizuje data, odstranilo víceméně veškeré deadlocky v průběhu testu. Důvodem je právě vlastnost izolační úrovně REPEATABLE READ, kde je transakci přiřazen S zámek po celou dobu jejího trvání. Tím pádem pokud dostala jedna transakce S zámek na záznamy v tabulce a poté byla spuštěna nad stejnou tabulkou aktualizační transakce, která nejdříve četla data, bylo jí znemožněno tyto data modifikovat.

Obr. 35. Srovnání vývoje průměrného počtu nedokončených požadavků pro izolační úroveň REPEATABLE READ pro všechny testovací databáze

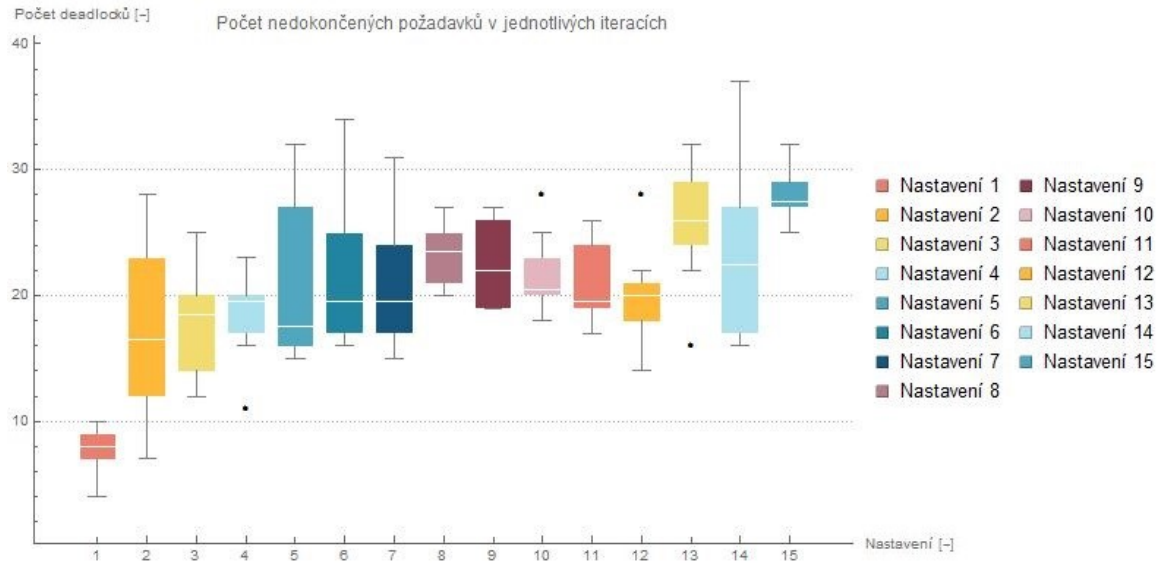
Z grafu vývoje průměrných počtů nedokončených požadavků (*Obr. 35*) je vidět rozdíl mezi využitím clusterovaných a neclusterovaných indexů, kde ze začátku databáze s clusterovanými indexy má dokonce více deadlocků do desátého nastavení a poté dosahuje stejného nebo lepšího průměrného počtu nedokončených požadavků jako databáze s neclusterovanými indexy. Jak je jasné už z předešlých grafů, tak databáze s optimalizovanými procedurami do jedenácté iterace průměrně nevznikají žádné nedokončené požadavky, pak ovšem tento počet rapidně stoupá, ale stále dosahuje nižšího průměrného počtu nedokončených požadavků než ostatní databáze.

8.4 Výsledky testu pro izolační úroveň SERIALIZABLE

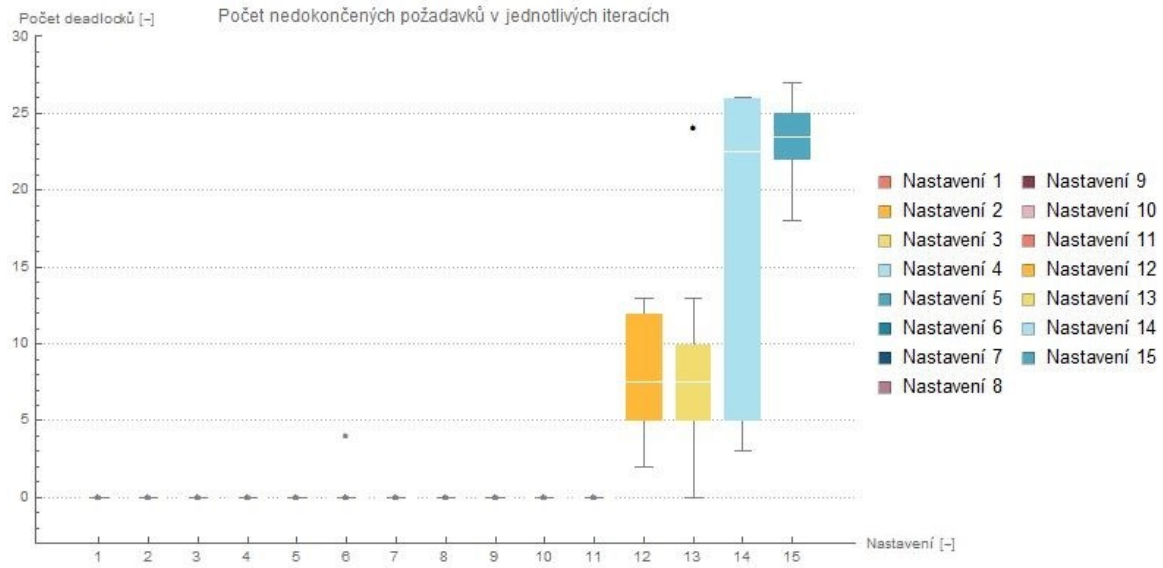
V testech izolační úrovně SERIALIZABLE se projevila vlastnost, že transakce si ponechává získané S zámky po dobu celé transakce, nikoliv jen pod dobu čtení záznamu. Také se projevilo uzamykání rozsahu čtených dat.



Obr. 36. Krabicový graf výskytu deadlocků pro izolační úroveň SERIALIZABLE v závislosti na počtu záznamů pro databázi s neclusterovanými indexy



Obr. 37. Krabicový graf výskytu deadlocků pro izolační úroveň SERIALIZABLE v závislosti na počtu záznamů pro databázi s clusterovanými indexy

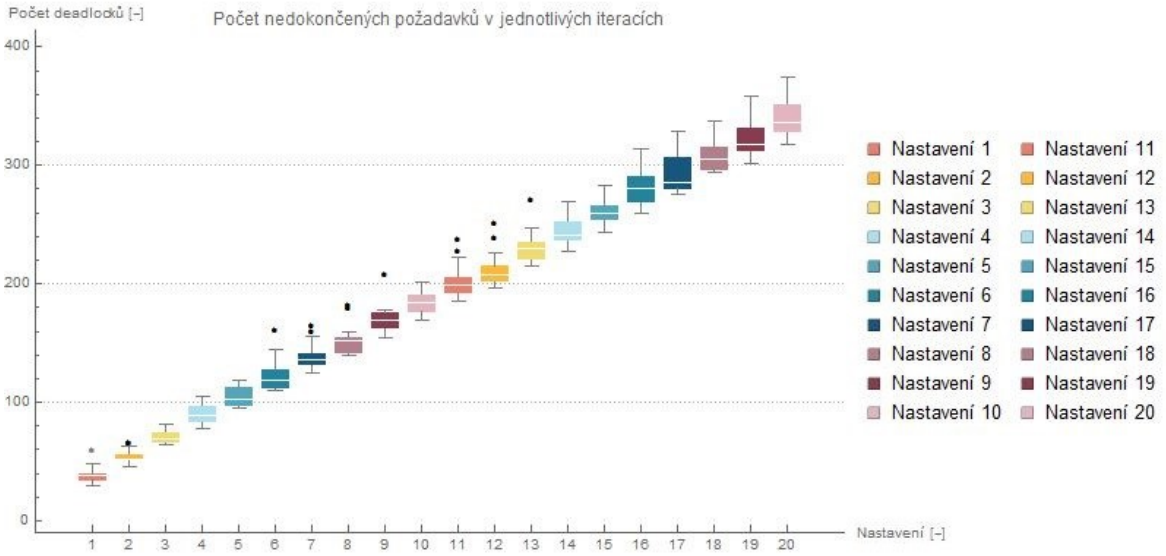


Obr 38. Krabicový graf výskytu deadlocků pro izolační úroveň SERIALIZABLE v závislosti na počtu záznamů pro databázi s optimalizovanými procedurami

Z krabicového grafu (*Obr. 36.*) lze vidět víceméně stálý medián počtu nedokončených požadavků na databázi. U krabicového grafu (*Obr. 37.*) lze vidět pomalý nárůst mediánu počtu nedokončených požadavků na databázi s rostoucím počtem záznamů v databázi. Důvodem je držení S zámků do konce transakce, stejně jako v případě izolační úrovně REPEATABLE

READ, kde se projevilo stejné chování jako důvod vzniku nedokončených požadavků. V izolační úrovni SERIALIZABLE se přidává ještě zamykání rozsahu indexů, které by mohly odpovídat požadavku. To způsobuje zamčení dalších záznamů, které by mohly být použity pro požadavek. Takže například při dotazu, který spojuje všechny tabulky ve struktuře na tabulku Order, aby zjistil všechny informace o jednotlivých objednávkách, tak je zamčena pro tuto transakci celá databáze a ostatní transakce mohou číst data, ale žádné data nesmí měnit. Proto jsou většinou jako oběť deadlocku vybrány aktualizační požadavky, protože svým IU zámkem blokují více požadavků, které se snaží o přečtení dat.

Krabicový graf výskytu deadlocků pro databázi s optimalizovanými procedurami (*Obr. 38.*) ukazuje opět, jak optimalizace procedury odstraňuje výskyt nedokončených operací. Ačkoliv i v tomto případě se ve dvanáctém nastavení a dále začnou objevovat nedokončené operace ve stejném měřítku jako u ostatních databází. Kde se jedná převážně o deadlocky vzniklé chováním izolační vrstvy SERIALIZABLE.

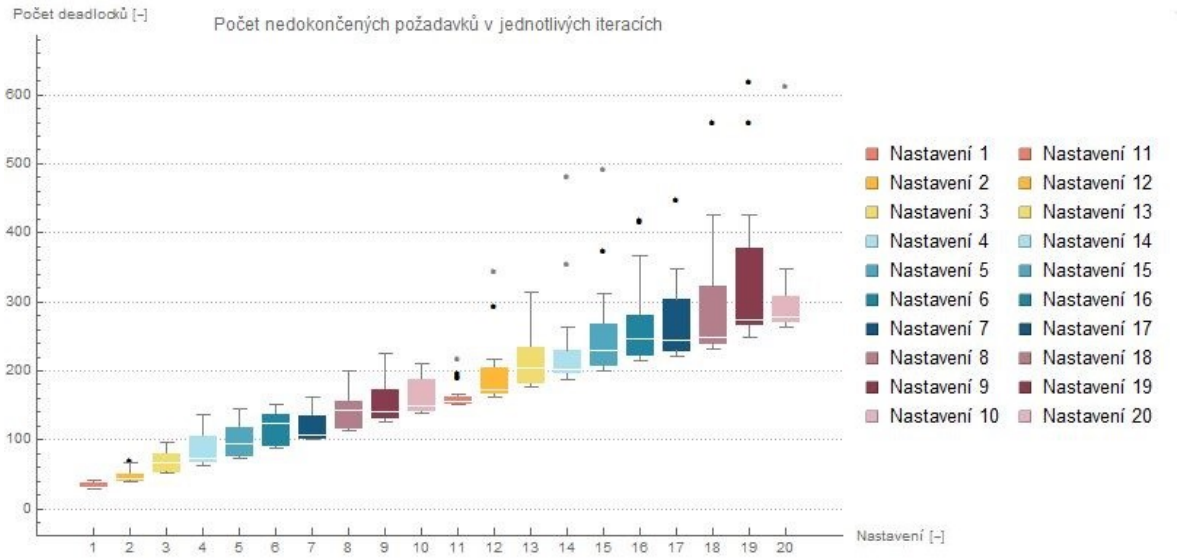


Obr. 39. Krabicový graf výskytu deadlocků pro izolační úroveň SERIALIZABLE v závislosti na počtu požadavků na databázi s neclusterovanými indexy

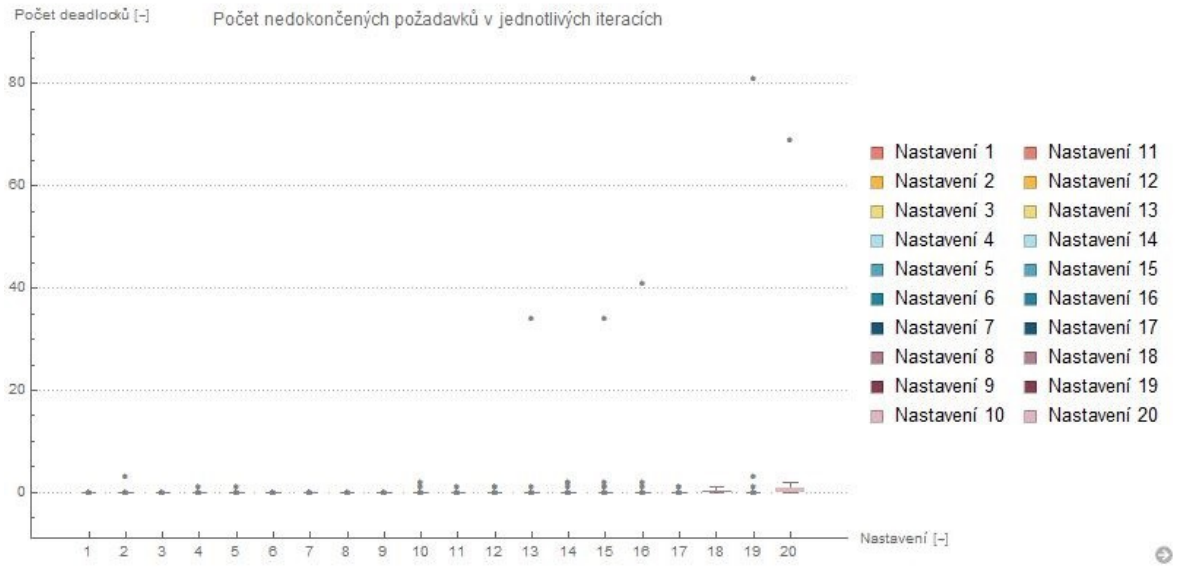
Z krabicového grafu (*Obr. 22.*) lze vidět lineární nárůst nedokončených požadavků s rostoucím celkovým počtem požadavků na databázi.

Jelikož pro první nastavení celkových sedmdesáti požadavků je čtyřicet zvoleno jako oběť deadlocku a ukončeno. Kdežto pro nastavení dvacáté, kde celkový počet požadavků je devět set deset požadavků a z toho je kolem tří set čtyřicet požadavků nedokončeno.

Důvodem menšího nárůstu nedokončených požadavků pro izolační vrstvu SERIALIZABLE oproti vrstvě REPEATABLE READ je serializace požadavků a tím delší čas nutný pro provedení požadavků.



Obr. 40. Krabicový graf výskytu deadlocků pro izolační úroveň SERIALIZABLE v závislosti na počtu požadavků na databázi s clusterovanými indexy



Obr. 41. Krabicový graf výskytu deadlocků pro izolační úroveň SERIALIZABLE v závislosti na počtu požadavků na databázi s optimalizovanými procedurami

Obr. 42. Srovnání vývoje průměrného počtu nedokončených požadavků pro izolační úroveň SERIALIZABLE pro všechny testovací databáze

V grafu vývoje průměrného počtu nedokončených požadavků (*Obr. 42.*) jde vidět, jak průměrný počet vzniklých deadlocků je víceméně stálý mezi jedna dvaceti a dvaceti čtyřmi deadlocky, pro databázi s neclusterovanými indexy. Pro databázi s clusterovanými indexy má průměrný počet indexů pomalu rostoucí tendenci. Pro databázi s optimalizovanými procedurami je situace obdobná jako u izolační úrovně REPEATABLE READ. Průměrná hodnota vzniklých deadlocků je pro prvních deset nastavení nulu, ale pro větší počet záznamů v databázi začne počet vzniklých deadlocků strmě stoupat.

ZÁVĚR

Cílem práce bylo analyzovat možnosti návrhu databáze uzpůsobené paralelnímu přístupu a možnosti kontroly chování takové databáze, popis možnosti detekce a řešení deadlocků vzniklých v databázi, jaké jsou dostupné metody pro předcházení a odstranění deadlocků v databázi. Na základě zjištěných poznatků poté navrhnout testovací databázi a webovou aplikaci zátěžového testu.

V první části práce jsem se zabýval analýzou používaných návrhových modelů databáze. V další části jsem se zaměřil na druhy prostředků návrhového modelu pesimistické souběžnosti pro řízení přístupu požadavků k datům a jejich možné typy a přechody mezi nimi.

Dále byly popsány transakce a význam jednotlivých ACID vlastností. Ve čtvrté části byly popsány takzvané zabranitelné fenomény čtení a také izolační úrovně dostupné v rámci Microsft SQL Server Database Engine a jejich povolené chování spolu s příklady možných rizik při jejich využití, například více násobné čtení pro izolační vrstvu READ COMMITTED a podobně.

V následující části byla provedena analýza typů deadlocků vyskytujících se v databázi a byly rozděleny do několika kategorií a popsány podmínky pro vznik jednotlivých typů deadlocků. Také byly analyzovány možnosti detekce deadlocků v databázi a poznačeno u příznaků trasování 1204, 1222 a SQL Profileru, že jejich podpora bude ukončena a v dalších vydáních SQL Serveru se již nemá využívat jejich implementace a Microsoftem je doporučeno přejít na Extended Events technologii. Také v této části bylo popsáno řešení deadlocků a jakým způsobem je možné delegovat chybový vztah do aplikace.

V kapitole o testovací databázi byla popsána struktura navržené testovací databáze a jaké typy uložených procedur má implementovány. Pro větší jednoduchost testování byly vytvořeny tři rozdílné databáze, kde první databáze byla navržena jako nepřizpůsobena pro paralelní přístup a další databáze jako implementující jednu z metod pro přizpůsobení paralelnímu přístupu.

Druhá databáze využívala clusterovaných indexů, které by měly urychlit hledání v databázi a tím zamezit vzniku několika deadlocků, protože transakce by nepotřebovaly dlouhý časový interval k dokončení. Třetí databáze měla nastaveny clusterované indexy a zároveň také byly optimalizovány procedury způsobující nejčastěji konflikt a tím riziko častého zvolení jako oběti deadlocku.

Pro další možnosti předcházení deadlocku nebyly provedeny testy ani vytvořeny databáze, jelikož optimalizace procedur byla označena jako nejčastější řešení deadlocků. V případě této práce byly procedury optimalizovány odstraněním SELECT části z procedury, která nejdříve přečetla data v databázi a následně byly data aktualizována.

Řešení deadlocku pomocí určení priority při volení oběti deadlocku bylo vynecháno taktéž, jelikož výsledkem je stále deadlock, akorát se jedná o upřednostnění důležitější operace před jinou.

Řešení vzniklého deadlocku jeho zachycením v SQL databázi a adekvátním zpracováním této informace bylo vynecháno ze stejného důvodu jako určení priority deadlocku. Vzniklý deadlock se řeší například opakovaným spuštěním požadované procedury, ale jedná se pouze o dočasné řešení a pro cíl této práce by takové řešení bylo spíše protichůdné.

Řešení deadlocku delegováním chybového stavu do aplikace, která komunikuje se samotnou databází, bylo využito v samotné aplikaci zátěžového testu.

V kapitole o vytvořené aplikaci zátěžového testu byly popsány využité technologie při vývoji a také jakým způsobem aplikace zpracovala požadované vstupní parametry testu od uživatele, následně provedla test na základě těchto parametrů a prezentovala získané výsledky uživateli.

V další kapitole bylo provedeno testování tří testovacích databází a vyhodnocení získaných výsledků testů. Z těchto testů bylo možné vyvodit několik faktů. Testy potvrzovaly dopad definovaného chování jednotlivých izolačních úrovní na míru vzniklých deadlocků v databázi.

V testech bylo jasně vidět nekonfliktnost a velmi nízká míra vzniku deadlocků v rámci izolačních úrovní READ UNCOMMITTED a READ COMMITTED, ale za ceny ztráty konzistence dat a tím další práce s nevalidními daty. Další izolační úrovně REPEATABLE READ a SERIALIZABLE zajišťovaly naopak konzistenci dat a jejich validitu, ale za cenu velké míry vzniku deadlocku.

Při porovnání testovacích databází a míry vzniku deadlocků pro jednotlivé izolační úrovně v závislosti na aplikované metodě předcházení deadlocků bylo vidět malý vliv clusterovaných indexů a to pro velký počet záznamů v databázi, čítající tisíce záznamů. Naopak zcela zásadní vliv měly procedury volané aplikací, kde databáze s optimalizovanými procedurami nevykazuje ani při izolační úrovni SERIALIZABLE vznik deadlocků do počtu záznamů v řádech tisíců.

SEZNAM POUŽITÉ LITERATURY

[1] DAVIDSON, Louis. *Exam Ref 70-762 Developing SQL Databases*. USA: Pearson Education, 2017. ISBN 978-1509304912.

[2] Parallel Programming in the .NET Framework. *Microsoft Developer Network* [online]. [cit. 2017-01-30]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx.

[3] SQL Server Technical Documentation. *Microsoft TechNet* [online]. [cit. 2017-01-30]. Dostupné z: https://technet.microsoft.com/en-us/library/ms130214.aspx.

[4] BEN-GAN, Itzik. *T-SQL Fundamentals*. 3rd. USA: Microsoft Press, 2016. ISBN 978-1509302000.

[5] SQL Server Deadlocks by Example. *Simple Talk* [online]. 2014 [cit. 2017-03-20]. Dostupné z: https://www.simple-talk.com/sql/performance/sql-server-deadlocks-by-example/

[6] Handling Deadlocks in SQL Server. *Simple Talk* [online]. 2012 [cit. 2017-03-20]. Dostupné z: https://www.simple-talk.com/sql/database-administration/handling-deadlocks-in-sql-server/

[7] DELANEY, Kalen. *SQL Server Concurrency: Locking, Blocking adn Row Versoning*. 1st. USA: Simple Talk Publishing, 2012. ISBN 978-1-906434-90-8.

[8] Lost update, uncommitted data, dirty read problem in transaction processing in SQL. *More Process* [online]. 2013 [cit. 2017-04-10]. Dostupné z: http://www.moreprocess.com/database/sql/lost-update-uncommitted-data-dirty-read-problem-in-transaction-processing-in-sql

[9] DB Concurrency Control with .NET – Overview. *Peter Meinl: Software Development Tips* [online]. 2011 [cit. 2017-04-10]. Dostupné z: https://petermeinl.wordpress.com/2011/03/05/db-concurrency-control-with-net/

[10] The Read Committed Isolation Level. *SQL Performance.com* [online]. 2014 [cit. 2017-04-10]. Dostupné z: https://sqlperformance.com/2014/04/t-sql-queries/the-read-committed-isolation-level

[11] The Serializable Isolation Level. *PerformanceSQL.com* [online]. 2014 [cit. 2017-04-11]. Dostupné z: https://sqlperformance.com/2014/04/t-sql-queries/the-serializable-isolation-level

[12] The Repeatable Read Isolation Level. *PerformanceSQL.com* [online]. 2014 [cit. 2017-04-11]. Dostupné z: https://sqlperformance.com/2014/04/t-sql-queries/the-repeatable-read-isolation-level

[13] Read Committed Snapshot Isolation. *PerformanceSQL.com* [online]. 2014 [cit. 2017-04-11]. Dostupné z: https://sqlperformance.com/2014/05/t-sql-queries/read-committed-snapshot-isolation

[14] The SNAPSHOT Isolation Level. *PerformanceSQL.com* [online]. 2014 [cit. 2017-04-11]. Dostupné z: https://sqlperformance.com/2014/06/sql-performance/the-snapshot-isolation-level

[15] Inconsistent data, non repeatable read, phantom insert problem in transaction processing in SQL. *More Process* [online]. 2013 [cit. 2017-04-11]. Dostupné z: http://www.moreprocess.com/database/sql/inconsistent-data-non-repeatable-read-phantom-insert-problem-in-transaction-processing-in-sql

[16] Transaction isolation levels. *Slideshare* [online]. 2012 [cit. 2017-04-11]. Dostupné z: https://www.slideshare.net/ErnestoHernandezRodriguez/transaction-isolation-levels

[17] DBCC TRACEON - Trace Flags (Transact-SQL). *Microsoft - Docs* [online]. 2016 [cit. 2017-04-11]. Dostupné z: https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-traceon-trace-flags-transact-sql

[18] SQL Server Profiler. *Microsoft - Docs* [online]. 2016 [cit. 2017-04-11]. Dostupné z: https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ACID	Vlastnosti transakcí (Atomicity Consistency Isolationd Durability).
ASP	Součást .NET frameworku pro tvorbu webových stránek.
AWE	Rozšíření pro funkce managementu paměti SQL Serveru 32 bit.
ID	Primární klíč tabulky.
IIS	Internet Information Service
SQL	Strukturovaný dotazovací jazyk (Structured Query Language).
T-SQL	Transakční SQL (Transact - SQL).
Zámek IS	Záměr sdíleného zámku.
Zámek IU	Záměr aktualizačního zámku.
Zámek IX	Záměr exkluzivního zámku.
Zámek I-N	Zámek rozsahu Indexu a NULL zámek zdroje.
Zámek S	Sdílený zámek.
Zámek SIU	Sdílený zámek záměr aktualizační.
Zámek SIX	Sdílený zámek záměr exkluzivní.
Zámek S-S	Sdílený zámek rozsahu a sdílený zámek zdroje.
Zámek S-U	Sdílený zámek rozsahu a aktualizační zámek zdroje.
Zámek U	Aktualizační zámek.
Zámek UIX	Aktualizační zámek záměr exkluzivní.
Zámek X	Exkluzivní zámek.
Zámek X-X	Exkluzivní zámek rozsahu a exkluzivní zámek zdroje.
.NET	Framework pro jazyk C#.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: POUŽITÝ SOFTWARE

Microsoft Visual Studio 2017 Community	Vývojové prostředí
SQL Server 2016	Databázový systém
SQL Server Management Studio v17.0	Prostředí pro správu databáze
IIS 10.0 Express	Webový server