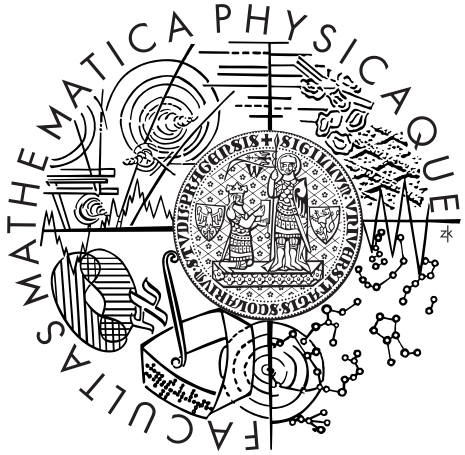
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Radek Hušek

Hybridní databáze

Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, Ph.D.

Studijní program: Informatika

Studijní obor: obecná informatika

Praha 2012

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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

V Praze dne ..... Podpis autora

Název práce: Hybridní databáze

Autor: Radek Hušek

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, Ph.D., Katedra aplikované matematiky

Abstrakt: Tato práce popisuje návrh a implementaci datové struktury, která se snaží kombinovat výhody databází a běžných datových struktur. Ze světa databází vychází především podpora pro persistenci dat prostřednictvím jejich uložení na disku a práce s daty pomocí transakcí, které umožňují paralelní přístup při zajištění konzistence dat. Od datových struktur naopak přichází implementace v podobě knihovny funkcí a snaha o maximální jednoduchost a uložení dat v paměti. Navržená databáze staví na konceptu transakční paměti a data jsou na disku ukládána ve formě záznamu provedených operací.

Klíčová slova: in-memory databáze, transakční paměť, datové struktury

Title: Hybrid Databases

Author: Radek Hušek

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš Ph.D., Department of Applied Mathematics

Abstract: This thesis presents design and implementation of a data structure, which tries to combine advantages of both databases and regular data structures. Main advantages of databases we try to retain are data persistence through storing data on a hard disk and working with data using transactions which allows us parallel access without danger of inconsistency. From data structures we borrow the implementation as a library of functions and the aim on simplicity and storing data in memory. Our implementation is built around the concept of (software) transactional memory; all data are stored on hard drive as log of operations.

Keywords: in-memory database, transactional memory, data structure

Obsah

Úvod

Tato práce se snaží vyplnit mezeru, která se rozevírá mezi (relačními) databázovými systémy na straně jedné a klasickými datovými strukturami na straně druhé. Dnešní databáze se zaměřují na zpracování obrovských množství dat, a proto užívají velmi sofistikované metody manipulace s nimi. Tyto pokročilé postupy mají však i svou stinnou stránku – nemalou časovou i prostorovou režii. Ta se u obrovských souborů dat neprojevuje, neboť je bohatě kompenzována vyšší efektivitou zpracování, ale u malých dat bývá významná. Oproti tomu datové struktury jsou velmi rychlé a mají flexibilnější rozhraní, které lze často přizpůsobit konkrétní aplikaci, ale postrádají možnost perzistentně ukládat data (tedy tak aby přežila restart či pád aplikace).

V důsledku zvětšování velikosti operační paměti počítačů stále častěji nastává situace, kdy se potřebná data vejdou do operační paměti. Pokud je u dat požadováno trvalé uložení mezi běhy aplikace a jsou tak cenná, že je nutné zajistit možnost jejich obnovy i v případě havárie systému, nezbývá programátorovi, než místo rychlé datové struktury použít databázi. Použití databáze zajistí potřebnou perzistenci a konzistentnost dat, ale oproti použití datové struktury negativně ovlivní výkon aplikace.

Z toho důvodu jsem se rozhodl vytvořit hybridní databázi, která se snaží kombinovat výhody databází a datových struktur. Udržuje všechna data v paměti, ve tvaru co nejpřátelštějším k programovacímu jazyku dané aplikace. Zajišťuje uložení jejich konzistentní verze na disku a umožňuje pracovat s nimi pomocí transakcí. Podpora transakcí je důležitá jak z důvodu určení konzistentní verze dat pro uložení na disku, tak především s ohledem na stále narůstající počet (logických) procesorů v dnešních počítačích, neboť elegantně eliminuje potřebu složité ruční synchronizace přístupu k datům, která je velmi častým zdrojem těžko odhalitelných chyb.

Součástí práce je i ukázková implementace navržené databáze v jazyce C ve formě knihovny funkcí. Jazyk C byl zvolen pro svou rozšířenost, a to především v unixovém světě. Výhodou jazyka C oproti C++ je jeho rozšířenost a „kompatibilita“ – zatímco užívat z C++ knihovny vytvořené v C je velmi snadné, je použití knihoven vytvořených v C++, především obsahuje-li jejich rozhraní šablony, v podstatě nemožné (přičemž podobná situace nastává s vazbami do jiných programovacích jazyků).

Rozčlenění práce

- První kapitola obsahuje motivaci k vytvoření hybridní databáze a hrubý popis jejího fungování.
- Druhá kapitola se zabývá softwarovou transakční pamětí a její implementací použitou v této práci.
- Třetí kapitola popisuje formát uložení databáze na disku.
- Čtvrtá kapitola obsahuje popis rozhraní navržené databáze.

- V páté kapitole jsou rozebrány implementačně závislé aspekty hybridní databáze.
- Příloha 1 obsahuje příklad jednoduchého programu, který využívá hybridní databázi.
- V příloze 2 naleznete výsledky testů škálovatelnosti.

1. Popis hybridní databáze

V této kapitole stručně popíšeme z jakých předpokladů jsme při návrhu hybridní databáze vycházeli, jak se promítly do požadavků na ni kladených a jak se nám tyto povedlo naplnit. Závěr kapitoly je věnován popisu základního fungovaní navržené struktury, který je hlouběji rozveden v kapitolách následujících.

1.1 Motivace

Jak bylo zmíněno v úvodu, hlavním motivem této práce je nabídnout alternativu programům, které potřebují perzistentně uchovávat data, a přitom nepotřebují komplexní databázový systém. Pro začátek uveďme, jaké klíčové vlastnosti od vytvářené databáze chceme:

- Implementovat databázi jako knihovnu funkcí, takže komunikace s databází znamená přímé volání těchto funkcí bez dalších mezivrstev.
- Ukládat data v paměti, takže čtení je vždy rychlé, a přitom paralelně udržovat obraz dat na disku, aby v případě havárie programu bylo možné obnovit data v konzistentní verzi.
- Vyhledávání dat místo dotazovacího jazyka zajišťují uživatelem vytvořené indexy. Ty mají s indexy v relačních databázích společné, že obsah obou je jednoznačně určen obsahem databáze. Na rozdíl od nich je jejich rozhraní mnohem flexibilnější – uživatel komunikuje s indexem přímo a rozhraní indexu určuje jeho tvůrce, není pevně zadrátováno v implementaci databáze.
- Podpora více vláken a udržování konzistence dat za pomoci transakcí.
- Skládání operací do větších celků pomocí zanořování transakcí.

1.1.1 Knihovna funkcí vs. samostatný proces

Většina databázových systémů, a to i nerelačních, se drží koncepce samostatného databázového serveru, s nímž klient komunikuje (typicky protokolem TCP/IP). To je výhodné pro velké systémy, neboť je možné server samostatně ovládat a přistupovat k němu z více aplikací. Navíc získáme síťovou transparentnost a větší spolehlivost, protože pád aplikace neublíží serveru a aplikace také nevidí paměť databáze, takže ji nemohou poškodit.

Pro naše potřeby, kdy počítáme s aplikací menších rozměrů, je síťová transparentnost nadbytečná a samostatné ovládání serveru spolu s přístupem k němu z více aplikací také není předpokládaným použitím. (Ačkoliv s úpravami ohledně alokace paměti je teoreticky možné navrhovanou databázi provozovat jako samostatný proces, který komunikuje s ostatními přes sdílenou paměť.)

Naopak díky integraci do klientského procesu získáváme rychlou komunikaci s databází spočívající v přímém volání jejích metod. A jako identifikátory jednotlivých objektů uložených v databázi můžeme používat přímo ukazatele, což opět výrazně snižuje režii.

1.1.2 Primární úložiště v paměti, záloha na disku

Jelikož cílíme na menší objemy dat, je výhodné implementovat hybridní databázi, aby pracovala s daty přímo v paměti, a jejich kopii na disku užívat jen jako zálohu, ze které budou obnovena při příštím startu. Tím sice zvýšíme paměťové nároky (dat nicméně není tolik, aby to představovalo výrazný problém), ale zároveň nemusíme mít diskový formát, ve kterém lze rychle vyhledávat. Můžeme tedy upustit od formátů, které respektují strukturu dat, a databázi zapisovat na disk jako log – tedy posloupnost operací, které vedly k jejímu současnému stavu. (Log samozřejmě nemusí obsahovat všechny události od vytvoření databáze. Běžný postup je, že pokud velikost logu přeroste rozumnou mez, je vygenerován nový, který obsahuje nejkratší posloupnost operací vedoucí k současnému stavu.)

To výrazně snižuje jak složitost zápisu, tak i počet I/O operací nutných k jeho provedení (protože nezapisujeme na přeskáčku, ale vždy jen na konec souboru), což se významně projeví především u rotačních pevných disků (ačkoli i pro disky SSD je tento přístup výhodnější, neboť jednotlivé zápisy jsou výrazně větší). Navíc při použití datového formátu respektujícího strukturu dat by stejně bylo nutné udržovat logu nebo jiným způsobem zajistit atomicitu prováděných operací.

1.1.3 Organizace dat v paměti

Data v paměti budeme chceme organizovat podobně jako datové struktury. Základní jednotkou dat je **uzel**. Každý uzel se skládá z **atributů**, které obsahují vlastní data, přičemž jak atributy jaké atributy obsahuje je určeno jeho typem. Při porovnání s relační databází typy uzlů odpovídají tabulkám, uzly řádkům tabulek a atributy jednotlivým buňkám. Z hlediska objektového programování, které je bližší skutečné implementaci, jsou typy uzlů třídami, uzly instancemi těchto tříd a atributy jednotlivými instančními proměnnými.

Objektovému pohledu také odpovídá, že každý uzel má svou jednoznačnou identitu – dokážeme rozlišit dva uzly, i když mají stejný obsah. Za běhu je vhodné jako identifikátor uzlu používat ukazatel na něj.

Atributy nemohou obsahovat libovolné hodnoty, ale každý má pevně určený typ. Typy atributů pokrývají základní typy jazyka C – jsou mezi nimi tedy celá čísla různých rozsahů s či bez znaménka a čísla s plovoucí řádovou čárkou – a obsahují několik užitečných typů navíc – řetězec schopný pojmout libovolnou posloupnost bytů (který se ale chová jako jednoduchý hodnotový typ, ne jako pole) a ukazatel na jiný uzel.

1.1.4 Uživatelsky definované indexy

Na rozdíl od relačních databází, kde indexy slouží „pouze“ k urychlení dotazů, v navrhované hybridní databázi hrají, především kvůli absenci dotazovacího jazyka, klíčovou roli. Indexem rozumíme uživatelem definovanou datovou strukturu jejíž stav (myšlen je samozřejmě stav z pohledu vnějšího světa, ne zakódování tohoto stavu uvnitř indexu) je jednoznačně určen stavem databáze bez ohledu na posloupnost operací, které k jejímu současnému stavu vedly.

Z pohledu databáze je index určen callback funkcí, kterou databáze volá při vytvoření, smazaní či změně uzlu, a kontextu, který je předán callback funkci. Uživatelské rozhraní indexu určuje jeho tvůrce s tím, že by uživatelské funkce neměly index modifikovat (protože jeho stav má být určen pouze stavem databáze). U každého indexu je při jeho definici možno určit, zda bude sledovat všechny uzly v databázi, nebo bude vázán pouze na určitý typ uzlu. Tento způsob svázání indexů s databází byl zvolen, protože představuje rozumný kompromis.

Pokud by všechny indexy byly globální a uživatel by filtroval zajímavé události až v callback funkci, mohlo by to znamenat nemalou režii, protože většina volání by končila zjištěním, že daný index událost nezajímá. Na druhou stranu možnost mít globální index je nezbytná, protože není možné ho přímočaře nahradit více menšími indexy. Jedinou zbývající možností je nechat uživatele při registraci indexu určit, k jakým uzlům je vázán.

Zde se nabízí mnoho variant, jak takovou volbu provést. Například vytvořit dotazovací jazyk, který by uživateli dovolil filtrovat události podle typu uzlu a jeho obsahu. Tím by bylo možné vyhnout se zbytečným voláním callback funkcí, ovšem režie takto komplexního systému by nejspíše převážila nad získanou úsporou. Proto jsem zvolil co nejjednodušší metodu, která by zároveň měla vyhovovat většině nasazení – tedy že indexy jsou registrované jako globální, nebo jsou vázány na jeden typ uzlu.

1.1.5 Podpora transakcí

Podpora transakcí je klíčová ze dvou důvodů. Za prvé databáze musí mít přehled, kdy jsou data v ní obsažená v konzistentním stavu, aby je jako taková označila při zápisu na disk. Kdyby tak nečinila, bylo by obtížné po pádu aplikace zjistit, zda jsou data poškozena, a nemožné obnovit korektní stav dat bez složité spolupráce uživatele databáze. Druhým a neméně významným důvodem pro používaní transakcí je velké zjednodušení práce s databází z více vláken současně, jelikož odpadá komplikovaná ruční synchronizace.

1.2 Abstraktní popis rozhraní databáze

Jak bylo řečeno v úvodu, hybridní databáze bude implementována jako knihovna funkcí v jazyce C. Protože definovat typy uzlů, index a databází v jazyce C by bylo velmi nepohodlné, bude k tomuto účelu vytvořen samostatný jazyk, který bude překládán do jazyka C. Popis jazyka definic i skutečného rozhraní databáze naleznete ve čtvrté kapitole. Zde pouze zmíním základní principy fungování databáze.

1.2.1 Transakce

Transakce budeme implementovat pomocí konceptu softwarové transakční paměti (STM). Každé vlákno, které bude chtít s databází pracovat, si nejprve vytvoří **handle** – objekt skrze, který bude probíhat další práce. Kdybychom

nezavedli handle, musela by každá transakce existovat jako objekt. To by znamenalo, že každá transakce na počátku musela provést alokaci tohoto objektu a na svém konci jeho uvolnění. To je možné implementovat celkem efektivně, ale databáze navíc potřebuje vědět o bežících transakcích. Proto by se tyto musely vždy nějak registrovat, a tedy modifikovat hlavní strukturu popisující databázi, což není žádoucí.

Pokud máme handly, tak každá transakce existuje uvnitř příslušného handlu. Ten je lokální pro dané vlákno, takže není problém s přístupem k němu. Aby měla databáze přehled o probíhajících transakcích, musejí se i handly registrovat. V případě handlů to ale nevadí, protože na rozdíl od transakcí existují dlouho a vznikají málokdy. (Typický scénář je, že každé vlákno si vytvoří jeden handle, který bude používat po celou dobu svého běhu.) Nad handlem jsou definovány všechny ostatní funkce pro práci s databází. Mezi základní operace patří začátek a konec transakce, zrušení probíhající transakce a funkce pro práci s uzly (tedy vytvoření a smazání uzlu, přečtení hodnoty atributu a zápis hodnoty atributu).

1.2.2 Indexy

Indexy byli již popsány výše v této kapitole, takže pouze pár doplnění. Jaké indexy databáze obsahuje je vždy definováno už před jejím vytvořením a za běhu indexy není možné přidávat ani odebírat. Naopak v diskovém formátu se přítomnost indexů nijak neprojeví – je tedy možné používat tutéž databázi v různých aplikacích s různými indexy.

Všem indexům je společné, že z pohledu databáze jsou tvořeny kontextem a callback funkcí. Kontexty indexů jsou součástí transakční paměti. To je nezbytné, aby stav indexů zůstal konzistentní se stavem databáze. Kvůli tomu je i s indexy nutné pracovat výhradně pomocí transakcí. O alokaci kontextů se databáze stará sama a uživateli nejsou přímo přístupné. S tím souvisí, že z pohledu uživatele jsou metody indexy obaleny makrem, které se stará o předání správného kontextu.

2. Softwarová transakční paměť

2.1 STM obecně

Softwarová transakční paměť je koncept pro řízení přístupu ke sdíleným datům. Oproti řešením založeným na zámcích je to koncept optimistický – tedy zatímco se zámky vlákno vždy nejprve musí získat potřebné zdroje (zámky) a až poté provést svou činnost, při použití STM vlákno začne optimisticky vykonávat, co potřebuje, prostředky získává implicitně za běhu, když je potřebuje, a případný konflikt s jiným vláknem se řeší, až k němu dojde. Typicky se konflikt řeší tím, že jedno z vláken odstraní provedené změny, čímž uvolní prostředky, takže druhé může pokračovat, a zrušené vlákno se zkusí provést později.

2.1.1 Fungování transakcí

Základním kamenem transakční paměti jsou transakce. Transakce je posloupnost operací se sdílenými daty, která se vnějšímu pozorovateli jeví atomicky. Tedy každé vlákno kromě toho, které danou transakci provádí, vidí sdílená data tak, že transakce buď nebyla provedena vůbec, nebo byla provedena celá, ale nikdy nepozoruje žádné mezistavy. Z vnějšího pohledu transakce převádějí data z jednoho konzistentního stavu do jiného konzistentního stavu.

Protože jsou transakce vykonávány optimisticky a prostředky pro svůj běh získávají, až když je potřebují, může se stát, že požadované prostředky již vlastní jiná transakce. Takováto situace se nazývá **kolize**. Řešit kolizi čekáním, než jsou potřebné zdroje uvolněny, není obecně možné, protože není globálně dáno žádné pořadí, v němž jsou zdroje získávány. Kvůli může snadno nastat deadlock (tedy situace, kdy graf vztahu „čekám až transakce X uvolní zdroje, které potřebuji“ obsahuje orientovaný cyklus). Standardním řešením je v případě kolize zrušit změny provedené jednou z transakcí – provést tzv. **rollback** – čímž se uvolní prostředky a druhá transakce může pokračovat.

Aby bylo možné transakci zrušit, musí provádět pouze operace, které je možné vrátit zpět (typicky může zapisovat do paměti, ale nesmí provádět vstup a výstup či komunikaci s jinými procesy). Všechny všechny operace vykonané v průběhu transakce jsou zaznamenávány do tzv. **logu**. Rozlišujeme dva základní typy logů – zpětný a dopředný. **Zpětný (undo) log** zaznamenává původní hodnoty dat, která transakce přepisuje. **Dopředný (redo) log** naopak funguje tak, že požadavky na zápis nejsou provedeny ihned, ale jsou uloženy do logu a provedeny až na konci transakce, když už víme, že kolize nenastala.

Vraťme se k restartování transakcí. Pokud budeme – tak jako v této práci – vždy rušit transakci, která chce získat nedostupné zdroje, vystavujeme se při vysokém zatížení systému riziku livelocku (situaci, kdy se transakce vzájemně ruší, aby získaly zdroje, ale žádná není dokončena, protože dříve, než se stihne dokončit je zrušena). Livelocku se dá vzdorovat zavedením nějaké varianty priorit transakcí, kdy několikrát restartovaná transakce získá vyšší prioritu, a při kolizi je zrušena transakce s nižší prioritou, nebo snižováním limitu současně běžících

transakcí, pokud nastává příliš mnoho kolizí. Za variantu snižování počtu současně běžících transakcí lze považovat i exponenciálně dlouhé čekání před restartem transakce.

V závislosti na implementaci může transakce detekovat kolizi v okamžiku, kdy k ní dojde, nebo až při pokusu o úspěšné ukončení transakce, tzv. **commitu**. Kolize při commitu nejčastěji nastává v případě, kdy zjistíme, že data čtená transakcí (tzv. **readset**) byla změněna jinou transakcí.

2.1.2 Historie a budoucnost transakční paměti

Poprvé byla transakční paměť představena v článku [1], kde byla předvedena hardwarová transakční paměť – počítala tedy s podporou ze strany hardware. Jako taková se příliš neujala, ale v roce 1995 byl i na základě jejích myšlenek publikován článek [2], kde byla představena softwarová varianta transakční paměti, která nevyžadovala (explicitní) hardwarovou podporu. Na tuto práci posléze navázalo mnoho dalších.

Za pozornost také stojí snaha integrovat podporu transakční paměti do překladače GCC [3] a hardwarová implementace transakční paměti, která bude obsažena v procesorech generace Haswell firmy Intel [4]. Integrace do překladače a s tím spojené rozšíření jazyka o atomické konstrukce umožní snadnější vývoj a hardwarová podpora může přinést výrazné vylepšení výkonu. Jelikož ale implementace transakční paměti v GCC je zatím ve velmi raném stádiu a procesory Haswell budou uvedeny na trh až v příštím roce, nebudu se jim zde dále věnovat.

2.2 STM TL2 engine a jeho rozšíření

V této práci jsem jako ideový základ použil rozšířenou verzi TL2 enginu z dizertační práce [5]. TL2 engine vznikl v roce 2006 [6]. V původní implementaci pracoval s redo logem, zamykal paměť po slovech a zámky tedy byly uloženy v globální tabulce. V roce 2007 v práci [5] byl tento engine rozšířen o podporu undo logu, zamykání objektů a umísťování zámků do objektů.

Kromě toho byla implementace výkonnostně optimalizována např. zjednodušením operace čtení, kde bylo nahrazeno dvojí kontrolování zámku (před a po načtení hodnoty) jednou kontrolou po načtení.

V následujících sekcích jsou probrány jednotlivé aspekty rozšířeného TL2 enginu spolu s popisem a zdůvodněním postupů zvolených v implementaci hybridní databáze.

2.2.1 Průběh transakce

Globální hodiny a verzované zámky

Základním prvkem STM stavící na TL2 enginu jsou verzované zámky spolu s globálními hodinami. **Verzovaný zámek** je neblokující zámek, který ve stavu odemčeno uchovává i informaci o době poslední změny a v uzamčeném stavu obsahuje ukazatel na transakci, která ho uzamkla. Neblokující zámek znamená, že jím lze chránit data proti paralelnímu přístupu, ale operace „zamkni“ nikdy neblokuje – odpovídá tedy operaci `trylock` běžných zámků. Typicky je implementován pomocí 8-bajtového slova, jehož nejnižší bit určuje stav zamčeno/odemčeno a zbylé bity podle situace ukazatel na vlastníka či verzi (používáme-li zarovnané ukazatele, nevadí že nemůžeme uložit jejich nejnižší bit, neboť je vždy nulový).

Globální hodiny jsou vlastně verzovaný zámek, který se vždy nachází v odemčeném stavu a zvětšuje svou hodnotu s ukončením (i neúspěšným) každé transakce, která zapisovala.

Transakce začíná načtením hodnoty globálních hodin. Při zápisu (skutečném zápisu, který v případě dopředného logu nastává až při commitu) je nejprve získán příslušný zámek (při tom se kontroluje, má-li hodnotu nejvýše rovnou začátku transakce), následně je hodnota zapsána. Zámky jsou uvolněny na konci transakce.

Každé čtení nejprve zkopíruje čtenou hodnotu do privátní paměti (tj. paměti přístupné jen danému vláknu) a poté zkontroluje, zda příslušný zámek je odemčený a jeho hodnota je menší nebo rovna času začátku transakce (nebo může být zámek zamčen, je-li vlastněn danou transakcí).

Teoreticky by stačilo kontrolovat readset až při commitu, čímž by se snížila režie. Vznikl by však problém, že transakce může vidět nekonzistentní data. U takovéto transakce by sice commit nikdy neuspěl, takže by konzistence databáze nebyla porušena, ale v kódu transakce by přestaly platit invarianty, které mají o datech platit. Pokud s tím kód nepočítá (a může být obtížné a pracné ošetřit všechny možné patologické případy), může se program např. dostat do nekonečné smyčky. Nebo jsou-li načtená data ukazatel, může nastat neoprávněný přístup do paměti, protože není zaručeno, kam ukazuje.

Kvůli výkonově kritickým částem je v implementaci hybridní databáze možnost kontrolu čtených dat obejít přímým přístupem k položkám uzlu. Po načtení dat je však nutné použité uzly explicitně ověřit, aby byly zařazeny do readsetu. Také je nezbytné ověřovat stav všech ukazatelů dříve než jsou dereferencovány.

Schéma průběhu transakce

Průběh transakce je následující:

1. Načtení globálních hodin.
2. Vlastní obsah transakce – čtení a zápisy dat.
3. V případě redo logu jsou zabrány všechny zámky potřebné k zápisu.

4. Je zkontrolován readset – tedy že všechny zámky, jimiž chráněná data byla čtena, mají hodnotu nejvýše počátku transakce či jsou aktuální transakcí vlastněny.
5. V případě redo logu jsou všechna data zapsána.
6. Globální hodiny jsou inkrementovány.
7. Jsou uvolněny zámky. Jejich nová hodnota odpovídá nové hodnotě globálních hodin.

V případě, že transakce pouze četla a čtená data průběžně validovala, je možné kroky 3 až 7 přeskočit. Transakce totiž nedrží žádné zámky, a nemusí tedy modifikovat globální hodiny. Kontrolu readsetu lze pominout, jelikož i kdyby případná validace skončila neúspěchem, nemůže tím být poškozena integrita databáze (protože ta nebyla transakcí měněna) a okolní svět nemůže poznat, v jakém pořadí transakce skutečně provedly commit.

2.2.2 Zamykání bloků vs. zamykání objektů

Důležitým rozhodnutím pro výkon STM je volba vhodného modelu zamykání paměti. Zamykat paměť lze po blocích – typicky přirozeně zarovnaných slovech či řádcích cache paměti dané architektury – či lze zamykat jednotlivé objekty (ve smyslu úseků paměti, které mají nějaký rozumný význam). Výhodou zamykaní po blocích je, že STM engine nemusí nic vědět o datech a kromě využívání funkcí STM ke čtení a zápisu dat není potřeba jiná spolupráce. Naopak problémem může být zápis do paměti přes hranice bloků či nevhodná granularita – zamykání po slovech může být příliš jemné, zatímco u zamykání po řádcích cache paměti hrozí, že tentýž zámek hlídá zcela nesouvisející data. Navíc zámky musejí být uloženy v globální tabulce a jsou blokům přiřazeny pomocí hashovací funkce – a zde opět hrozí riziko, že jeden zámek hlídá nesouvisející data.

Avšak u tabulky zámků určuje kolize hashovací funkce a přesné adresy dat v paměti, a tedy mají více náhodný charakter. Jejich pravděpodobnost lze také snížit zvětšením tabulky, což může mít negativní dopad na cache paměti (tabulka v nich zabírá více místa) a na rychlost transakcí pokud je např. readset ukládán jako bitmapa tabulky zámků.

Naopak zamykání objektů vyžaduje spolupráci se STM enginem, který musí být schopen každé čtené či zapisované adrese přiřadit objekt (třeba tak, že rozhraní bude „čti položku z objektu“ místo „čti z adresy“). Dále je možné zámky ukládat i přímo do objektů místo do globální tabulky, takže nemohou nastávat problémy, kdy jeden zámek chrání nesouvisející data. To samozřejmě zvýší paměťovou náročnost, jelikož zámků globální tabulce bývá podstatně méně než objektů.

Zvolená implementace

Z pohledu granularity je, po vzoru mnoha relačních databázových enginů, nejvhodnější zamykání po uzlech jakožto nejmenších samostatných objektech. Stejně jsou zamykány indexy, ačkoli u nich je až konkrétní implementací určeno, co je objekt (a tedy kolika zámky je celý index chráněn).

Původním cílem bylo implementovat zamykání s globální tabulkou, protože v běžných implementacích STM výkonem nezaostává za zámky v objektech a přitom je implementačně jednodušší a šetří paměť. Po provedení měření jsem implementoval i zamykání po objektech, jelikož implementace s globální tabulkou zámků špatně škálovala v závislosti na počtu vláken. Tato odchylka od běžných STM je způsobena především prací navíc při commitu – tedy zpracováním transakčního logu, aby ho bylo možné zapsat na disk.

V případě, že jsou transakce velmi malé (a řešení s globální tabulkou tedy v průběhu transakce může škálovat), je jich velký počet, takže čas strávený při commitu je významný (a nebo je zahlceno servisní vlákno, navíc vysoké množství commitů přispívá ke cachovému „ping pongu“ mezi transakčními vlákny a servisním vláknem). A jsou-li transakce větší, tak se pravděpodobnost kolize velmi rychle blíží jedné, takže se transakce vzájemně serializují. Použití globální tabulky větší než několik stovek zámků také není vhodné, protože příliš velká tabulka zabírá příliš místa v cache paměti, a tím výrazně zpomaluje program.

2.2.3 Zpětný vs. dopředný log

Jelikož transakce musí být v případě kolize schopny odčinit provedené změny, je nezbytné nějakým způsobem tyto změny zaznamenávat. K tomu existují dva přístupy – zpětný (undo) a dopředný (redo) log. Při použití zpětného logu transakce zapisuje do logu hodnoty sdílených dat, které změnila, a při kolizi projde log pozpátku a obnoví přepsaná data. Naopak dopředný log zaznamenává požadavky na zápis do sdílené paměti a skutečně je provede až při commitu, poté co bylo ověřeno, že žádná kolize nenastává.

Zpětný log

Výhodou zpětného logu je jeho jednoduchost, kdy k implementaci stačí prostý zásobník, a minimum práce navíc při commitu, kdy po jeho úspěšném provedení stačí jen uvolnit paměť zpětného logu. Průběh zápisu se zpětným logem je přímočarý. Nejprve získáme potřebný zámek, původní hodnotu přidáme na zásobník (spolu s informacemi o její adrese) a zapíšeme novou hodnotu. Nastane-li rollback transakce, projdeme zásobník od vrcholu a obnovíme hodnoty, poté odemkneme zámky.

Nevýhodou této implementace je, že velikost zásobníku je přímo úměrná počtu zápisů, ačkoli míst, kam bylo zapisováno, může být mnohem méně. Za jistých velmi specifických okolností (mezi zámky a objekty je bijekce, do logu ukládáme vždy celý objekt a nemáme vnořené transakce) lze prostorovou složitost této implementace snížit na lineární s počtem modifikovaných míst tím, že do logu

zapíšeme hodnotu, pouze pokud daný zámek opravdu zamkneme, a ne pokud ho již vlastníme.

Pokud potřebujeme optimální prostorovou složitost a se zásobníkovou implementací jich nelze dosáhnout, nezbývá než užít k ukládání logu datovou strukturu jako strom či hashovací tabulka, což zvýší režii, v případě stromu dokonce logaritmus-krát zpomalí každou transakci.

Poznámka: Protože uvolnění paměti obecně nelze vrátit, je vhodné paměť, která měla být uvolněna v průběhu transakce, zapisovat do logu a uvolnit ji až při úspěšném commitu. (Tedy z hlediska uvolňování paměti fungovat jako dopředný log.)

Dopředný log

Dopředný log je na implementaci složitější, ovšem oproti zpětnému logu minimalizuje dobu držení zámků, a tím usnadňuje paralelní běh transakcí. Větší složitost je způsobena nutností při každém čtení kontrolovat, zda čtená data nebyla modifikována a nejsou tedy uložena v logu. Kvůli tomu musí být modifikovaná data uložena v hashovací tabulce či podobné struktuře s možností rychlého vyhledávání a modifikace.

Výjimkou může být situace, kdy víme, že transakce sebou modifikovaná data nepotřebuje číst, či jí nevadí číst jejich starou verzi. (Což zjevně neplatí, chceme-li podporovat skládání transakcí.) V takovémto případě lze dopředný log implementovat pomocí fronty, jejíž každý prvek je požadavek na zápis. Další nevýhodou může být prodloužení commitu, neboť v jeho průběhu je nutné veškerá modifikovaná data skutečně zapsat do sdílené paměti.

Poznámka: Alokaci paměti není samozřejmě možné odsunout a musí být alokována již v průběhu transakce, takže se s ní musí pracovat pomocí zpětného logu.

Zvolená implementace

Transakce typicky více čtou než zapisují, takže optimalizace čtení má větší efekt než optimalizace zápisu. Proto jsem v ukázkové implementaci použil zpětný log implementovaný zásobníkem. Zpětný log vyžaduje při čtení kromě čtené hodnoty pouze kontrolu stavu příslušného zámku, takže jde o velmi rychlou operaci.

Použitá zásobníková implementace je vhodná za předpokladu, že jedna transakce nebude opakovaně přepisovat totéž místo. To je rozumný předpoklad, jelikož transakce by se celkově měly snažit minimalizovat přístupy do sdílené paměti a provádět výpočty v rychlejší privátní paměti.

3. Ukládání na disk

V této kapitole nejprve prozkoumám úpravy, které je nutné provést v implementaci transakční paměti, aby mohl být průběh každé transakce zaznamenán, takže po restartu aplikace může být průběh transakce „přehrán“. Nejprve popíši v jakých formátech je možné ukládat obsah databáze na disk, a poté jaký formát jsem při implementaci zvolil.

3.1 Rozšíření STM o zápis změn

Pro implementaci hybridní databáze je třeba STM rozšířit tak, aby při commitu transakce zajistila zapsání změn na disk. Tedy doplnit logování v průběhu transakce o informace, které umožní její opětovné provedení, a upravit commit, aby tyto informace zapsal na disk.

Zápis na disk může být pomalý a vlákno s transakcí nemusí chtít čekat na jeho dokončení. Navíc zapisovaná data je nutno v souboru seřadit (při použití globálních hodin nejlépe dle času commitu). Proto je vhodné zavést pomocné vlákno (dále servisní vlákno), které bude od ostatních vláken dostávat data, která mají být zapsána, a bude se starat o vlastní zápis. Data je nejlépe předávat pomocí globální fronty požadavků (která se postará o správné pořadí zápisu).

3.1.1 Ukládání dat v průběhu transakce

Možností, jak držet potřebná data v průběhu transakce, je několik. Nejpřímočařejší je zapisovat do bufferu transakci tak, jak má být uložena na disku. Výhodou je relativně snadná implementace (pokud je diskový formát vhodný), naopak nevýhodnou může být prodloužení doby transakce (každý zápis se dělá dvakrát). Další variantou je místo zapisování logu v nějakém vhodném interním formátu, který budu převeden do formátu pro zápis až při / po (úspěšném) commitu.

Toto řešení s předchozím sdílí problém vyššího množství zápisů do paměti, avšak při vhodném návrhu je výhodnější, protože zkracuje čas strávený uvnitř transakce. Naopak zvětšuje čas pro zápis logu na disk, ale to tolik nevadí, protože tato operace může probíhat již mimo transakční paměť. Významnou výhodou obou těchto přístupů je nezávislost na implementaci transakční paměti, takže mohou být použity i v případě, že implementace transakční paměti je pevně daná – což může být užitečné s nástupem hardwarových implementací transakční paměti.

Poslední variantou je integrace logu pro zápis na disk do standardního transakčního logu, který je již součástí implementace transakční paměti. Výhodou tohoto přístupu je snížení jak paměťové, tak časové režie, protože místo dvou logů obsahujících z velké části duplicitní informace (např. adresy, kam bylo zapisováno) máme log pouze jeden. Podmínkou pro použití tohoto přístupu je

samozřejmě možnost modifikovat implementaci transakční paměti. Protože implementace transakční paměti je součástí této práce, rozhodl jsem se pro tento přístup.

3.1.2 Zpracování dat při commitu

Pokud jsou potřebná data v průběhu transakce ukládána do bufferu přímo v diskovém formátu, stačí je v průběhu commitu zařadit do fronty ke zpracování servisním vláknem. Pokud jsou ovšem uložena v jiném tvaru, nabízí se otázka, zda je má do formátu pro zápis převést vlákno vykonávající příslušnou transakci, nebo až servisní vlákno. Otázka času zpracování se nabízí také pro uvolňování paměti (které vždy funguje jako dopředný log).

Výhodou odsunutí práce do servisního vlákna je zrychlení commitu. To může vést k vyššímu výkonu aplikace, pokud zápisů není tolik, aby servisní vlákno zahltily (výrazného zrychlení lze dosáhnout především ve chvíli, kdy zapisuje pouze jedno vlákno). Druhou výhodou může být snazší implementace, protože paměť uvolňuje pouze servisní vlákno (při vhodném řešení rollbacku), takže odpadají synchronizační problémy.

Naopak výhodou ponechání práce na jednotlivých vláknech je vyšší propustnost servisního vlákna. Problémem však může být komplikovanější synchronizace servisního vlákna, jelikož frontu nestačí implementovat jako spojový seznam (chráněný zámkem při přidávání položek vlákny transakcí) se semaforem, který určuje počet nezpracovaných položek v něm, ale je nutné nejprve transakci zařadit do fronty, poté zpracovat transakční log a nakonec oznámit servisnímu vláknu, že transakce je připravena k zápisu. (Složitější synchronizaci se lze vyhnout zpracováním logu pod ochranou zámku fronty, což je ale nevhodné, protože to výrazně prodlouží dobu držení zámku, nebo zpracováním na počátku commitu před přidáním do fronty, což zvyšuje riziko kolize s jinou transakcí.)

3.2 Zvolený diskový formát

V této sekci popíši možnosti, jak ukládat databázi na disk, a jaké řešení jsem zvolil pro hybridní databázi.

3.2.1 Datové formáty obecně

Většina databází (především relačních) používá souborový formát respektující strukturu dat – typicky soubory odpovídají tabulkám a každý soubor je vnitřně rozdělen na části stejné velikosti, které odpovídají jednotlivým řádkům (v praxi používané formáty jsou samozřejmě složitější kvůli podpoře pro data neomezené délky, přímo do datového souboru zakomponované indexy atp.). Výhodou je jednoduchá práce s daty, protože ta lze přímo načítat a zapisovat do souboru bez složitého zpracování v paměti. Naopak problémem je nízká rychlost zápisu, protože změny v různých řádcích databáze znamenají zápisy na různá místa různých souborů, což je u rotačních disků velmi pomalé a u ssd disků může

velmi snižovat životnost (protože disky SSD jsou schopny přepisovat data pouze po velkých blocích (řádově desítky až stovky kilobytů [7]) a každý blok vydrží jen omezené množství přepsání a i sebemenší změna v databázi přepíše celý blok).

Ještě významnějším problém je nemožnost zaručení konzistence dat, protože zápis změn zjevně není atomickou operací. Proto se většinou tento způsob uložení dat doplňuje ještě o transakční log, kam se zapisují operace, které se na databázi provádějí. S jeho pomocí je v případě havárie možné převést data zpět do posledního uloženého konzistentního stavu. Užití logu může mít i pozitivní dopad na výkon. Celková propustnost se sice sníží, protože je kromě změn přímo v datových souborech je navíc zapisován i log, ale transakci je při vhodné implementaci možné považovat za úspěšnou již po jejím zapsání logu, které je rychlejší než modifikace datových souborů, protože do logu se zapisuje sekvenčně, takže latence při commitu transakce může výrazně klesnout.

Datový formát založený na logu změn je naopak oblíben u noSQL in-memory databází (např. Redis [8]). Jeho výhodou je snadný zápis a rychlý zápis. Naopak problémem může být nutnost zpracovat celý log dříve, než můžeme číst data (což příliš nevadí, čteme-li log pouze při startu databáze). Závažnější je zvětšování logu v průběhu práce s databází, protože do logu se data pouze připisují a nikdy se neodebírají. Běžným řešením je při překročení určité velikosti logu vygenerovat log nový, který obsahuje nejkratší posloupnost operací vedoucí k současnému stavu databáze, a jím nahradit ten starý, příliš dlouhý. Aby tento přístup fungoval, nesmí být množství zápisů do databáze příliš velké.

3.2.2 Formát Hybridní databáze

Protože Hybridní databáze uchovává za běhu všechna data v paměti, zvolil jsem diskový formát založený na logu. Navržený diskový formát se skládá ze dvou vrstev. Spodní vrstva se stará o detaily uložení na disku a poskytuje vyšší vrstvě jednoduchou abstrakci pro strukturovaná data. Vyšší vrstva pak definuje prostředky té nižší, jak jsou uloženy jednotlivé uzly a operace s nimi.

Nižší vrstva

Cílem spodní vrstvy je poskytnout vyšší vrstvě abstrakci pro strukturovaná data, v jejímž rámci je pak vybudován vlastní formát databáze, při zachování maximální jednoduchosti a tím i rychlosti zpracování. Každý datový soubor se skládá z jednotlivých tzv. kořenových elementů. Kořenovým elementem může být pole (seznam) či řetězec. Řetězcem je zde myšlena konečná posloupnost bytů (tedy speciálně může obsahovat znak `'\0'`) a pole je konečná posloupnost jiných polí a řetězců.

Klíčovou odlišností kořenových elementů oproti elementům vnořeným je, že před každým z nich je uvedena celková délka daného elementu a na konci každého z nich může být CRC kód. Pro přehlednost uveďme gramatiku popisující nižší vrstvu, přičemž přesný způsob uložení jednotlivých částí bude popsán dále:

soubor := *kořenový_element*∗
kořenový_element := '[' "délka kořenového elementu" ']'
element ('#' "CRC32 Castagnoli")? '.'

element := *řetězec* | *pole*
pole := '(' *element*∗ ')'
řetězec := "délka řetězce" "obsah řetězce"

Délka kořenových elementů je užitečná spíše pro snadné načítání, kdy je celý kořenový element načten do bufferu v paměti, takže při čtení jednotlivých řetězců může být předáván přímo ukazatel do tohoto bufferu. CRC (Cyclic Redundancy Check) je druh kontrolního součtu, který umožňuje odhalovat chyby při přenosu či ukládání dat. Jeho přítomnost v kořenových elementech je důležitá pro implementaci transakcí. Každá transakce je uložena jako pole na kořenové úrovni, takže CRC kód na jejím konci lze použít k ověření, že byla korektně uložena. Z různých verzí CRC byla zvolena verze CRC32 Castagnoli [9], která je použita kupř. v protokolu iSCSI [10] či souborovém systému Btrfs [11]. Navíc je implementace této varianty CRC jednou z instrukcí SSE4.2 [12], takže v moderních procesorech může být výpočet CRC kódu výrazně urychlen. (Ač tato optimalizace není v současné implementaci obsažena.) Měření totiž ukazují, že výpočet CRC je, po práci s diskem samozřejmě, časově nejnáročnější částí práce s daty v tomto formátu.

Co se týče způsobu uložení jednotlivých prvků, tak délka kořenového elementu a CRC kód jsou vždy uloženy jako 8-bajtové resp. 4-bajtové celé číslo v little-endian tvaru. Zajímavější je uložení řídicích znaků `[]()#.` a délek řetězců. Řídicí znaky jsou zobrazeny na celá čísla (tzv. řídicí kódy) z rozsahu 0 až 15 (interval je větší než počet řídících znaků pro případ, že by se ukázalo jako vhodné formát nějak rozšířit) a délky řetězců na řídicí kódy od 16 výše (tedy např. řetězec délky 10 bude mít kód 26). Díky tomuto přístupu se délky řetězců nijak neodlišují od jiných řídicích znaků a zjistit další element lze načtením jednoho řídicího kódu.

Rozsah řídicích kódů je 0 až $2^{64}-1$. Pro úsporu místa však nejsou ukládány s pevnou šířkou, ale způsobem připomínajícím UTF-8 [13], např. kódy menší než 128 jsou uloženy v jednom bajtu způsobem `0b0xxxxxxx`, kódy od 128 do $2^{14}-1$ ve tvaru `0b10xxxxxx xxxxxxxx` atd. Tedy počet jedniček na začátku prvního bajtu je o jedna menší, než kolik bajtů celý řídící kód zabírá. Pokud je těchto jedniček méně než osm, následuje po nich nula a až za ní vlastní kód. Je-li první bajt `0xFF`, tak žádná nula vložena není a následuje přímo osm bajtů kódu. Do vyhrazených bitů je kód uložen následovně: Nejvyšší bity jsou uloženy v prvním bajtu a zbylé jsou uloženy do následujících bajtů little-endian způsobem.

Vyšší vrstva

Vyšší vrstva obsahuje popis formátů databáze prostřednictvím prostředků nižší vrstvy. Databáze se skládá z několika souboru, přičemž jméno každého z nich se skládá ze jména konkrétní databáze a přípony. Vždy přítomný je soubor s příponou `.schema`, který obsahuje popis databáze, tj. název typu databáze, seznam typů uzlů a u každého typu uzlu popis jeho atributů. Při otevírání databáze je

tento soubor porovnán s popisem příslušné databáze v paměti, aby se předešlo otevření nekompatibilní databáze.

Ostatní soubory jsou datové a mají jako příponu přirozené číslo. Absolutní hodnota čísla není důležitá, slouží pouze k seřazení souborů do správného pořadí a zjištění, zda v posloupnosti souborů nejsou díry. Přípona není na detekci správného pořadí dostatečná, jelikož je snadné soubory pomíchat, především ve chvíli, kdy se je z snažíme přejmenovat tak, aby začínaly od jedné. Proto obsahuje každý datový soubor na začátku a na konci magický řetězec. Magický řetězec je osm bajtů dlouhý a náhodně generovaný, přičemž jeho hodnota se na konci jednoho souboru a začátku na něj navazujícího souboru shoduje. Tím je zajištěno (s rozumnou pravděpodobností), že se soubory nepomíchají.

Uložení dat ve více souborech je nutné především kvůli možnosti zkrátit log vypsáním databáze do nového souboru. Výhodou takového uspořádání je také možnost velmi snadného vytváření snímků stavu databáze v určitém čase – ve chvíli, kdy chceme vytvořit snímek, založíme nový datový soubor a všechny následující transakce budou zapisovány do něj.

Samotný formát datových souborů je velmi prostý. Prvním kořenovým elementem je pole obsahující typ databáze, k níž soubor patří, pak následuje řetězec obsahující magický řetězec a případně řetězec označující začátek výpisu databáze. První načítaný soubor vždy musí obsahovat výpis databáze. Poté následují jednotlivé transakce, přičemž každá transakce je uložena jako pole na kořenové úrovni. Prvky transakce jsou opět pole. Každé z nich popisuje jednu operaci s databází a to následujícím způsobem. Prvním prvkem je řetězec popisující druh operace (možné operace jsou vytvoření či smazání uzlu a změna hodnoty atributu uzlu), za ním následuje identifikátor uzlu (celé kladné číslo). Dále podle druhu operace buď číslo typu uzlu (pro vytvoření nového uzlu), nic (při mazání), nebo pořadí upravovaného atributu a jeho nová hodnota. Pokud je soubor ukončený (což jsou všechny kromě posledního), obsahuje na konci ukončovací řetězec a magický řetězec navazujícího souboru.

Formát uložení jednotlivých typů atributů je následující: Řetězce (`String` a `RawString`) jsou uloženy jako řetězce stejně jako v paměti, pouze ukončovací `'\0'` u typu `String` se do souboru neukládá. Znaménkové číselné typy (`Int{8,16,32,64}`) a čísla s plovoucí řádovou čárkou (`Float`, `Double` a `LDouble`) jsou uložena jako řetězce tak, jak je architektura AMD64 ukládá v paměti (tedy little-endian tvar a čísla s plovoucí řádovou čárkou jsou v tvaru dle normy IEEE 754 [14]). Bezznaménková celá čísla také zachovávají little-endian tvar, ale pro úsporu místa se nejvyšší nulové bajty neukládají. Ukazatele na uzly jsou na disku reprezentovány identifikátorem uzlu, na nějž odkazují. Ten je uloženo jako bezznaménkové celé číslo.

3.2.3 Vypsání databáze

Důležitou součástí každého formátu databáze založeného na logu je způsob, jakým řeší nahrazení příliš dlouhého logu novým, krátkým se stejným obsahem. První možností by bylo ve chvíli, kdy se rozhodneme, že současný log je příliš dlouhý, zastavit databázi, vypsat její obsah do nového souboru a pokračovat

v práci. Problém toho přístupu je, že čas strávený výpisem databáze může být dlouhý a nemusí být žádoucí aplikaci na tuto dobu zastavit.

Druhou možností, použitou např. v databázi Redis [8], je provést systémové volání `fork`. To „rozdělí“ současné vlákno na dvě, přičemž nově vytvořená kopie se stane samostatným procesem. Tento proces má zkopírované všechny zdroje (paměť i otevřené file deskriptory) svého rodiče. Může tedy vypsat databázi, protože vlastní snímek paměti rodiče ze chvíle, kdy bylo volání `fork` vykonáno. Poté pouze informuje rodiče, že vypsal databázi, ten do ní dopíše kousek logu, co vznik v čase, kdy výpis probíhal, a pokračuje v práci s novým logem. Nevýhody `fork`u spočívají právě v jeho schopnosti vyrobit snímek zdrojů rodičovského procesu. Kromě potenciálně velkého počtu zkopírovaných file deskriptorů, které musí potomek uklidit, ač je vůbec nepotřeboval, může nastat problém s nedostatkem paměti.

Linux implementuje `fork`, pomocí přístupu copy-on-write [15], takže samotné volání je velmi rychlé a efektivní. Pokud ale rodič začne do paměti více zapisovat, jsou měněné stránky kopírovány, protože jsou sdíleny s potomkem, který v nich pořád musí vidět původní data. V extrémním případě tedy potomek spotřebovává tolik paměti, co jeho rodič. Že by se toto stalo s pamětí užívanou hybridní databází je nepravděpodobné, ale velmi snadno může tyto problémy způsobit část programu, která s databází vůbec nesouvisí a pouze hodně zapisuje do paměti. V případě Redisu to nevadí, protože běží jako samostatný proces a má tedy svou paměť i deskriptory pod kontrolou. Naopak databáze implementovaná jako knihovna funkcí nemá žádnou kontrolu nad prostředky, které alokuje zbytek aplikace, takže využití `fork`u je nevhodné.

Proto jsem zvolil komplikovanější přístup, který se ale obejde jak bez zastavení práce s databází, tak bez volání `fork`. Když chceme vypsat databázi, tak je založen nový datový soubor a na jeho začátek je zapsáno, že obsahuje výpis databáze. Nové transakce se zapisují do tohoto nového souboru. Výpis databáze vypadá jako speciální druh transakcí (v souborovém formátu má jako první prvek v poli řetězec nulové délky), které jsou zamíchány mezi běžné transakce. Tyto speciální transakce obsahují výpisy jednotlivých uzlů (současná implementace obsahuje v každé transakci výpis jednoho uzlu). Výpis uzlu je pole, v němž je identifikátor typu uzlu (jeho pořadí v rámci typu databáze, číslováno od 0), identifikátor uzlu a za nimi následují hodnoty atributů. Když jsou vypsány všechny uzlu, je do souboru zapsána značka „konec výpisu“, takže při načítání databáze lze určit, zda vypsání proběhlo úplně, nebo bylo přerušeno.

Samotné vypisování uzlů je prováděno servisním vláknem. To ve chvíli, kdy by mělo z fronty vybrat další požadavek, zkontroluje, zda neprobíhá výpis databáze. Pokud probíhá, tak provede vypsání přednastaveného počtu uzlů (v základním nastavení 5, ale počet je možné změnit konfigurační volbou), a až pak provede další událost čekající ve frontě. Je-li fronta prázdná, vypíše další uzly a opět zkontroluje frontu. Drobným zádrhelem, na který je dobré pamatovat, je, že současná implementace při výpisu obchází frontu transakcí (což je z hlediska výkonu dobře), takže výpis uzlu se v souboru může objevit před transakcí, která ho vytvořila. Proto kód načítající databázi musí být na tuto možnost připraven a správně ji ošetřit, tedy požadavek na vytvoření existujícího uzlu ignorovat a ne vyvolat chybu.

4. Rozhraní databáze

Jak bylo zmíněno výše, hybridní databáze je implementována jako knihovna funkcí v jazyce C. Protože jazyk C neumožňuje provést definice typů uzlů, indexů a databází ve tvaru rozumně zpracovatelném pro člověka i počítač, jsou tyto definice provedeny ve speciálním, k tomuto účelu vytvořeném, jazyce. Tyto definice jsou následně přeloženy do jazyka C. (Při implementaci překladače jsem použil jazyk Perl [16] s knihovnou RecDescent [17].) Popis návrhu hybridní databáze začneme popisem rozhraní z hlediska běžného uživatele, poté představíme jazyk pro generování definic. V příloze 1 naleznete jednoduchý příklad použití databáze.

4.1 Hybridní databáze z pohledu uživatele

Nejprve popíšeme rozhraní, které bude uživatel používat při běžné práci s databází. Datové typy `Node`, `Handle` a `Database` jsou abstraktní třídy. Proto všechny funkce, které je mají jako parametr, jsou v implementaci obaleny makry, takže bez explicitního přetypování je jim jako parametry možné předat i podtřídy daných tříd. Pokud je nějaká funkce má jako návratovou hodnotu, většinou existuje makro s jiným jménem obalující funkci tak, aby přímo vracela správný typ (často se parametry makra mírně liší od funkce, protože kupř. makra berou jako parametry místo deskriptorů typů jejich názvy).

4.1.1 Vytváření a zánik databáze

```
Database *database_create(const DatabaseType *type,
                          const char *file,
                          enum DbFlags flags);
#define dbCreate(Type, file, flags)

void database_close(Database *D);
```

Databázi je možné vytvořit voláním funkce `database_create` nebo makra `dbCreate`. Výhodou makra je, že jeho návratový typ není `Database*`, ale ukazatel na konkrétní podtřídu typu `Database` (uvnitř makro stejně volá funkci `database_create`).

Funkce `database_create` nejprve alokuje a inicializuje objekt databáze. Poté zkontroluje schéma databáze `file.schema` (tj. soubor obsahující popis typů uzlů a jejich atributů, které databáze může obsahovat) a načte data uložená v souborech `file.N`, kde `N` je přirozené číslo. Čísla souborů nemusí začínat od jedné, ale posloupnost nesmí obsahovat mezeru, jinak vytváření databáze skončí chybou. (Protože dává smysl smazat první soubor za předpokladu, že databáze byla vypsána do druhého souboru, ale není možné odstranit nějaké transakce uprostřed historie.) První soubor v posloupnosti také musí obsahovat výpis databáze. (Podobnější informace o diskovém formátu naleznete ve třetí kapitole.)

Když jsou všechna data v paměti, proběhne inicializace indexů a na závěr jsou spuštěna pomocná vlákna.

Pomocná vlákna jsou dvě – servisní vlákno a vlákno časovače. Servisní vlákno se stará o zápis transakcí na disk. Ty jsou mu předávány pomocí globální fronty jednotlivými transakčními vlákny. (Míra zpracování transakcí v okamžiku jejich předání servisnímu vláknu závisí na konfiguračních volbách databáze.) Kromě toho ještě zajišťuje vypsání databáze na disk a zakládání nových souborů. Servisní vlákno transakce nezapisuje okamžitě (není-li to explicitně vyžadováno), ale ukládá je do bufferu, který je zapsán, až když je plný. Aby bylo zajištěno, že i pří nízkém množství zápisů budou data zapsána v konečném čase, existuje vlákno časovače. Vlákno časovače periodicky (v základním nastavení každých 5 s) posílá zprávu servisnímu vláknu, aby zapsalo obsah bufferu na disk a zavolalo funkci `fsync`. Ta přinutí systém data skutečně zapsat, a ne držet je ve vlastních bufferech.

Databáze se uzavírá voláním `database_close`. Je na uživateli databáze, aby zajistil, že v době tohoto volání již žádné vlákno s databází nepracuje.

4.1.2 Handly a transakce

```
#define dbHandleCreate(D)
Handle *db_handle_create(Database *D);

void db_handle_free(Handle *H);

void tr_begin(Handle *H);
void tr_abort(Handle *H);
bool tr_commit(Handle *H, enum CommitType commit_type);

void tr_hard_abort(Hadler *H);
```

Dříve než může vlákno začít transakci, potřebuje handle. Handle představuje jakýsi kontext transakce – obsahuje transakční log, readset a všechny další nezbytnosti pro její běh. Transakce samy nemají podobu objektů, ale vždy je funkcím v rámci transakce předáván handle. Výhodou je, že jeden handle může být postupně využit pro mnoho transakcí, čímž se šetří opakované alokace paměti, a protože databáze potřebuje mít přehled o běžících transakcích, i množství práce s centrální strukturou.

Funkce `tr_begin` zahájí novou transakci. Co skutečně vykoná závisí na tom, zahajujeme-li hlavní transakci nebo běží-li již nějaká transakce a zahajujeme pouze vnořenou transakci. Práce s hlavní transakcí závisí na implementaci STM, která byla popsána ve druhé kapitole, takže se ji zde nebudu věnovat. Naopak vnořené transakce se odehrávají zcela v rámci příslušné hlavní transakce a fungují jako „záložky“ v transakčním logu – když je zahájena vnořená transakce, je na zásobník vnořených transakcí, který každý handle obsahuje, umístěna položka popisující současný stav transakčního logu a readsetu.

Obdobně při úspěšném ukončení transakce pomocí `tr_commit` nebo jejím zrušení voláním `tr_abort`. Při ukončení vnořené transakce pomocí `tr_commit`,

volání `tr_commit` vždy uspěje a pouze odstraní položku z vrcholu zásobníku vnořených transakcí, čímž se daná vnořená transakce stane součástí nadřazené transakce. Parametr `commit_type` určuje, zda bude commit synchronní, tj. funkce `tr_commit` se vrátí až po zapsání celé transakce na disk, či asynchronní, kdy návrat z funkce proběhne okamžitě po předání dat servisnímu vláknu.

Funkce `tr_abort` a `tr_hard_abort` provedou rollback transakce (tedy vrátí změny, které transakce doposud provedla). Rozdíl mezi nimi je ten, že `tr_abort` zruší pouze aktuální (vnořenou) transakci, zatímco `tr_hard_abort` vždy zruší celou hlavní transakci. To je užitečné pokud nějaká operace uvnitř transakce selže z důvodu kolize s jinou transakcí. V ideálním světě by v tu chvíli příslušná operace sama provedla rollback a restart transakce (například vyvoláním k tomu určené výjimky). Jelikož ale jazyk C neobsahuje výjimky a použití nějaké konkrétní implementace výjimek by uživatele nutilo používat danou implementaci v celém jeho kódu, vracejí všechny funkce, které mohou selhat, hodnotu typu `bool` přičemž `true` značí úspěch a `false` selhání. Při selhání je pak na uživateli, aby zajistil zrušení transakce voláním `tr_hard_abort` a její následné restartování.

4.1.3 Makra pro usnadnění práce s transakcemi

```
#define trFail

#define trBegin
#define trCommit(commit_type, restart_action)
#define trAbort
```

Pro pohodlnější práci s transakcemi jsme implementovali i makra, která doplňují jistou podporu pro automatické restartování transakcí. Makra `trBegin` a `trCommit` spolu tvoří závorky obalující transakci. Pokud selže commit nebo nějaká operace uvnitř transakce, tak se transakce sama zruší a restartuje. Aby se předešlo live-locku, transakce před restartem počká náhodný čas, přičemž maximální čas čekání exponenciálně roste v závislosti na tom, o kolikátý restart dané transakce se jedná.

Jak tato makra zjistí, že operace uvnitř transakce selhala, když nemají k dispozici výjimky? Jednoduše – každé makro při selhání vykoná makro `trFail`. To je defaultně nastaveno jako skok na návěstí `tr_failed`, které je definované uvnitř makra `trCommit`. Tento přístup má samozřejmě svá omezení, např. nefunguje transparentně, pokud práce s databází probíhá uvnitř volání funkce. Naopak ale umožňuje uživateli přizpůsobit ho svým potřebám předefinováním makra `trFail`. Z důvodu jisté omezenosti toho přístupu má makro `trCommit` jako druhý, volitelný parametr kousek kódu, který je proveden, pokud daná transakce selže a není transakcí hlavní. V tom případě nemůže být přímo restartována, ale musí propagovat selhání až do hlavní transakce.

Makra `trBegin`, `trCommit` a `trAbort` pracují s handlem jménem `H`. Pokud je nutné použít jiný handle, existují k těmto makrům varianty, které mají na konci jména podtržítko a mají navíc jako první parametr handle.

4.1.4 Práce s uzly a indexy

```
#define trNodeCreate(NodeType)
#define trNodeDelete(node)

#define trRead(node, Attribute)
#define trWrite(node, Attribute, new_value)

#define trUpdateIndexes(node)

#define nodeCast(NodeType, node)
```

Stejně jako výše uvedená makra, i tato pracují s handleem jménem `H` (kromě makra `nodeCast`) a při selhání provedou makro `trFail`. Makro `trNodeCreate` vytvoří nový uzel daného typu a aktualizuje příslušné indexy. Obdobně makro `trNodeDelete` uzel smaže a provede aktualizaci indexů. Naopak makro `trWrite`, které slouží pro zápis do existujícího uzlu, po svém provedení indexy neaktualizuje.

K tomuto chování má dva důvody: Za prvé není neobvyklé, že v rámci jedné transakce je jednomu uzlu modifikováno několik atributů, a pokud by indexy byly aktualizovány po zápisu každého z nich, představovalo by to zbytečnou režii. Druhý důvod je možná ještě důležitější – pro indexy může být výhodné předpokládat, že každý uzel splňuje určité invarianty, čehož není možné dosáhnout, je-li index aktualizován po každém zápisu, protože přechod mezi korektními stavy může vyžadovat modifikaci několika atributů. Proto existuje makro `trUpdateIndexes`, které informuje indexy o změně příslušného uzlu. Toto makro je nutné zavolat na každý modifikovaný uzel před koncem příslušné transakce.

Makro `trRead` slouží ke čtení hodnot atributů uzlů. Jeho důležitou vlastností je, že poskytuje konzistentní pohled na data (problematika konzistentního čtení je důkladněji probrána v následující kapitole), takže uživatel je odstíněn od změn prováděných jinými transakcemi.

4.1.5 Práce s indexy

```
#define trIndex(jméno_indexu, metoda_index, parametry...)
```

Jak bylo zmíněno výše, indexy jsou klíčem pro přístup k jednotlivým uzlům. V rámci této práce rozumíme indexem libovolnou datovou strukturu, jejíž stav je jednoznačně určen stavem databáze, nad níž je vytvořena. Protože jsou indexy, jako jediný způsob pro (počáteční) přístup k uzlům databáze, pro práci s ní klíčové, byly navrhovány se snahou vytvořit je co nejobecnější při zachování jednoduchosti práce.

Z hlediska databáze je rozhraní indexů velmi jednoduché – každý index se skládá z kontextu, tedy vlastní datové struktury (která je databází automaticky alokována) a callback funkce, která je volána při změně (relevantní části) databáze. Callback má jako parametry kontext, typ události, uzel, k němuž se daná událost váže, a samozřejmě handle, protože update indexů vždy probíhá v rámci příslušné transakce. Protože je i celý kontext indexu součásti transakční paměti,

chová se z pohledu ostatních vláken stejně jako zbytek databáze – především tedy ostatní vlákna vždy vidí index v konzistentním stavu. Kromě callback funkce obsahuje každý index ještě funkce pro inicializaci kontextu a uvolnění prostředků indexem užívaných.

Z pohledu uživatele je rozhraní indexu zcela v režii jeho tvůrce. K jednotlivým metodám se přistupuje pomocí makra `trIndex`. Toto makro obaluje skutečnou funkci, která má hlavičku:

```
bool jméno_metody(const Typ_kontextu* context, Handle *H,
                  NávratovýTyp *value, další_parametry...)
```

Návratová hodnota je vždy typu `bool` a signalizuje, zda bylo volání úspěšné, nebo došlo ke kolizi a je nutné transakci restartovat (`true` značí úspěch). Toto chování je uživateli neviditelné. V případě selhání makro `trIndex` provede `trFail` a bude tak selhání propagovat dále. V případě úspěchu vrátí hodnotu `*value`. Z pohledu tvůrce indexu stojí za pozornost, že uvnitř definice metody indexu je makro `trFail` definováno jako `return false` místo běžné definice `goto tr_failed`.

4.2 Generování definic

Součástí ukázkové implementace je i jazyk pro deklarování typů indexů, uzlů a databází. Původně jsem se chtěl obejít bez tvorby nového jazyka, ale ukázalo se, že tyto typy není možné dost přehledně deklarovat pouze pomocí jazyka C a jeho vestavěného preprocesoru. Proto jsem vytvořil jazyk pro tyto definice. Zdrojový kód v tomto jazyce je překládán pomocí skriptu v jazyce Perl [16] do jazyka C. Kromě deklarací typů užívám obdobný jazyk i pro generování funkcí pro práci s jednotlivými typy atributů. Perlový překladač staví na knihovně RecDescent [17], která umožňuje snadné parsování textu na základě jeho gramatiky. Nejprve ukážeme základní syntaxi navrženého jazyka, a poté popíšeme jednotlivé typy definic.

4.2.1 Základní syntaxe

file `:= (` *interface* `|` *implementation* `|` *include* `|` *node_type* `|`
index_type `|` *database_type* `)*`

interface `:= 'Interface'` *block*
implementation `:= 'Implementation'` *block*
include `:= 'Include' /"([^"]+)"/`

Nejprve pár slov k syntaxi. Jazyk rozlišuje velikost písmen. Parser zpracovává kód po tokenech, které mohou být odděleny libovolným počtem bílých znaků. Za bílý znak je považován i středník a komentáře, které začínají znakem `#` a pokračují do konce řádku. Element `block` představuje blok kódu uzavřený ve složených závorkách. Ty se mohou uvnitř bloku vyskytovat za předpokladu, že jsou spárované. Celý blok je považován za jeden token.

Element `Include` slouží k načtení obsahu jiného souboru. Jméno souboru v uvozovkách je interpretováno jako relativní či absolutní, nedochází k prohledávání žádných adresářů s hlavičkovými soubory. Při vkládání souboru není kontrolováno opakované vložení souboru. Sémanticky odpovídá direktivě `#include` jazyka C. Elementy `Interface` a `Implementation` slouží k vložení bloku kódu v jazyce C před vygenerované deklarace resp. definice. (Tedy pokud v implementaci indexu budu chtít použít funkce z hlavičkového souboru `foo.h`, tak do kódu vložím `Implementation { #include "foo.h" }`.) Těchto elementů se v jednom souboru může nacházet libovolné množství, přičemž ve výsledném souboru jsou za sebou tak, jak byly uvedeny.

4.2.2 Definice uzlů

node_type `:= 'NodeType'` *name* `'{'` *node_attribute*`* '}'`
node_attribute `:= 'Attribute'` *name* `':'` *attribute_type*

Typy uzlů se definují pomocí klíčového slova `NodeType`. Definice pak obsahuje jména atributů a jejich typy, přičemž na pořadí atributů záleží. Současně implementované jsou následující typy atributů (které pokrývají základní typy jazyka C):

- Znaménkové typy `Int8`, `Int16`, `Int32` a `Int64` jsou v paměti realizovány datovými typy `int{8,16,32,64}_t`.
- Neznaménkové typy `UInt8`, `UInt16`, `UInt32` a `UInt64` jsou v paměti realizovány datovými typy `uint{8,16,32,64}_t`.
- Reálná čísla `Float`, `Double` a `LDouble` jsou realizovány datovými typy `float`, `double` a `long double`.
- Typ `String` představuje textový řetězec ukončený znakem `'\0'` a je realizován datovým typem `char*`.
- Typ `RawString` představuje řetězec, který může obsahovat hodnotu `'\0'`. V paměti je reprezentován typem `struct RawString`.
- Typ `Pointer` představuje ukazatel na jiný uzel. Z technických důvodů je realizován jako `union` identifikátoru uzlu a ukazatele na uzel. Z pohledu uživatele se jeví jako pouze jako `Node*`. Použití `union`u je nutné, protože při načítání dat ze souboru je do atributu typu `Pointer` nejprve uložen identifikátor uzlu, na který ukazuje, a až když jsou načteny všechny uzly, je možné je projít a změnit identifikátory uzlů na ukazatele na ně.

4.2.3 Definice indexů

index_type `:= 'IndexType'` *name* `':'` *type_of_context*
`'{'` *i_init i_destroy i_update i_method*`* `'}'`
i_init `:= 'Init'` *block*
i_destroy `:= 'Destroy'` *block*
i_update `:= 'Update'` *block*
i_method `:= 'Method'` *name* `(` `'('` *arguments...* `')'` `)?` `':'`
return_type block

Každý typ indexu se skládá alespoň z typu kontextu, konstruktoru, destruktoru a metody `Update`. Dále by měl obsahovat alespoň jednu metodu, protože metody indexu jsou jediným způsobem, jak z něj získat informace. Typ kontextu je datový typ, který bude sloužit jako kontext pro příslušný typ indexu, tj. ukazatel na objekt toho typu bude předáván všem metodám indexu.

Konstruktor a destruktor (tedy metody `Init` a `Destroy`) jako jediné pracují mimo transakce, protože pracují ještě před startem, resp. po zastavení transakční paměti. Obě mají jako parametr kontext a `GenericAllocator`, pomocí něhož mohou alokovat, resp. uvolňovat paměť. K práci nemohou použít `malloc` a `free`, protože celý index musí být součástí transakční paměti, aby mohl správně fungovat případný rollback transakce.

Metoda `Update` slouží k aktualizaci indexu a je kromě konstruktoru a destruktoru jedinou metodou, která smí upravovat kontext. Jejími parametry jsou kontext, handle transakce, druh události a uzel, jehož se událost týká. Událost smazání uzlu je provedena dříve, než je příslušný uzel smazán, takže metoda `Update` s ním může pracovat. Naopak při vytvoření uzlu je událost vyvolána, až když je uzel vytvořen a inicializován. V metodě `Update` lze k práci s pamětí používat funkce `tr_memory_alloc()` a `tr_memory_free()`. Obě mají jako první parametr handle.

Uživatelské metody vytvořené pomocí klíčového slova `Method` mají jako parametry kontext, handle transakce, ukazatel na návratovou hodnotu a uživatelem definované argumenty. Uživatelské metody stejně jako `Update` vždy probíhají v rámci transakce a mají návratovou hodnotu typu `bool`, která značí, zda metoda byla úspěšná (`true`), nebo došlo ke kolizi a je třeba restartovat transakci (`false`). Proto uživatelské metody nemohou vracet žádnou další informaci přímo, ale je nutné využít parametru typu ukazatel.

4.2.4 Definice databází

database_type `:= 'DatabaseType'` *name* `'('` *version* `')'`
`'{' (` *node* `|` *index* `)? '}'`
node `:= 'NodeType'` *name*
index `:= 'Index'` *name* `'for' (` *node_type* `| 'all' ) ':'` *index_type*

Databáze je určena jménem, verzí (což je libovolné „slovo“, jediným účelem je moci rozlišit mezi různými verzemi databáze téhož jména) a typy uzlů, které obsahuje. U typů uzlů záleží na pořadí, v němž jsou uvedeny. Důležité je, že definice indexů na disk ukládána není, takže je možné používat jednu databázi v různých programech s různými indexy. Každý index je určen jménem, pod nímž bude uživatelům přístupný, svým typem a typem uzlů, na který je navázán. Je-li jako typ uzlu uvedeno klíčové slovo `all`, je index informován o změnách všech uzlů v databázi.

5. Ukázková implementace

Tato kapitola popisuje specifika provedené implementace a její omezení. Nejprve je krátce pojednáno o zvoleném programovacím jazyce a platformě. Následuje popis správy paměti a dalších vnitřních mechanismů, především generování definic typů databází / uzlů / indexů uživatelem. Na závěr jsou zmíněny omezení provedené implementace a výsledky několika jednoduchých benchmarků.

Ukázková implementace je napsána pod licencí GPL verze 2 [18]. Jelikož nejsem zběhlý v programování v assembleru a překladač GCC neposkytuje vhodné atomické operace, převzal jsem implementaci atomických operací z linuxového jádra verze 3.3.4 [19]. (Jedná se o soubor `utils/atomic_amd64.h`, je pod licencí GPL verze 2, čímž implikuje licenci pro celý projekt, neboť ten lze považovat za odvozené dílo.)

5.1 Jazyk implementace

Ukázková implementace je provedena v jazyce C, přesněji v jeho verzi C99 [20] s použitím GNU rozšíření a vestavěných funkcí překladače GCC dostupných ve verzi 4.5 [21]. Jako implementační platforma byl zvolen Linux na architektuře AMD64.

Architektura AMD64 byla upřednostněna před IA-32, protože na rozdíl od ní obsahuje všechny 8-bajtové atomické instrukce (především `xadd`, `xchg` a `cmpxchg`) nutné pro implementaci verzovaných zámků a dalších struktur pracujících atomicky (např. globální hodiny či některé alokátory zmíněné níže vnitřně užívají atomické instrukce). Implementaci je možné portovat i na procesory architektury IA-32, které podporují 8-bajtovou instrukci compare-exchange (`cmpxchg8`), což jsou procesory Intel Pentium a novější. Je však nutné neexistující instrukce emulovat pomocí `cmpxchg8`.

Proč jsem použil GNU rozšíření jazyka C a nezůstal u standardizované verze C99 [20]? Hlavním důvodem byla podpora operátoru `typeof` a vestavěných funkcí `__builtin_types_compatible` a `__builtin_choose_expr`. Díky nim je možné implementovat makra `trRead` a `trWrite` tak, že jako parametry mají pouze uzel a jméno jeho atributu s nímž pracují, přičemž potřebná typová informace se odvodí uvnitř makra.

5.2 Správa paměti

5.2.1 Opožděné uvolňování paměti

Když transakce uvolní paměť, může nastat problém, pokud jiná transakce drží na tuto paměť ukazatel a pokusí se ho dereferencovat. Původní řešení v TL2 enginu bylo užití instrukcí, které při pokusu o čtení paměti, kam nemá program

přístup, vrací nulovou hodnotu. Architektury IA-32 ani AMD64 žádné takové instrukce nemají, a tak se port TL2 enginu tyto architektury [5] s tímto problémem musel vypořádat jinak.

Řešením v [5] bylo odchytávat signál neoprávněného přístupu do paměti. Ač je tento signál možné odchytit, tak jedinou užitečnou informací, kterou handler dostane je adresa, k níž bylo přistupováno. Ta však nestačí k ověření, zda důvodem tohoto signálu byl skutečně přístup k uvolněné transakční paměti či chyba v programu, takže handler musí signál prostě zahodit, čímž může skrýt chybu v programu. Dalším problémem je, že handler neoprávněného přístupu do paměti se nesmí vrátit, neboť to způsobí nekonečnou smyčku. Proto musí být užita nějaká varianta nelokálních skoků (např. volání `setjmp` / `longjmp`). Z důvodu režie dává smysl použít `setjmp` pouze na počátku transakce a pomocí `longjmp` transakci rovnou restartovat, což je v rozporu s tím, že ukázková implementace neobsahuje automatické restartování transakcí. Navíc registrací signal handleru se můžeme dostat do konfliktu s jinou částí aplikace, protože pro každý signál může být registrován pouze jeden handler. Naopak výhodou tohoto přístupu je kromě jednoduchosti i výkon, jelikož šance, že transakce A načte ukazatel, transakce B uvolní paměť odkazovanou tímto ukazatelem a transakce A ukazatel dereferencuje, je velmi malá.

V této práci jsem místo tohoto přístupu použil opožděné uvolňování paměti – paměť není uvolněna, když transakce žádá její uvolnění, ale až ve chvíli, kdy je jisté, že žádná transakce na ni nemůže držet ukazatel. Tohoto chování je docíleno tak, že transakční paměť se nealokuje pomocí volání `malloc`, ale pomocí volání `node_allocator_alloc` pro alokaci uzlů a `generic_allocator_alloc` pro zbylou paměť užívanou v transakcích. Při uvolňování je předán jako parametr i čas, kdy byla paměť uvolněna. (Pokud byla paměť uvolněna v čase x, tak na ni mohou mít pointer pouze transakce, které začaly nejpozději v čase $x - 1$.) Skutečné uvolnění paměti ve správnou dobu pak zajistí příslušný alokátor.

Výhodou opožděného uvolňování paměti je, že nemůže zakrývat chyby při práci s pamětí, ani nevyžaduje použití `setjmp` / `longjmp`. Nevýhodou může být nutnost udržovat uvolněné bloky v paměti a s tím spojená režie. Tuto režii je možné vyvážit tím, že bloky k uvolnění budeme používat k uspokojení požadavků na alokaci nové paměti, jak to činí `VPageAllocator`.

5.2.2 GenericAllocator

`GenericAllocator` je určen pro alokaci různě velkých bloků paměti užívané v transakcích. V současné implementaci pouze zdržuje uvolnění paměti, tím způsobem, že uvolňovanou paměť umístí do vnitřního seznamu a když velikost seznamu přesáhne danou mez (makro `DB_GENERIC_ALLOCATOR_CACHE`), tak zjistí počáteční čas nejstarší transakce t a uvolní ze seznamu paměť starší než t. Uvnitř `GenericAllocatoru` je k práci s pamětí užívány funkce `malloc` a `free`, protože jejich implementace v knihovně `glibc` je dobře optimalizovaná a implementace rychlého alokátoru paměti je velmi netriviální záležitost.

5.2.3 NodeAllocator

`NodeAllocator` slouží k alokaci uzlů. Zatímco `GenericAllocator` je globální pro každou instanci databáze, `NodeAllocator` má v rámci databáze každý typ uzlu vlastní. Protože jsou všechny uzly jednoho typu stejně velké, je implementace `NodeAllocatoru` více optimalizovaná a odpovídá struktuře alokátoru SLAB [22] pro malé objekty. Tím je dána i limitace současné implementace – uzel nemůže být větší než cca 4000 bajtů (velikost stránky minus hlavičky).

Jako v alokátoru SLAB, jsou objekty alokovány z větších bloků – v současné implementaci bloky přímo odpovídají stránkám paměti – a ty jsou uloženy jako spojový seznam, přičemž ne zcela zaplněné bloky jsou udržovány na konci seznamu. Je-li potřeba alokovat prvek, je alokován z bloku na konci seznamu. Pokud je tím blok zaplněn, je přesunut na počátek seznamu. Pokud je poslední blok zcela plný, alokuje se nový blok.

Pro alokaci bloků je užíván `VPageAllocator`. Vyprázdněný blok je okamžitě předán `VPageAllocatoru` a jako čas uvolnění je předán čas uvolnění posledního uzlu z daného bloku (což je korektní, protože posloupnost časů uvolnění je neklesající).

Významným důvodem pro použití alokátoru typu SLAB byla potřeba šetřit místo. Kromě ukazatelů spojového seznamu bylo možné z uzlů odstranit i ukazatel na typ uzlu. Ten se přesunul na počátek bloku a je k němu možné se dostat díky tom, že bloky odpovídají stránkám paměti, které jsou vždy zarovnané na svou velikost. Pokud by uzly byly alokovány např. `malloc`em, musel by v hlavičce každého z nich být uložen ukazatel na typ uzlu, k němuž přísluší, a dva ukazatele, které by všechny uzlu propojily do spojového seznamu (nutné pro jejich uvolnění při uzavírání databáze a pro vypsání databáze na disk). Připočteme-li režii `malloc`u, uspoříme použitím alokátoru typu SLAB až 40 bytů na uzel, což může být víc než je velikost užitečných dat v daném uzlu.

5.2.4 PageAllocator

`PageAllocator` slouží jako zdroj stránek pro jiné alokátory a strukturu `FastStack` (implementaci zásobníku, která alokuje paměť po blocích, používanou pro transakční log). Stránky alokuje po blocích pomocí funkce `mmap` a nepoužité stránky ukládá do spojového seznamu pro další využití. Pokud množství volných stránek přesáhne daný limit, jsou navráceny systému voláním `munmap`. V celé aplikaci typicky existuje jeden globální `PageAllocator`.

5.2.5 VPageAllocator

`VPageAllocator` je tenká vrstva nad `PageAllocator`em, která se stará o opožděné uvolňování paměti. Funguje jako `GenericAllocator` s tím rozdílem, že stránky v seznamu může použít pro uspokojení nových alokací. Existuje jeden pro každou instanci databáze.

5.3 Omezení ukázkové implementace

Protože implementace, která je součástí této práce, je zamýšlena spíše jako ukázková a základ pro další testování různých implementací konkrétních částí databáze (různé varianty servisního vlákna či přístupu k zamykání atp.), než jako kód připravený pro ostré nasazení, obsahuje také jistá omezení. Ta se pokusím shrnout v této sekci.

První omezení se týká práce se signály. Pokud vlákno přijme signál a začne vykonávat uživatelem přiřazený handler ve chvíli, kdy je zablokováno v určitých systémových voláních (např. `sem_wait`), skončí tato volání s chybou `EINTR`, pokud vracejí chybový status. Co hůře systémové volání `write` může skončit s tím, že zapsalo pouze část dat. Chybě `EINTR` se lze vyhnout, je-li při registraci signal handleru použit příznak `SA_RESTART`. S ním systém dané volání místo vrácení chyby restartuje. Protože současná implementace chybu `EINTR` neošetřuje, je při registraci handlerů signálů nutné použít příznak `SA_RESTART`.

Další omezení spočívají v implementaci alokátorů paměti. Méně významné z nich souvisí s `NodeAllocator`em. Protože ten staví na `PageAllocator`u, je omezena maximální možná velikost uzlu na cca 4000 bajtů (jedna stránka mínus hlavička přidaná `NodeAllocator`em). Vzhledem k tomu, že největší z atributů má velikost 10 bajtů (`long double`), a tedy uzel přesahující velikost 4000 bajtů by musel mít přes 400 atributů, nepovažuji toto omezení za významné. Navíc k jeho odstranění by bylo potřeba alokaci paměti významně pozměnit (pravděpodobně by způsob alokace pro uzly pod touto hranicí zůstal stejný a byl by přidán speciální způsob alokace pro uzly příliš velké (např. skrz `GenericAllocator`)).

Poslední problém, který tu zmíním se týká `GenericAllocator`u. Současná implementace pouze odkládá uvolňování paměti (jak bylo popsáno výše), ale není schopna paměť recyklovat. Hlavním důvodem k tomuto chování je nemožnost zjistit velikost u bloku alokovaného pomocí funkce `malloc`. Aby tedy bylo možné bloky čekající na uvolnění použít znovu, bylo by nutné de facto implementovat celý alokátor pro paměťové bloky různé velikosti. Jelikož toto samo o sobě je velmi komplikované, pokud chceme získat maximální výkon, a s ohledem k optimalizacím, které jsou v GNU implementaci obsaženy (jelikož je dlouhodobě používána a vylepšována), jsem usoudil, že tato implementace bude stále rychlejší než neuměle napsaný obecný alokátor s integrovaným recyklováním bloků paměti.

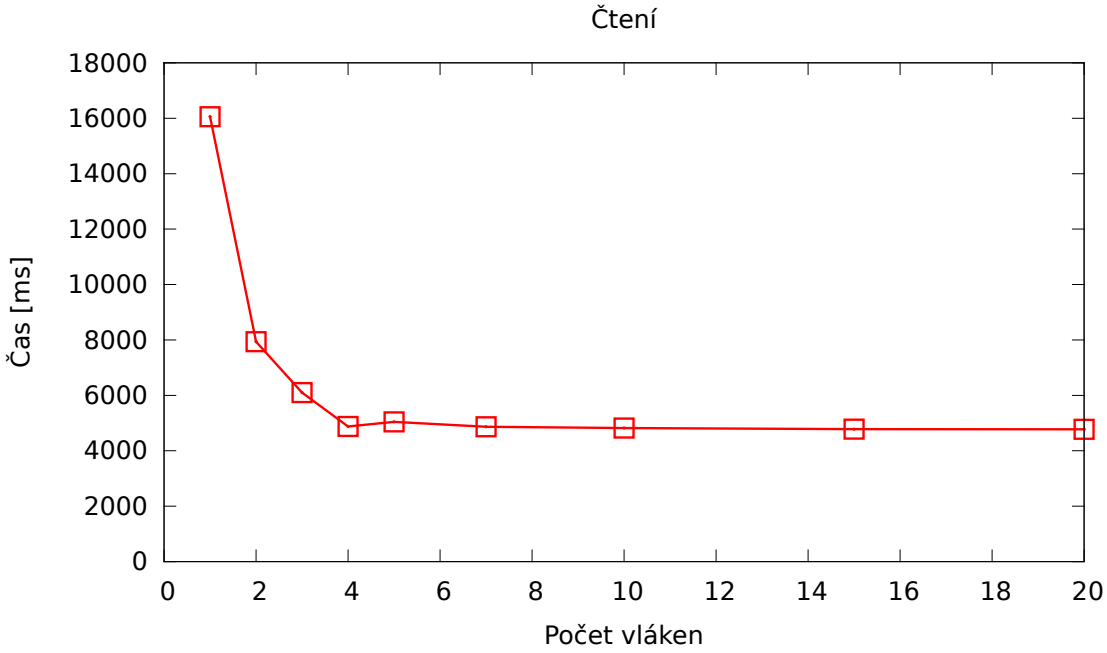
5.4 Výsledky testů

Na závěr této kapitoly uvede ještě výsledky několika testů. Jde o jednoduché syntetické benchmarky, které prověřují škálování ukázkové implementace při provádění různých operací a při různých nastaveních databáze. Data, ze kterých byly vytvořeny tyto grafy, naleznete v příloze 2.

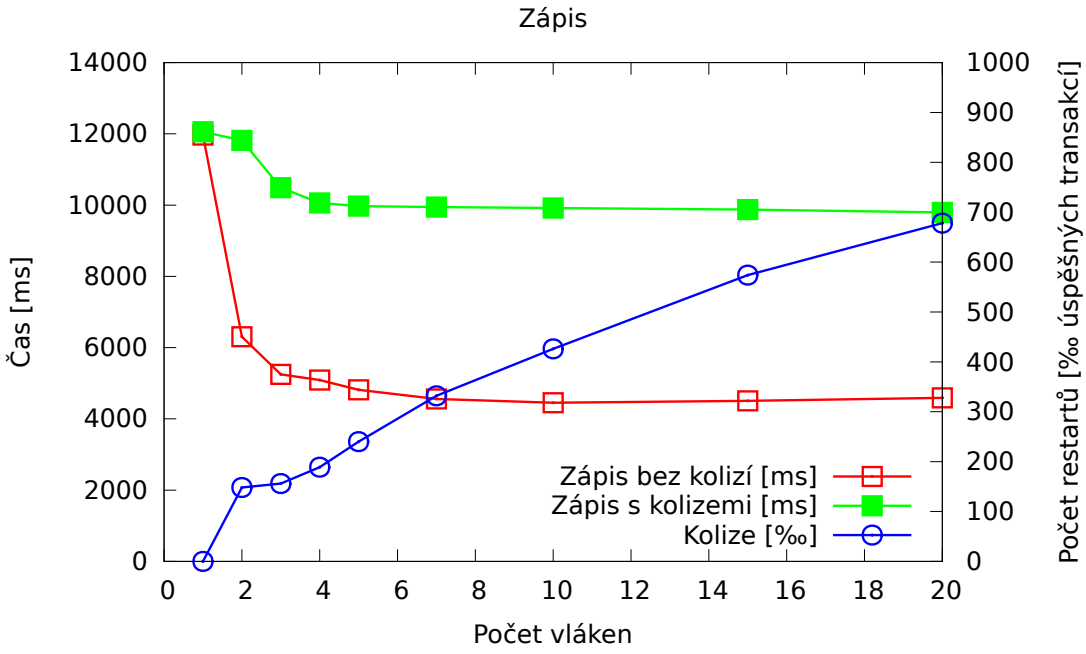
Všechny testy probíhaly na notebooku s procesorem Intel Core i5 520M, který má dvě fyzická jádra doplněná HyperThread technologií (takže má 4 logická jádra), 8 GB paměti a Linuxem verze 3.3.4. Frekvence procesoru byla pevně nastavena na 2133 MHz, abych předešel zkreslení výsledků v důsledku technologie TurboBoost. Při všech testech byla databáze vytvořena s příznakem `DB_READ_ONLY`,

který způsobí, že výstup je místo do souboru s databází zapisován do `/dev/null`, aby výsledky měření nebyly ovlivněny rychlostí pevného disku.

První test zjišťuje škálování uvedené implementace při čtení. V každém testu byl program spuštěn s různým počtem vláken, přičemž mezi běhy zůstával stejný celkový počet vykonaných transakcí a odpovídajícím způsobem se měnil počet transakcí vykonaných jedním vláknem. Z grafu je patrné, že implementace dle očekávání škáluje do čtyř vláken a poté rychlost zůstává konstantní.



Test zápisu také podává dobré výsledky, pokud spolu transakce nekolidují. Pokud začnou pracovat na stejných datech, je zřejmé výrazné zpomalení v důsledku restartování transakcí. Jelikož cílovou skupinou jsou aplikace s nízkým množstvím zápisů, tak to příliš nevadí.



Závěr

V této práci jsem se pokusil navrhnout a naprogramovat datovou strukturu, která kombinuje výhody databází jako perzistenci a transakce s rychlostí běžných datových struktur. Výsledkem práce je návrh databáze, která udržuje data v paměti, implementované jako knihovna funkcí. Práce s databází a podpora transakcí je realizována pomocí konceptu transakční paměti. Díky tomu je s databází možné snadno pracovat paralelně z více vláken.

Stále ale zbývá mnoho možností, jak současnou implementaci rozšířit či vylepšit. Technicky nejjednodušší způsob zrychlení současné implementace je možnost přepsat některé části, především výpočet CRC, s pomocí SSE instrukcí a tak je významně urychlit. Mezi podstatně složitější, ale v jistém směru užitečnější vylepšení patří implementace mechanismu pro automatické restartování transakcí.

Současná implementace sice obsahuje makra `trBegin` a `trCommit`, která automatické restartování umožňují, ta ale používají exponenciálně dlouhé čekání před dalším pokusem. To je výhodné v situaci, kdy jsou transakce malé a množství kolizí nízké. Implementace pokročilejšího algoritmu by nejspíše vyžadovala možnost nějak zvýhodnit konkrétní transakce (především ty, které již opakovaně selhaly), což nyní není možné.

Dalším rozšířením, které by mohlo být velmi užitečné z praktického hlediska, je zavedení složených typů atributů jako jsou seznamy či slovníky (množiny párů klíč–hodnota s rychlým hledáním podle klíče). Hlavní obtíží v tomto případě je návrh vhodného rozhraní pro práci s nimi a serializaci daných operací na disk. Všechny současné atributy si totiž vystačí s operacemi „přečti hodnotu“ a „přiřaď hodnotu“, zatímco pro složené atributy jsou tyto operace krajně nevhodné a bylo by nutné zavést operace nové.

Posledním možným rozšířením, o kterém se zmíním, jsou rozšířené kontexty indexů – tedy že by kontext indexu nebyla pouze jeden objekt globální pro celou databázi, ale indexy by mohly ukládat data i do jednotlivých uzlů. To by mohlo zrychlit operace, kdy je třeba uzel vyhledat v rámci datové struktury představující index.

Seznam použité literatury

[1] Knight, Tom. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, 1986.

[2] Shavit, Nir and Touitou, Dan. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. August 1995.

[3] Transactional Memory. In: *GCC Wiki* [online, cit. 2012-05-23]. Dostupné z: `http://gcc.gnu.org/wiki/TransactionalMemory`

[4] Reinders, James. *Transactional Synchronization in Haswell* [online], 2012 [cit. 2012-05-23]. Dostupné z: `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`

[5] da Cunha, Goncalo Vasco Trincao Bento. *Consistent State Software Transactional Memory*. Master thesis. Faculdade de Ciencias e Tecnologia, Universidade Nova de Lisboa, 2007.

[6] Dice, Dave, Shalev, Ori and Shavit, Nir. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing* (DISC 2006), pages 194–208, Stockholm, Sweden, 2006.

[7] Shimpi, Anand Lal. *The SSD Anthology: Understanding SSDs and New Drives from OCZ* [online], 2009 [cit. 2012-05-23]. Dostupné z: `http://www.anandtech.com/print/2738`

[8] Sanfilippo, Salvatore et al. *Redis* [program], Version 2.4.14, 2009. Dostupné také z: `http://redis.io/download`

[9] Koopman, Philip. 32-Bit Cyclic Redundancy Codes for Internet Applications. In *The International Conference on Dependable Systems and Networks*, pages 459–468, 2002.

[10] Satran, Julian et al. RFC 3720, *Internet Small Computer System Interface (iSCSI)*. The Internet Society, 2004.

[11] Btrfs design. In: *btrfs Wiki* [online, cit. 2012-05-23]. Dostupné z: `https://btrfs.wiki.kernel.org/index.php/Btrfs_design`

[12] Intel Corporation. *Intel SSE4 Programming Reference* [online]. 2007 [cit. 2012-05-23]. Dostupné z: `http://softwarecommunity.intel.com/isn/Downloads/Intel%20SSE4%20Programming%20Reference.pdf`

[13] The Unicode Consortium. *The Unicode Standard*. Version 6.1.0. Mountain View, CA: The Unicode Consortium, 2012. ISBN 978-1-936213-02-3

[14] IEEE 754:2008, *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., 2008.

[15] Love, Robert. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, Inc., 2007.

[16] Wall, Larry et al. *Programming Language Perl* [program]. Version 5.4.12. 2010. Dostupné také z: `http://www.perl.org/get.html`

[17] Conway, Damian and Braun, Jeremy T. *Parse::RecDescent – Generate Recursive-Descent Parsers* [program]. Version 1.965.1. 2003. Dostupné také z: `http://search.cpan.org/~dconway/Parse-RecDescent-1.965001/lib/Parse/RecDescent.pm`

[18] Free Software Foundation, Inc. *GNU General Public License*. Version 2. 1991.

[19] Torvalds, Linus et al. *Linux* [program]. Version 3.3.4. 2012. Dostupné také z: `http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.3.4.tar.gz`

[20] ISO/IEC 9899:1999, *Programming Language C*. International Organization for Standardization 1999.

[21] Free Software Foundation, Inc. C Extensions. *Using the GNU Compiler Collection (GCC)* [online]. Version 4.5.0. 2008 [cit. 2012-05-23]. Dostupné z: `http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gcc/C-Extensions.html`

[22] Bonwick, Jeff. *The Slab Allocator: An Object-Caching Kernel Memory Allocator*. Sun Microsystems, 1994.

Příloha 1: Ukázkový program

Jednoduchý program využívající hybridní databázi. Obsahuje databázi jmen a pozdravů. Umožňuje přidat jméno či pozdrav a vypsat náhodný pozdrav.

```
# Hybrid database - simple example

NodeType Name     { Attribute value : String }
NodeType Greeting { Attribute value : String }


# Simple index
# Stores all nodes in array
IndexType Array : struct {
  IndexLock lock;
  size_t size, capacity;
  Node **data;
} {
  Init {
    *context = (typeof(*context)){
      .size = 0,
      .capacity = 0,
      .data = 0
    };
  }

  Destroy { tr_memory_late_free(allocator, context->data); }

  Update {
    switch (event) {
      // Put new node in index
      case CBE_NODE_CREATED:
      case CBE_NODE_LOADED: {
        trIndexLock(context);

        if (context->size == context->capacity) {
          size_t new_cap;
          void *data = tm_array_expand(H, context->data,
            sizeof(void*) * context->capacity, &new_cap);
          trMemoryInternalWrite(&context->data, data);
          trMemoryInternalWrite(&context->capacity, new_cap /
             sizeof(void*));
        }

        trMemoryInternalWrite(context->data + context->size,
           node);
        trMemoryInternalWrite(&context->size, context->size + 1);

        break;
      }

      // Remove deleted node from index
      case CBE_NODE_DELETED:
        trIndexLock(context);

        for (int i = 0; i < context->size; i++)
          if (typeUncast(context->data[i]) == node) {
```

```
51              trMemoryInternalWrite(&context->data[i],
                    context->data[context->size - 1]);
52              trMemoryInternalWrite(&context->size, context->size
                    - 1);
53
54              if (context->size * 2 < context->capacity) {
55                size_t new_cap;
56                void *data = tm_array_shrink(H, context->data,
57                  sizeof(void*) * context->capacity, &new_cap);
58                trMemoryInternalWrite(&context->data, data);
59                trMemoryInternalWrite(&context->capacity, new_cap
                      / sizeof(void*));
60              }
61
62              return true;
63            }
64          break;
65
66        // Just ignore node modification
67        case CBE_NODE_MODIFIED: break;
68      }
69    }
70
71    # Returns # of nodes in array
72    Method size : size_t { *value = trMemoryRead(context, ->size);
         }
73
74    # Returns node at index @c index
75    Method at(size_t index) : Node* {
76      Node **data = trMemoryRead(context, ->data);
77      *value = data[index];
78    }
79
80    # Returns random node from array or NULL if array is empty
81    Method get_random : Node* {
82      size_t size = trMemoryRead(context, ->size);
83      Node **data = trMemoryRead(context, ->data);
84
85      *value = size ? data[rand() % size] : 0;
86    }
87 }
88
89 DatabaseType HelloWorldDatabase (v1_0) {
90    NodeType Name
91    NodeType Greeting
92
93    Index Names for Name : Array
94    Index Greetings for Greeting : Array
95 }
96
97 Interface {
98 // We have to include database.h because Perl parser doesn't know
99 // where database.h is
100 #include "../database.h"
101 }
102
103 Implementation {
104
```

```
105 #include <stdio.h>
106
107 // simulates realloc in transactional memory
108 static void *tm_array_resize(Handle *H, void *ptr, size_t
      old_cap, size_t new_cap) {
109   void *new_ = tr_memory_alloc(H, new_cap);
110   memcpy(new_, ptr, old_cap);
111   tr_memory_free(H, ptr);
112
113   return new_;
114 }
115
116 static void *tm_array_expand(Handle *H, void *ptr, size_t size,
      size_t *new_cap) {
117   *new_cap = size * 2 + 30;
118   return tm_array_resize(H, ptr, size, *new_cap);
119 }
120
121 static void *tm_array_shrink(Handle *H, void *ptr, size_t size,
      size_t *new_cap) {
122   *new_cap = size / 2;
123   if (*new_cap < 30) *new_cap = 30;
124
125   return tm_array_resize(H, ptr, *new_cap, *new_cap);
126 }
127
128
129 static void help(const char *name) {
130   printf("Usage: %s command command_options...\n%s\n", name,
131     "  Available commands are:\n"
132     "    ag - adds greeting. Takes one argument. Argument should
          once contain\n"
133     "          %s sequence. It will by replaced by name.\n"
134     "    an - adds name. Takes one argument.\n"
135     "    p - prints random greeting with random name.\n"
136     "    h - shows this help\n"
137   );
138 }
139
140 int main(int argc, char **argv) {
141   if (argc < 2) { help(argv[0]); return 1; }
142   int ret = 0;
143
144   HelloWorldDatabase *D = dbCreate(HelloWorldDatabase,
        "hello_db", DB_CREATE);
145   if (!D) {
146     printf("db creation failed\n");
147     return 1;
148   }
149
150   HelloWorldDatabase_handle_t H[1];
151   db_handle_init(D, H);
152
153   tr_begin(H);
154   switch (argv[1][0]) {
155     case 'a':
156       if (argc != 3) goto die;
157       switch (argv[1][1]) {
```

```
158         case 'g': {
159           Greeting *h = trNodeCreate(Greeting);
160           trWrite(h, value, argv[2]);
161           break;
162         }
163
164         case 'n': {
165           Name *n = trNodeCreate(Name);
166           trWrite(n, value, argv[2]);
167           break;
168         }
169
170         default: goto die;
171       }
172       break;
173
174     case 'p': {
175       Name *name = nodeCast(Name, trIndex(Names, get_random));
176       Greeting *greeting = nodeCast(Greeting, trIndex(Greetings,
              get_random));
177
178       // We can do this, because we are the only transaction so
              we cannot fail
179       // The right way to do this would be sprintf to some buffer
180       printf(greeting ? trRead(greeting, value) : "Hello %s!",
181              name ? trRead(name, value) : "world");
182       printf("\n");
183       break;
184     }
185
186     case 'h':
187       help(argv[0]);
188       break;
189
190     default:
191       printf("Unknown command\n");
192       help(argv[0]);
193       goto die;
194   }
195
196   if (0) {
197     die:
198     tr_failed:
199     printf("Error\n");
200     ret = 1;
201     tr_abort(H);
202   } else if (!tr_commit(H, CT_ASYNC))
203     printf("Error: transaction failed, but there is no one to
            colide with!\n");
204
205   db_handle_destroy(H);
206   database_close(D);
207
208   return ret;
209 }
210
211 }
```

Příloha 2: Výsledky testů

Měření rychlosti čtení

Hodnoty v tabulce udavájí čas v milisekundách, který trvalo provedení milionu transakcí. Za proměnnou `$i` byl dosazen počet vláken. Testovací databáze byla vytvořena příkazem `create2.exe` a obsahovala milion prvků.

Příkaz: `./read.exe /tmp/test_db $i $(((1000 * 1000) / $i)) 50 1`

Počet vláken	Běh 1	Běh 2	Běh 3	Běh 4	Běh 5	Průměr
1	16160	16044	16061	16019	15996	16056
2	7926	7905	7932	7986	7939	7937
3	6219	5963	6120	6189	6021	6102
4	4866	4881	4872	4859	4904	4876
5	4997	5114	5014	5028	5067	5044
7	4870	4851	4894	4813	4908	4867
10	4819	4858	4781	4881	4779	4823
15	4804	4773	4773	4789	4773	4782
20	4763	4773	4773	4810	4771	4778

Měření rychlosti zápisu

Hodnoty v tabulce udavájí čas v milisekundách, který trvalo provedení 200 000 transakcí. Za proměnnou `$i` byl dosazen počet vláken. Testy s kolizemi a bez kolizí se liší pouze tím, zda si pracovní vlákna rozdělila databázi na části a každé pracovalo se svou částí nebo zda všechna vlákna přistupovala k celé databázi. Testovací databáze byla vytvořena příkazem `create2.exe` a obsahovala 100 000 prvků.

Bez kolizí:

Příkaz: `./write.exe /tmp/test_db $i $(((1000 * 200) / $i)) 50 1`

Počet vláken	Běh 1	Běh 2	Běh 3	Běh 4	Běh 5	Průměr
1	11966	12011	11781	11982	12104	11968
2	6234	6264	6303	6340	6393	6306
3	5252	5282	5174	5253	5285	5249
4	4980	5073	5141	5182	5079	5091
5	4751	4751	4932	4901	4742	4815
7	4655	4491	4537	4543	4571	4559
10	4455	4449	4487	4453	4412	4451
15	4514	4468	4632	4456	4474	4508
20	4794	4576	4557	4542	4471	4588

S kolizemi:

Příkaz: `./write_col.exe /tmp/test_db $i $(((1000 * 200) / $i)) 50 1`

Počet vláken	Běh 1	Běh 2	Běh 3	Běh 4	Běh 5	Průměr
1	11993	11896	12111	11996	12303	12059
2	11480	11593	12145	11833	12034	11817
3	10750	10362	10228	10902	10204	10489
4	10049	10004	10046	10118	10082	10059
5	9874	9858	10062	9949	10107	9970
7	9978	9868	10007	10010	9857	9944
10	9828	9983	9761	9915	10112	9919
15	9780	9742	9760	10256	9844	9876
20	9924	9703	9793	9766	9786	9794

Kolize:

Následující tabulka obsahuje počet kolizí vyjádřený jako promile úspěšně dokončených transakcí. Data pocházejí z běhu zápisového testu s kolizemi.

Počet vláken	Běh 1	Běh 2	Běh 3	Běh 4	Běh 5	Průměr
1	0	0	0	0	0	0
2	147	149	144	147	154	148
3	175	152	155	151	149	156
4	176	188	186	201	194	189
5	221	243	252	250	238	240
7	302	321	334	374	331	332
10	416	485	387	409	436	426
15	581	552	539	613	589	574
20	659	662	702	685	686	678