XStore

Pavel Parízek

Table of Contents

Úvod

XStore bude databázový server umožňující ukládat data v XML a podporující dotazování v XPath a přístup přes DOM rozhraní. Později se nad tím může postavit implementace jazyka XQuery. Cílem projektu bude navrhnout kostru celé aplikace a implementovat databázový storage system, transakce, indexy, reprezentaci XML dokumentů v DOM a provádění dotazů v jazyce XPath (to půjde jednoduše pomocí DOM a indexů). Implementace XQuery by se mohla dodělat jako další projekt někdy v budoucnosti - ten by stavěl na našem základu a kvůli tomu musíme dobře udělat návrh kostry celé aplikace. Dalším cílem bude i vyzkoušení odděleného ukládání a chytrých indexů, které si pamatují strom dokumentu. Uvidí se, jestli to je dost efektivní ve srovnání s jinými způsoby ukládání XML dokumentů.

Celková architektura

Server bude modulární a rozdělený do vrstev. Modulárnost umožní snadné doplňování dalších modulů (např šifrování a zálohování) a také výměnu již existujících modulů za lepší. Moduly poskytující určitou činnost by musely splňovat jisté pevně definované API použitelné ostatními částmi aplikace. Každá vrstva bude obsahovat jeden či více modulů s tim, že žádný modul by neměl zároveň patřit do více vrstev. Moduly budou jen v oddělených zdrojových souborech (a object files). Nativní XML databáze bude nejspíš udělaná jako knihovna, nad kterou se budou stavět ty další moduly.

Musíme to navrhnout a implementovat tak, aby to zvládalo provádět velké množství dotazů (transakcí, operací storage systému) v krátkém časovém rozmezí. Také by tomu neměly dělat potíže velké soubory. Mělo by to být v rozumné míře scalable a robust - to znamená, že při nárůstu počtu požadavků (a transakcí) může dojít k lineárnímu zpomalení a chyba jedné transakce nesmí ovlivnit běh ostatních.

Popis vrstev

Nejnižší vrstvou bude databázový storage system, který se bude starat o ukládání dat na disk a jejich poskytování vyšším vrstvám. Storage system nebude znát strukturu XML dokumentu. Nad ním bude vrstva pro transaction processing, která bude zajišťovat transakce a zámky nad daty. Nad transakcemi budou indexové struktury, které budou uchovávat stromovou strukturu XML dokumentů a také budou urychlovat provádění dotazů. Nejvyšší vrstvy z XML databáze, které budeme implementovat, jsou XPath processor, XML parser, rozhraní DOM a Transaction Manager. Tyto objekty budou tvořit rozhraní pro různé nadstavby a aplikace. XPath processor se bude používat pro dotazy, DOM pro modifikace dokumentů a aplikační přístup a TransactionManager pro uživatelské ovládání transakcí.

Nad naší knihovnou se může dodělat XQuery processor a vrstva spravující sessions, uživatele, role a přístupová práva. Součástí našeho projektu bude ještě síťový server, který se bude starat o příjem požadavků a odesílání výsledků.

Popis komunikace mezi vrstvami

Rozhraní pro nadstavby nad nativní XML databází se bude skládat z objektů tříd XPathProcessor, DocGroupManager a TransactionManager. Kromě nich se pro přístup bude používat i DOM - především pro modifikace a aplikační využití. Všechny rozhraní z nativní XML databáze směrem ven k serveru musí být thread-safe - to znamená, že musí být připravené na to, že je bude volat víc vláken najednou. Ty vlákna vytvoří server nebo jiná nadstavba.

Vkládání do front mezi vrstvami bude synchronní - pokud je fronta plná, tak vkládající vlákno čeká. Vybírání může být synchronní (receive čeká na zprávu) i asynchronní (receive se hned vrátí s tím, že tam zpráva není). Asynchronní receive je potřebný např pro SocketManager, když bude kontrolovat frontu s výsledky. Záznamy ve frontách budou potomci třídy Message.

Pro každou frontu mezi vrstvami bude existovat instance třídy MessageQueue, která umí ukládat instance potomka třídy Message. Prvky každé z front budou instance jiného potomka třídy Message, protože ponesou jiné informace. Ale pro fronty by mohla stačit jen třída MessageQueue.

Ty fronty zpráv se budou chovat jako pipes - na jednom konci se zapisuje, na druhém se čte. Ke každé instanci MessageQueue bude existovat ReadingPipe pro čtení a WritingPipe pro zápis. Ostatní objekty (např SocketManager) vždy dostanou správnou pipe, aby mohli na té frontě dělat jen tu jednu operaci - o té frontě vespod ale nebudou vědět nic. Takle se ty vrstvy od sebe ještě trochu víc oddělí.

Inicializace a konfigurace

Na začátku se v inicializační funkci nativní XML databáze zavolají init funkce pro všechny moduly a vyrobí se objekty. Ta inicializační funkce pro databázi se bude volat v serveru nebo v jiné nadstavbě (aplikaci). Funkce main() bude v nadstavbě nebo jiné aplikaci, ale ne v databázi (to bude knihovna). Na konci inicializace se vyrobí Thread Manager a všechny vlákna. Vlákna se rozdělí do skupin hned při vytvoření. Před vlákny se ještě vytvoří všechny globální objekty. Hlavní funkce jednotlivých vláken budou statické metody - jako parametr se jim předá objekt, kterého se týkají. Vlákna v thread poolu budou ve své hlavní funkci kontrolovat fronty požadavků a pro každý přečtený request zavolají nějaké objekty, viz kapitola o zpracování požadavků.

Konfigurační soubor bude uložen jako /etc/xstore.conf a bude platný system-wide. Nebudou žádné lokální konfiguráky pro uživatele, takže všechny instance Xstore v systému budou mít stejnou konfiguraci. Tím pádem nemá smysl pouštět víc než jednu, aby se nehádali o data, apod. Některá nastavení by také měla jít upravovat přes command-line parameters - zejména ta, která se budou měnit častěji.

Konfigurace v souboru i prostřednictvím command-line parameters se bude uchovávat v jednom globálním objektu, který při inicializaci přečte obsah toho souboru a parametry při spuštění. Ostatní objekty se tohoto správce konfigurace budou ptát na aktuální nastavení. Ten objekt se může jmenovat ConfigManager. Bude mít metodu readConfigFile(), která načte obsah konfiguračního souboru (/etc/xstore.conf) a metodu readCmdLine(), které se předají proměnné argc a argv a ona ty parametry zpracuje. Obě ty metody se budou volat v konstruktoru. Nastavení na příkazové řádce mají přednost před obsahem souboru. Metoda readConfigFile() bude počítat s tím, že parametry v konfiguračním souboru mají tvar proměnná=nějaká hodnota (bez uvozovek). V souboru musí být každá proměnná na samostatné řádce. Maximální délka řádky je 512 znaků. První výskyt znaku ’=’ od začátku řádku bude značit konec názvu proměnné. Znak ’#’ na začátku řádku označuje komentář. Metoda readCmdLine() bude počítat s tím, že parametry na command line maji tvar --proměnná="nějaká hodnota".

Paralelní zpracování požadavků

Každá skupina vrstev mezi dvěmi frontami bude mít svou vlastní skupinu vláken. Mezi každou tou skupinou vrstev budou vždy dvě fronty - jedna vstupni a druhá výstupní, bráno z pohledu vyšší vrstvy.

Skupina vláken v každé vrstvě bude spravovaná instancí třídy ThreadGroup - ta třída bude mít metody pro přidání dalšího vlákna do skupiny, apod. Kromě toho bude v systému instance třídy ThreadManager, která bude udržovat seznam všech objektů Thread a ThreadGroup a poskytovat další funkce - např tvorbu vláken, jejich rušení a změnu atributů. Synchronizovat se bude pouze přístup k ThreadManageru, protože všechny instance tříd ThreadGroup a Thread jsou přístupné jen přes něj. Maximální počet vláken bude určen konfigurační proměnnou "threadnum". Minimum bude nastaveno na 3, protože potřebujeme jedno maintenance vlákno do storage systemu, dále aspoň jedno vlákno pro zpracování požadavku (bude v thread poolu) a jedno vlákno pro network server. Zbylé vlákna budou vložena do thread poolu, kde budou čekat na požadavky. Takže pokud admin nastaví počet vláken na 8, tak v thread poolu jich bude 6.

Všechna vlákna vyrobí inicializační procedura XML databáze a serveru s pomocí ThreadManageru. Počet vláken tedy bude pevný - pokud bude moc požadavků, tak si nějaké trochu déle počkají ve frontě. Možná by bylo dobré udělat nějaký konfigurační parametr, kterým se nastaví maximální počty vláken v jednotlivých vrstvách - když to poběží na více procesorech, tak bude moci admin nechat udělat více vláken. To ale až jako případné rozšíření. Na vlákna, na která se nebude muset volat pthread_join(), by se měl zavolat pthread_detach() hned po vytvoření (přes změnu atributů).

Každý požadavek bude vykonáván právě jedním vláknem, ale různé požadavky jednoho připojeného klienta budou moci provádět různá vlákna. Vždy se použije náhodně vybrané volné vlákno v thread poolu. Počet vláken ve všech vrstvách bude pevně daný - worker threads budou postupně vybírat požadavky z fronty od network serveru a dávat tam výsledky.

Vlákno, které nemá co dělat nebo právě dokončilo zpracování jednoho požadavku, se podívá do fronty, jestli tam není další - pokud je, tak si ho vezme, a pokud ne, tak se uspí přes condvar_wait(). Až nějaký požadavek přijde, tak

fronta probudí spící vlákna pomocí condvar_signal(). Hlavní funkce vláken budou obsahovat cyklus, ve kterém se vždy budou snažit získat požadavek, vykonat ho a výsledek dát do jiné fronty.

Všechny objekty, které budou vlákna potřebovat (třeba kvůli volání metod), budou dostupné jako globální proměnné nebo se získají přes ty globálni objekty. Globálni proměnné (objekty) budou definované na jednom místě v header file (src/xstore.h), který bude muset includovat každý další soubor. V tom header file budou jen pointery na ty objekty - konkrétní instance budou vytvořené při inicializaci ve funkci xstore_init() v src/xstore.cpp. Ty globální objekty budou muset být thread-safe, protože k nim může přistupovat víc vláken najednou. Při ukončování se zase musí zavolat uklidová metoda xstore_destroy, která ty globální objekty zruší.

Network server and thread pool

Sit'ový server bude fungovat jako nadstavba nad nativní XML databází a bude využívat a dále poskytovat všechny možnosti přístupu nabízené databází. Samotný server bude rozdělen na dvě části. Jedna se bude starat o síťovou komunikaci a vkládání požadavků přečtených ze socketů do fronty a druhá část se bude starat o thread pool, ve kterém budou vlánka zpracovávající požadavky. Ty vlákna budou brát požadavky z fronty od network serveru a budou volat XPathProcessor nebo další objekty poskytované databází. Thread pool bude mít asociovánu ThreadGroup, která bude obsahovat vlákna využívající objekty od toho poolu až k práci s diskem.

Server bude muset nějak poznat, na který socket má poslat odpověď na požadavek. To by šlo udělat tak, že do fronty mezi network serverem a thread poolem by se spolu s textem dotazu vložila i nějaká identifikace toho klienta (socketu). Vlákno zpracovávající požadavek si tu identifikaci přečte a pak ji pošle zpátky spolu s odpovědí. Podle toho pak server pozná, kam tu odpověď poslat. Musí si samozřejmě někde uložit přiřazení mezi identifikacemi klienta a sockety. Budeme přitom předpokládat, že klient vždy pošle dotaz a pak si počká na odpověď, takže nebudeme řešit situaci, kdyby mohly přijít zároveň dva požadavky od jednoho klienta bez toho, aby byl aspoň jeden zpracovaný a jeho výsledek poslaný zpět klientovi.

Požadavky od klientů přijmuté po síťi musí obsahovat text dotazu v XPath nebo operaci pro DOM a odpovědi posílané zpět musí obsahovat výsledek dotazu v odpovídajícím formátu. Záznamy ve frontách pro sockety budou tedy obsahovat jen ty data, která se budou posílat přes síť klientům. Síťová vrstva k tomu jen přidá identifikaci klienta (socketu), ze kterého ten dotaz přišel, kvůli tomu, aby pak odpověď poslala na správný socket.

Network I/O

O komunikaci po síti se bude starat objekt třídy SocketManager. Ten bude uvnitř fungovat tak, že bude volat dokola poll() na všechny sockety a když se na nějakém socketu bude dát číst nebo psát, tak zavolá zaregistrovanou metodu objektu třídy Socket odpovídající tomu socketu. Pokud se na acceptujícím socketu objeví nové připojení, tak vyrobí nový objekt třídy Socket a zaregistruje všechny potřebné metody pro upozornění na možnost čtení a/nebo psaní nebo na příchod výsledku dotazu z fronty.

Vlákno v serveru bude sockety testovat pomocí poll() s nějakým timeoutem. Samozřejmostí je, že ty sockety budou non-blocking. Jedno vlákno v síťové vrstvě by mělo stačit i na poměrně velký počet současně připojených klientů. Mohlo by to jít i pomocí dvou vláken, ale pak by si musely rozdělit sockety, které by hlídaly pomocí poll() - objekty třídy Socket by ani pak nemusely být synchronizované, pokud by to rozdělení bylo fixní. V případě, že by obě vlákna mohla koukat do fronty výsledků a dávat ty údaje objektům třídy Socket bez toho rozdělení, tak by v těch objektech Socket muselo být zamykání.

V případě čtení by objekt pro socket volal read() v non-blocking módu a to, co by přečetl, by uschoval v nějakém bufferu. Až by přečetl i konec zprávy, tak by ji předal do fronty k thread poolu. Z té fronty to pak vyndá jedno vlánko, které bude zpracovávat ten požadavek. V případě připravenosti pro zápis by objekt Socket uložil flag, že může zapisovat pomocí write, až bude muset. SocketManager bude střídat volání poll() s kontrolováním fronty s výsledky požadavků od thread poolu - pokud tam nějaký bude, tak ho vezme a předá tu zprávu příslušnému objektu třídy Socket, který ho pošle, pokud bude mít nastaven flag pro možnost zápisu, nebo ten výsledek uloží do bufferu a pošle ho, až bude moci psát.

SocketManager dostane ako parametry konstruktoru pipes, přes které bude komunikovat s thread poolem. Uvnitř instance SocketManager se samozřejmě budou používat další objekty (např pro jednotlivé sockety), ale rozhraní ven bude jen přes něj a přes pipes nad frontami.

Musíme se snažit volat read() a write() pro co největší množství dat, aby se nevolal kernel moc často. Také je nutné dávat pozor na to, že vlákna přistupující na sockety mohou běžet na různých procesorech - prostě se musí dobře synchronizovat přístup na sockety různými vlákny.

Práce s pamětí

Implementace malloc() a free() (respektive new a delete) na Linuxu a dalších Unixech jsou dostatečně efektivní i pro častou alokaci a dealokaci, takže dále zmíněné věci jsou jen takový experiment, který se možná někdy použije. Jediný větší problém standardní implementace je fragmentace, pokud se alokuje příliš mnoho malých bloků paměti - to by vlastní memory manager zčásti řešil. Místo častého volání malloc() a free() (nebo new a delete) na bloky stejné velikosti je lepší si ty bloky místo uvolnění dát do nějakého seznamu a brát je odtamtud při potřebě. Taky může být někdy užitečné naalokovat si paměť do zásoby při startu serveru - zavolat malloc() nebo new pro blok každé velikosti tolikrát, kolikrát bude potřebný v budoucnosti, apod.

To vše bude dělat třída MemoryManager, která při startu naalokuje určité množství bloků všech nutných velikostí a pak je bude na požádání přidělovat a uvolňovat. Každý modul bude poskytovat funkci init_mem(), kterou zavolá hlavní vlákno při startu serveru. V té funkci se zavolá (i vícekrát) nějaká metoda třídy MemoryManager s počtem a velikostí bloků, které by bylo vhodné předalokovat. MemoryManager přialokuje další bloky v případě, že bude počet bloků nějaké velikosti hodně malý - bude k tomu sloužit nějaké vlákno s nižší prioritou, které vždy projede seznam volných bloků všech velikostí a pak zavolá yield(). MemoryManager bude bloky alokovat po větších kusech - při požadavku na alokaci 20 bloků velikosti 32 bytu rovnou naalokuje jeden blok velikosti 640 bytu, ale uvnitř si to rozdělí do položek seznamu. Při doalokování dalších bloků se možná taky bude muset přealokovat paměť pro seznamy volných bloků, pokud bude docházet - udělat ji na začátku pro každou velikost bloku 2x větší, než je potřeba.

Pro použití třídy MemoryManager v C++ si musíme napsat vlastní globální operátory new a delete (viz Thinking in C++, Vol 1, Chapter 13), které budou volat jeho metody. Pokud metoda třídy MemAllocator vrátí chybu (nedostatek paměti), tak musí operator new vyvolat exception pomocí throw bad_alloc().

Je vhodné používat "zero-copy" algoritmy z důvodů efektivity, ale musí se upřednostňovat bezpečnost a spolehlivost kódu před maximální efektivitou. Ta výhoda "zero-copy" algoritmu se projeví hlavně při práci s obsahem XML dokumentů. Nejlepší by asi bylo každý dokument reprezentovat nějakým objektem a mezi různými metodami předávat jen ten pointer. Na některých místech se ale budou dělat zdánlivě zbytečné kopie dat - například při konstrukci objektu třídy DocData. To je nutné za účelem větší separace modulů a vrstev od sebe. Konkrétně objekt třídy DocData se bude tvořit ve storage systému při načtení dat z disku (z cache) nebo v XML parseru před uložením dat do databáze. Při čtení dat z disku je kopie nutná, aby pozdější změny v kopii neovlivnily data na disku (v cache) a v druhém případě se bude kopie tvořit právě kvůli té izolaci vrstev od sebe.

Character sets

Data se budou interně ukládat v kódování UTF-8, protože použití wchar_t (pro iso10646) je plácání místem. Každý řetězec bude uložen jako pole hodnot typu char, kde nějaké textové znaky v UTF-8 mohou být složené z více hodnot typu char.

Vstupní XML dokumenty mohou být v libovolném kódování, takže musíme umět pracovat se všemi podporovanými (iso-8859-x, windows-1250, apod). Během parsování dokumentu se text převede do našeho interního kódování a při posílání dat klientům se zase převede do kódování, které požaduje klient. Ke každému dokumentu by se mohlo uložit jeho defaultní výstupní kódování. Během převádění se musí dát pozor na binární data - viz kapitola o Ukládání XML dokumentů. Třída StringConverter funguje jako wrapper pro iconv(). Pokud iconv()

nebo fungovat, tak budeme muset použít knihovnu ICU od IBM. Knihovna ICU umí převody mezi různými kódováními a další věci související s Unicode, i18n a l10n. Licence k ICU dovoluje použití knihovny bez omezení, pokud se v dokumentaci k aplikaci objevi copyright a permission notice od IBM (viz dokumentace k ICU).

Error handling

Pro zpracování errorů (okrajových, chybových a nestandardnich stavů) budeme používat exceptions místo návratových hodnot všude, kde to půjde. Exceptions se budou vyvolávat i při chybách v syscallech (funkcích dostupných z glibc). V obou případech (exceptions i return values) se musí promyslet, co se má stát, když dojde k chybě u socketu nebo třeba v query processoru. Jde o to, kam až se má ta výjimka (chyba) propagovat a musí se nějak odlišit fatální chyby od ostatních. Mezi okrajové stavy patří například špatná syntax dotazu v XPath, chyba I/O nebo vkládání non well-formed XML dokumentu (to se otestuje při tvorbě stromu).

Vyjimky, které by mohly způsobit abort transakce, se budou posílat vyšším vrstvám, které se rozhodnou, jak zareagují. Mohou říct Transaction Manageru, aby provedl abort, nebo udělají něco jiného - například dodají metodě jiné parametry.

Coding style, Documentation, Testing

Pro třídy, které by uživatel mohl chtít změnit nebo u kterých se může měnit implementace a definice privátních metod, se udělá samostatný interface (abstraktní třída) s abstraktními virtuálními (a některými obyčejnými) metodami a skutečná implementační třída pak bude potomkem toho interface. Uživatelé té třídy se budou moci spolehnout jen na metody uvedené v tom interface - žádné jiné ani neuvidí. Pracovat se tedy bude jen s proměnnou typu "pointer na odpovídající interface". Přiklad tvorby objektu: ConfigManager* confMngr = new ConfigManagerImpl();.

Tvorba objektů z nějaké implementační třídy odpovídající danému interfacu se může provádět buď klasicky nebo přes class factory. Class factory je reprezentovaná třídou ObjectFactory, která bude mít samé statické metody - jednu pro každý interface. To znamená, že nebude žádná instance té ObjectFactory. Každá metoda vyrobí instanci odpovídajícího interface. Jen ta metoda bude přesně znát typ té instance. Ta factory se použije jen pro třídy, které jsou potomky samostatného interface, protože pro jiné třídy to nemá smysl.

Celé XStore bude napsané jako aplikace pro Linux a přenositelnost na Windows a další platformy se řešit nebude. Také se nebudou používat autotools, protože v něčem dost omezují. Připadně by se použil jen autoconf kvůli přenositelnosti zdrojových kódů na další unixové systémy. Spíš se pak ale udělá Makefile pro každou podporovanou platformu. Při psaní se musíme snažit dodržovat a využívat standardy, takže se kód bude kompilovat s makrem XOPEN_SOURCE=500 a připadně i GNU_SOURCE. Kód se musí psát s cílem minimalizovat počet warningu při kompilaci. Měli bychom využívat jen ty prvky C++ a STL, co jsou v standardu (draft z roku 97 je volně dostupný na webu). Důležité je také použití makra THREAD_SAFE kvůli lepší podpoře vláken. Kromě přenositelnosti na úrovni zdrojáků je také problém s přenositelností datových souborů mezi různými architekturami (i386, sparc, apod). Přikladem je endianita celočíselných typů - datové soubory vytvořené na i386 nebudou správně interpretovatelné na sparcu, protože do popisných souborů se natvrdo zapisují proměnné typu unsigned int nebo uint64_t. Taky mohou být potíže s interpretací utf-8. Řešením jsou nějaké konverzní funkce nebo makra, která se mohou dodělat později v případě potřeby.

Kód musí být co nejkorektnější a nejúplnější, protože to umožní odhalit hodně chyb už při kompilaci. To znamená, že se musí definovat i copy-constructor a operator= (oba případně private pro nekopírovatelné třídy), pak je také vhodné používat const metody a parametry metod, const iteratory, apod. Také by se měly využívat věci poskytované STL, protože jsou poměrně standardizované a napsané dost efektivně. A ušetří dost práce s vlastní implementací. Příkladem jsou containers a smart pointers (auto_ptr).

Asi by to mělo být určený pro enterprise oblast, takže se musí použít 64-bitový čísla pro unikátní identifikátory (aspoň pro některé), syscally jako open64 a také musíme předpokládat prakticky neomezený velikosti souborů, apod.

Dokumentace musí být poměrně obsáhlá. Měla by se psát v javadoc formátu pro Doxygen proto, aby měla nějaký formát. K dokumentaci patří i UML schémata vyrobené v Dia nebo jiném sofistikovanějším nástroji pro UML (např ArgoUML). Na začátku každého souboru by mělo být jméno autora (vlastníka) souboru.

Měly by se používatt i nějaké test suites (pravděpodobně vlastní výroby) a pravidelně je pouštět (viz src/test). Do komentářů k funkcím se mohou psát preconditions a postconditions a aspoň v debug modu mohou být začátku a na konci těla funkce asserty. Spouštění testů by asi bylo v Makefiles pod cílem 'test' a muselo by se to provádět pravidelně - třeba po každé kompilaci. Konkrétní implementace test suites se ještě dořeší - buď bude v každé třídě metoda testIt(), která se bude překladat jen v DEBUG módu a bude se nějak volat nebo budou nějaké samostatné testovací prográmky pro každé možné použití třídy. Použití testovacích prográmků je asi lepší. Můžeme zkusit i něco jako code-review aspoň ve dvojicích.

Testování správné funkce zámků a transakcí se dá dělat tak, že se do databáze vloží nějaká data, pak se na ně pustí posloupnost operací v několika transakcích a na konci se zkontroluje, jestli jsou výsledná data správná - jestli odpovídají nějakému sériovému běhu transakcí. Ty transakce budou vykonávané více vlákny, kde každé vlákno bude mít přiřazenou jistou pevnou posloupnost transakcí ke zpracování. Průběh testu by se měl logovat a test by se měl vždy pouštět několikrát za sebou, protože jeho výsledek taky závisí na tom, jak bude operační systém plánovat vlákna, apod.

Databázový storage system

Vlastnosti storage systemu

Na disk se bude ukládat obsah elementů, hodnoty atributů, mapování uložených dat na odpovídající Id, informace o transakcích, žurnál a indexové struktury. Storage systém musí obsah XML dokumentů a indexové struktury ukládat tak, aby k nim byl co nejrychlejší přístup s rozumnými prostorovými nároky. K datům se bude přistupovat přes různá Id, která bude generovat storage system při přidání prvku. Při požadavcích na storage systém se ale musí předat i DocId, aby se vědělo, kterého dokumentu se požadavek týká.

Přidělování Id ve storage systemu bude fungovat tak, že si někde bude pamatovat nejvyšší použité Id pro každý typ a při požadavku na nové přidělí Id o jedna vyšší. Nebudou se opakovaně používat Id smazaných bloků dat a dokumentů, protože hledání volného Id by bylo časově příliš náročné. Jednotlivé typy pro Id mají dostatečný rozsah (32 nebo 64 bitů) na to, aby se mohla přidělovat jen v rostoucí sekvenci bez hrozby vyčerpání. Všechna Id používaná ve storage systemu se budou přidělovat nezávisle na struktuře dokumentů (na structure Ids - napr NID). Pokud by mezi storage Ids a structure Ids byla nějaká závislost, tak by se ty storage Ids musely přepočítávat při každém updatu dokumentu a to by storage system asi zpomalovalo.

Storage systém bude objekty z XML dokumentu ukládat jako kousky dat a nebude se starat o jejich obsah. Storage system si taky nebude pamatovat strom dat v XML dokumentu. Každý prvek XML dokumentu bude asociován s nějakým DocDataId a každý blok dat z indexů bude asociován s nějakým IndexDataId nebo DocIndexDataId. Storage system si bude v hash table pamatovat přiřazení různých Id k datům.

Při čtení dat z cache nebo z disku se vyšším vrstvám (nad Storage Managerem) musí dát úplná kopie (deep copy) těch dat, ne jen pointer. Jinak by totiž ty vyšší vrstvy mohly ty data blokovat a samovolně je měnit. To samé se bude aplikovat i při pohybu dat opačným směrem - do cache se uloží kopie toho, co přišlo v požadavku. Tohle chování bude zajišťovat StorageManager ve spolupráci s CacheManager pomocí tříd DocData, IndexData a DocIndexData. Data z dokumentů předaná storage systému musí být v kódování UTF-8, takže vstup v jiném kódování se musí převést někde ve vyšších vrstvách do UTF-8 pomocí StringConverteru (wrapper pro iconv()).

Storage system bude generovat DocDataId pro všechny prvky XML dokumentů, které mohou být dotazovány přes XPath - elementy, atributy, processing instructions, text, apod. Při vkládání nového bloku dat se jako hodnota DocDataId bude posílat 0, což bude označovat nový kus dat. Skutečné DocDataId přiřazené datům se vrátí z té metody jako return value. V případě vkládání bloku s jiným DocDataId se to bude brát jako update a stará hodnota se přepíše novou. Pokud předaná hodnota nebude rovna nějakému existujícímu DocDataId nebo hodnotě 0, tak se

příslušná metoda vrátí s chybou. Hodnoty DocDataId budou unikátní přes všechny bloky dat v jednom dokumentu v databázi. DocDataId se může vyrobit třeba jako kombinace typu objektu a umístění (bude to prostě klíč do hash table).

Globální indexy se budou ukládat po blocích různé velikosti identifikovaných pomocí IndexDataId. Pokud vyšší vrstvy předají blok s IndexDataId rovným konstantě 0, tak se vezme jako nový a vrátí se jeho skutečné (právě přidělené) IndexDataId. Pokud vyšší vrstvy budou storage systemu dávat blok s existujícím IndexDataId, tak se provede update - stará hodnota se nahradí novou. Pokud vyšší vrstva nepředá existující IndexDataId ani hodnotu 0, tak se ohlásí chyba. Analogicky pro dokumentové indexy a DocIndexDataId. V souborech s indexy budou vznikat prázdná místa, takže občas bude nutné soubor reorganizovat (setřást) - stejně jako pro dokumenty.

Vyšší vrstvy (nad transakcemi) si musí hlídat, aby se operacemi, které požadují po storage systému, nějak nepoškodila stromová struktura dokumentu. Příkladem je smazání elementu, který má nějaké potomky nebo update elementu (přidání nebo ubrání potomka). Transakce takovou operaci zablokují jen v případě kolize zámků, ale strukturu stromu hlídat nemohou. V podobných případech by se v souborech mohly objevit "orphaned elements", které by už v dokumentu fakticky nebyly, ale storage system by je pořád měl uložené, protože by mu nikdo nedal pokyn k jejich smazání.

Reprezentace XML dat, indexů a skupin dokumentů

Storage systém bude data pro vrcholy stromu skupin a dokumenty ukládat do adresářů a souborů. Uzly stromu skupin budou odpovídat adresářům. V každém adresáři bude kromě souborů s dokumenty a podadresáři ještě soubor s informacemi o skupině, která je reprezentována tím adresářem. V tom souboru s informacemi o skupině bude seznam podskupin a dokumentů ve skupině včetně těch informací o dokumentech, které musí být persistentní - například přiřazení jména dokumentu na DocId. Pro každý dokument bude tedy existovat soubor s jeho obsahem a soubor s popisem struktury souboru s obsahem. Stejné to bude pro jednotlivé dokumentové indexy. Všechny globální indexy nejspíš budou v samostatném adresáři s rezervovaným názvem, protože nebudou svázané s žádnou skupinou. Vždy při založení nového dokumentu nebo skupiny se pošle této vrstvě zpráva a ta vyrobí adresář nebo soubor. Seznam skupin a dokumentů si taky bude držet StorageManager. To, že je skupina dokumentů reprezentovaná adresářem, bude vědět jen StorageManager. Vyšší vrstvy budou používat abstrakci pomocí skupin a jejich GroupId. StorageManager si taky bude pamatovat mapování GroupId na jména adresářů a DocId na jména dokumentů. CacheManager bude vždy dostávat absolutní cesty k souboru, protože ten už zas o žádné struktuře skupin neví.

Adresáře reprezentující skupiny nejvyšší úrovně budou v adresáři "root", který je podadresář jednoho adresáře vyhrazeného pro ukládání dat (označíme data_dir). Ten vyhrazený adresář se bude určovat konfigurační proměnnou "datadir". Je to takový interní kořen stromu skupin. Kromě adresáře se skupinami nejvyšší úrovně v něm ještě bude adresář pro globální indexy, který bude mít název např "global_indexes_". V data_dir bude soubor "maxids.dsc" obsahující maximální přidělené DocId, IndexId, DocIndexId a GroupId.

Dokument se určí cestou ve stromu skupin nebo pomocí DocId. DocId bude vygenerováno při založení dokumentu nebo adresáře. Dotazy v XPath budou dokumenty referencovat pomocí cesty ve stromu skupin, ale indexy nebo XPath processor bude muset zavolat DocGroupManager kvůli převodu cesty na DocId pomocí nějaké tabulky. V požadavcích na storage systém a transaction manager musí být jen DocId z důvodů efektivity (práce s integerem je rychlejší než práce se stringem).

Textový obsah elementů, komentáře, processing instructions, atributy a CDATA z dokumentu bude storage system ukládat do souboru nezávisle na sobě. Bude si pamatovat přiřazení mezi DocDataId a těmi kousky dat a pomocí unikátního DocDataId bude ty data rozeznávat. Samostatně se budou ukládat všechny atributy jednoho elementu (vždy společně) a textový obsah elementu. Textový obsah elementu je text, který může být mezi otevíracím a uzavíracím tagem mimo všechny vnořené elementy. Také musíme vyřešit případ, kdy element bude obsahovat střídavě text a vnořený element. To se dá vyřešit vložením nějakých entit typu "&#elem;", aby bylo poznat, kam patří ty vnořené elementy. XPath se na hodnoty elementů stejně nemůže ptát - jen se element vrátí s celým jeho

obsahem. Další způsob řešení tohoto problému je ukládání těch kusů textu mezi elementy samostatně. XPath se možná stejně umí zeptat jen na každou část textu v mixed contentu samostatně.

Z důvodů efektivity je nejspíš dostatečné to, že storage systém bude ukládat elementy patřící do jednoho dokumentu do stejného bloku (nebo do sousedních), protože s každým požadavkem na disk přijde i Id dokumentu (DocId). Při uložení elementů bez vazeb mezi uzly stromu musí pro každý dokument existovat tabulka přiřazení DocDataId na offsety v souboru s dokumentem.

Při smazání nějakých dat ze souboru pro dokument nebo z bloku pro index budou v souborech vznikat díry, takže občas bude muset maintenance thread provést reorganizaci souboru ve formě setřesení. Díry budou vznikat i při updatu dat asociovaných s jistým DocDataId v případě, kdy nová hodnota bude větší než původní. Storage system si bude pro každý datový soubor udržovat seznam a velikost děr, aby tam mohl dávat úplně jiné menší bloky dat. Reorganizace v maintenance thread se bude provádět v případě, že počet děr bude příliš velký - to znamená, pokud počet přesáhne 75 % z maxima. Při reorganizaci se celý soubor setřese a zruší se všechny díry po smazaných blocích. To je poměrně náročná operace, ale zase se nebude provádět tak často, protože hodně děr se zaplní při insertech během normálního provozu databáze. Seznam děr bude seřazený podle offsetů těch děr, aby se s ním lépe pracovalo. Po smazání bloku se zkontroluje, jestli sousední díry nejsou hned vedle a pokud ano, tak se sloučí do jedné. Reorganizace se taky bude provádět hned po vložení dalšího unused bloku v případě, že jejich seznam bude v tom okamžiku plný.

Každý otevřený soubor s dokumentem nebo indexem bude odpovídat instanci třídy DataFile. Ta třída by asi měla být template parametrizovaná přes Id souboru (DocId, IndexId, DocIndexId), přes Id položky souboru (DocDataId, IndexDataId, DocIndexDataId) a přes příslušnou DataItem (kvůli vkládání a čtení). Parametrizace umožní využít jednu třídu pro všechny typy souborů. V instanci třídy DataFile budou uloženy údaje o souboru - Id, velikost, maximální přidělené Id prvku, seznam děr po smazaných datech, apod. StorageManager bude pomocí těchto objektů generovat Id pro nové prvky dokumentu a také provádět operace se soubory - vkládání, čtení a mazání. Také v těch objektech bude hash table mapující Id položek na offsety do soubory pro CacheManager. Z toho vyplývá, že CacheManager budou nejspíš volat tyto objekty a ne StorageManager.

Ty údaje o souboru s obsahem (velikost, seznam děr, mapovací hash table) budou uloženy v samostatném souboru s příponou "dsc". Soubor s obsahem bude mít příponu "dat". StorageManager tedy třídám odvozeným z DataFile předá jméno dokumentu (to znamená plnou cestu bez přípony) a ty k němu přidají příponu podle toho, jaký soubor zrovna budou chtít číst nebo měnit. Při každé změně dat se bude muset upravit i ten soubor s popisem - to bude znamenat jisté zpomalení. Jména souborů pro indexy spojené s dokumentem se budou tvořit tak, že za filename se připojí označení indexu a až za něj se bude dávat přípona - příklad: mydoc_idx1.dsc.

Soubory s popisem struktury souborů s obsahem dokumentu budou mít pevně daný formát popsaný na následujícím řádcích. Na offsetu 0 bude uint64_t s velikostí datového souboru. Dál bude maxDataId určující maximální přiřazené Id nějakému bloku dat. Potom bude skutečná velikost seznamu děr (nepoužitých bloků) a za ní rezervované místo pro ten seznam. Rezervované místo bude mít velikost 6144 B, což odpovída 512 položkám seznamu. Každá položka má 12 B - skládá se z 64-bitového offsetu a 32-bitové velikosti. Za prostorem pro seznam děr bude uložena velikost hashovací tabulky s mapováním DataId na offsety do datového souboru a pak bude následovat ta tabulka. Mapovací tabulka je úplně na konci popisného souboru, protože má proměnnou velikost. Každá položka mapovací tabulky obsahuje DataId, offset a velikost bloku dat. Výše popsaná struktura popisného souboru má tu výhodu, že se při updatu dat v dokumentu nebude muset provádět přílíš mnoho přístupů k tomuto souboru na disku. Především je důležitá fixní velikost seznamu děr - kdyby byla proměnná, tak by se často musela posouvat mapovací tabulka a to by updaty dost brzdilo.

Pro každý dokument bude existovat několik indexů. Některé budou vázané přímo na konkrétní dokument a ostatní budou obsahovat informace týkající se více dokumentů - to znamená, že budou tak trochu globální. Při tvorbě nového dokumentu se rovnou připraví indexy svázané s dokumentem, ale budou samozřejmě prázdné. Indexy pro dokument se budou unikátně identifikovat pomocí DocIndexId, které vznikne jako kombinace DocId a typu indexu a globální indexy se budou identifikovat pomocí IndexId. Pro bloky indexových dat se budou používat identifikátory DocIndexDataId a IndexDataId. Hodnoty typu DocIndexDataId budou muset být unikátní v rámci

jednoho indexu pro dokument - bude je určovat storage system, stejně jako v případě IndexDataId. Mechanismus práce bude stejný jako v případě DocDataId - hlavně co se týče vkládání nových bloků. Hodnoty DocIndexId určí storage systém při vytváření nového dokumentu tak, že jeden parametr metody createDocument bude počet indexů a on vytvoří tolik souborů pro indexy a jako DocIndexId jim přiřadí DocId zvětšené o pořadí indexu v té skupině spojené s dokumentem. Hodnoty pro IndexId se budou určovat tak, že vyšší vrstvy zavolají metodu objektu DocGroupManager, která ten globální index vytvoří a vrátí k němu IndexId.

Kromě datových nebo indexových souborů bude v každém adresáři existovat taky soubor obsahující informace o příslušné skupině. Ten soubor bude mít název odpovídající jménu skupinu a příponu "dsc". Bude v něm seznam dokumentů a podskupin příslušné skupiny a mapování jmen prvků skupiny na jejich Id. K těmto popisným souborům se bude přistupovat přes instance třídy GroupFile. V souboru s popisem skupiny bude nejprve seznam podskupin a pak až seznam dokumentů, protože dokumenty se nejspíš budou přidávat a mazat častěji než podskupiny.

Mezi důležité operace patří také vytváření a mazání adresářů a souborů na disku. Vytvářet adresáře a soubory bude CacheManager v metodě openFile - pokud nebude otevíraný soubor existovat, tak bude vytvořen a zároveň se vytvoří všechny adresáře v jeho cestě, které zatím neexistují. Mazání souboru bude také provádět CacheManager v samostatné metodě deleteFile, protože může mít části daného souboru v cache, apod. To se hodí i proto, že všechny vrstvy nad CacheManagerem pracují s logickými jmény souborů a až CacheManager před ně dává cestu do adresáře, kde má xstore všechny svoje soubory. Ten adresář se bude číst z konfigurace třeba z proměnné data_dir. Ty vyšší vrstvy vidí adresářů a souborů tak, že mají root directory v té data_dir. Pokud bude chtít StorageManager něco provést se souborem /root/obchod/potraviny/cenik.xml, tak zavolá CacheManager a předá mu tu cestu. Ten se podívá na hodnotu data_dir (bude třeba /var/lib/xstore) a udělá systémovou cestu /var/lib/xstore/root/obchod/potraviny/cenik.xml, kterou bude předávat systémovým funkcím pro práci s filesystémem. Adresáře bude vytvářet a mazat také CacheManager a při mazání smaže i všechny soubory v něm. Adresář se stejně bude moci smazat, až když bude prázdný (bez dokumentů).

Objekty v databázovém storage systému

StorageManager bude mít samostatnou metodu pro každou podporovanou akci. Bude obsahovat různé hash tables pro klíče DocId, DocIndexId a IndexId Fungovat to může tak, že ty hash table podle DocId vrátí odkaz na odpovídající instanci DocFile a ta bude obsahovat hash table pro ten konkrétní soubor. Hash tables v DocFile budou vracet offsety do souboru, které se pak předají CacheManageru. Pro dokumentové a globální indexy to bude fungovat stejně. StorageManager bude v hash table mít jen mapování pro dokumenty a indexy, které se používají - t.j. pro dokumenty, pro které bylo zavoláno getDocId() nebo createDocument(), ale ještě nebylo zavoláno releaseDocId(). Volání metody releaseDocId() bude storage systemu říkat, že práce s příslušným dokumentem prozatím skončila. Pro indexy to bude opět fungovat analogicky. Pro hash tables můžeme použít hash_map z STL (je to sice SGI extension, ale podporovaná v libstdc++ a nejen v ní).

StorageManager bude také umět převést cestu ke skupině nebo dokumentu (nebo jméno globálního indexu) na DocId nebo GroupId (nebo IndexId) a také bude převádět všechny ty Id na absolutní jména souborů nebo adresářů pro filesystém, kde jména souborů budou bez přípony - ty už jsou v kompetenci třídy DataFile. Absolutní jména souborů a adresářů budou nejspíš odpovídat logickým cestám a názvům od vyšších vrstev nebo uživatelů. Nebude tedy možné mít v jedné skupině dva dokumenty nebo podskupiny stejného jména.

Ve storage systému musí být ještě jedno vlákno kromě těch, co zpracovávají požadavky klientů, které bude sloužit jako maintenance thread. Ve chvílí, kdy všechny worker threads čekají na požadavek od klientů, tak to vlákno ve storage systému může provádět nějaké maintenance operace (setřásání souborů, apod). To také znamená, že nekteré operace s daty ve storage systému se budou muset zamykat pomocí mutexů.

Ve storage systému budeme používat 64-bitové verze synchronních I/O operací (open64, read, write) spolu s mapováním souborů do paměti. Práce v paměti je rychlejší, takže se nebudou vlákna blokovat na syscallech. Použití non-blocking file descriptorů a poll() pro přístup na disk nemá moc smysl, protože by vlákna stejně

nedělala nic jiného než čekala na splnění operace. Ten samý důvod mluví proti použití AIO - je to složitější na implementaci a v našem případě to nepřináší žádný užitek.

Přístupy na disk a cache

Je pravděpodobné, že při běhu serveru bude velké množství dat v operační paměti v cache - například indexy a často přistupované dokumenty. Pracovat se s nimi bude stejně jako s daty na disku, takže by bylo dobré mít nějaké společné rozhraní pro přístup k datům. Práci s cache a hlavně její správu bude zajišťovat Cache Manager. Všechny požadavky na disk pujdou přes něj, aby si mohl updatovat obsah cache. Zápisy na disk se ale vždycky budou propagovat ještě dolu, aby se natvrdo zapsaly na disk - to je nutné pro zachování persistence.

CacheManager si v paměti si bude držet bloky dat (části dokumentů), na které se nejčastěji přistupuje. Účelem je minimalizovat přístupy na disk třeba i za cenu většího kopírování a přesunů dat v paměti. Dlouho nepoužívaná data se z cache budou vyhazovat. Data v cache budou organizovaná do bloků - při čtení z disku se vždy načte celý blok. Bloky dat nejsou to samé co dokumenty - dokument může mít teoreticky libovolnou velikost, ale blok bude mít pevnou velikost odpovídající alokační jednotce filesystemu - nejspíš 4 KiB. Jeden blok bude obvykle obsahovat data z jednoho dokumentu, takže při čtení prvního kusu dat se ten blok vloží do cache a další přístupy k datům z dokumentu už budou rychlejší. Data, která bude CacheManager vracet, mohou být klidně z různých stránek, ale na disk to bude ukládat jen po těch blocích. Např bude muset zapsat celý blok, i když se v něm změní jen 2 byty. Pokud bude plná cache, tak se použije nějaký algoritmus pro vyhazování (LRU, LFU, apod). Při updatování nějakého bloku dat se změny okamžitě zapíší na disk.

StorageManager bude mít mapování DocDataId (IndexDataId, DocIndexDataId) na offset do souboru a CacheManager bude mít mapování pro některé ty offsety do paměti vyhrazené pro cache. Místo DocId (IndexId, DocIndexId) bude CacheManager pracovat se jmény souborů. Metody v Cache Manageru se podívají, jestli je blok v cache (jestli offset patří do nějakého intervalu) - pokud tam bude, tak vrátí kopii, jinak ho přečtou z disku a rovnou taky uloží do cache. Všechny bloky v cache budou ve spojáku a vyhazovat se bude vždy ten, který je na konci seznamu, pokud bude paměť pro cache plná. Ten starý blok vyhodí ten worker thread, který tam vkládá nový blok. Také by se mohlo udělat to, že maintenance thread ve storage systému bude sám vyhazovat moc staré bloky v cache, aby tam vždy bylo nějaké volné místo (třeba 10 procent).

Každý blok v cache bude reprezentován objektem (třídy CachedBlock) ve spojovém seznamu, který bude upravován při každém přístupu do cache. Objekt reprezentující blok, do kterého se přistupovalo, se přesune na začátek seznamu. Pro každý blok si ten objekt bude pamatovat jméno souboru a interval offsetů z daného souboru, který je v cache v paměti. Daný offset se bude testovat na náležení do některého z intervalů reprezentovaných pomocí bloků v cache. Při hledání bloku, který reprezentuje daný offset v souboru, se vždy bude muset projít ten seznam od začátku. Po nalezení bloku se provede přislušná operace s daty a protože si Cache Manager bude pamatovat iterátor, který ukazuje na prvek seznamu, tak ho bude moci jednoduše přesunout na začátek seznamu - nebude muset seznam procházet znovu. Místo seznamu (double linked list z STL) by se mohl použít i B-tree. Obě datové struktury vyjdou nejspíš přibližně nastejno (B-tree maličko rychlejší), protože nejvíc času se stejně stráví voláním kernelu kvůli diskovým operacím. Asi bych zvolil tu datovou strukturu, se kterou se bude snadněji pracovat.

CacheManager nebude provádět žádné upřednostňování indexových bloků, protože by to bylo implementačně složité a hlavně také zbytečné. Pokud se indexový blok dostane až na konec seznamu, tak už stejně asi není používaný a může se vyhodit.

Důležité ale je, aby cache manager bloky dat, ke kterým vrátil pointery v readData(), hned někam nepřesunul nebo dokonce nevyhodil z cache. Blok, na který ukazuje vrácený pointer, musí zůstat v paměti na stejném místě minimálně do doby, než nějaká vyšší vrstva (např DataFile) udělá kopii těch dat. Proto se uvnitř metody readData ten blok zamkne v cache a odemkne se, až stejná vyšší vrstva zavolá metodu unlockData. Potom už si cache manager s tím blokem může dělat co chce. Důležité je také zamykat cached blok před zápisem jeho obsahu na disk a pak ho odemknout, proto aby ho jiné vlákno během nepřepsalo. Jinak by se ale před voláním jakékoliv

funkce pracující s diskem mělo odemknout co nejvíc synchronizačních zámků řídících přístup k různým datovým strukturám, aby jiná vlákna nemusela čekat, až to aktuální dokončí tu práci s diskem.

Cache bude implementována přes načítání bloků do paměti pomocí read() a Direct IO. CacheManager bude pro cache používat paměť, jejíž velikost v kilobytech specifikuje admin systému pomocí konfiguračního parametrů "cachesize". Admin by měl určit takovou velikost cache, aby odpovídající rámce ve fyzické paměti nebyly moc často swapovány, protože žádné virtualní stránky se nebudou zamykat. Pokud na počítači poběží více aplikací, tak se nejspíš budou "přetahovat" o fyzickou paměť a výkon systému bude klesat. Minimální doporučovaná velikost cache je 16 MB. Paměť pro cache se nejspíš nebude jeden souvislý velký blok, protože to vyřeší několik problémů, které by způsobila cache jako jeden souvislý blok. Velikost jednoho cached bloku bude 4kB - read bude data číst po 4kB blocích. Pozdější implementace by mohla využívat mapování souboru do paměti přes mmap(). Použití mmap() je možná efektivnější, protože kernel si interně mapuje všechny otevřené soubory do paměti. Ušetřily by se dvě kopírování dat - jedno při načtení a druhé při zápisu dirty dat z cache na disk. Přístup k spojovému seznamu bloků musí být synchronizován nějakým zámkem. Základní implementace cache manageru bude používat read-write lock pro každý přístup. Časem se třeba vymyslí něco lepšího - např. datová struktura, která je thread-safe sama o sobě nebo se bude spoják zamykat v userlandu přes test_and_set.

Často může nastat situace, kdy je požadovaný kus dat přesně na rozhraní dvou cached bloků, které spolu ale v cache nesousedí. Protože cache asi nebude v jednom souvislém bloku paměti, tak stačí někde naalokovat místo pro všechny cached bloky, kterých se to týká (nebo jen pro ty, které se budou muset přesunout za největší z nich) a do toho nového prostoru nakopírovat obsah těch bloků. Na jejich původním místě by se pak smazali a použitá pamět by se uvolnila. Takhle zůstane celková paměť použitá cache managerem konstantní a navíc to je docela rychlý a hezký algoritmus - takový přímočarý a jednoduchý. Ty přesunuté bloky by se rovnou mohly sloučit do jednoho většího, kterému by se přiřadil timestamp rovný aktuálnímu času. Při nutnosti vyhodit nějaký blok z cache kvůli uvolnění místa by stačilo vyhodit část nějakého velkého a jeho zbytek tam nechat, podle toho, kolik volného místa by bylo potřeba. Požadované bloky dat budou většinou malé, takže je výše popsaný algoritmus asi docela efektivní. Najítí všech relevantních bloků v cache je O(N) a ostatní operace (přesun cached blocks na nové místo, apod) se zrychlit stejně nedají. Pokud nebudou všechny potřebné bloky v cache, tak se zbytek načte ze souboru.

V případě zápisu stačí v O(N) najít cached bloky, které se musí pozměnit a nová data do nich postupně zapsat. Není nutné cached bloky nějak přesouvat nebo kopírovat, protože nový data se nemusí zapisovat najednou, ale kousek do jednoho cached blocku, pak další kousek do dalšího cached blocku, atd. Nová data se stejně musí zapsat i na disk a to zabere nejvíc času.

Velikost bloků dat požadovaných po cache manageru může být obecně neomezená, takže musíme řešit i šílené případy, kdy někdo chce blok o velikosti 1 MB, apod. Hlavně to ale musí dobře fungovat pro malé bloky, protože s těmi se bude pracovat mnohem častěji než s extrémně velkými. Navíc pokud by někdo chtěl hodně velký blok dat (větší třeba než 512 kB) a cache by byla plná, tak by se pro něj rovnou naalokovalo místo někde stranou a pak v unlockData by se hned smazalo - je to rychlejší, než kvůli tomu vyhazovat spoustu bloků z cache nebo je různě přesouvat.

CacheManager si také bude muset ukládat tabulku přiřazení filename na file descriptor, protože tak zjistí, které soubory už jsou otevřené (existuje přiřazení) a taky bude bude moci v metodě closeFile najít file descriptor pro dané file name a zavolat na něj close().

V případě, že budeme chtít obejít cache pro filesystem v kernelu, tak je možné použít Direct IO. To znamená, že volání open() se jako jeden z parametrů předá flag O_DIRECT. Přistup k takto otevřenému souboru pak nebude využívat cache v kernelu, ale bude pracovat přímo s diskem, takže to bude efektivnější. V naší vlastní cache můžeme cachovat bloky, které uznáme za vhodné, bez toho, aby se nám do toho pletl kernel. Třeba JournalManager bude potřebovat okamžité zapisování na disk kvůli zachování dat v případě výpadku - po návratu z funkce write() musí být data na disku.

Thread-safe datové struktury a kód

Psát vlastní thread-safe wrappery pro kontejnery z STL podle mě nemá moc velký smysl a ani to možná nejde kvůli těm parametrům šablon. Jednodušší přistup bude ten, že každý ve svém kódu bude dávat pozor na kritické sekce v případě potřeby použije zámek nebo condition variable, nejlépe ve formě objektů z src/core/synch. Všechny důležité globální objekty musí být vytvořené jako thread-safe. Při přístupu k kontejnerům z STL je nutné synchronizovat i iterátory, aby dvě vlákna nemohla využít ten samý iterátor ve stejnou dobu pro přístup k datové struktuře, která už bude thread-safe.

Možná si ale budeme muset napsat i nějaké vlastní datové struktury a u nich by se už uplatnila interní synchronizace. Tyto datové struktury musí mít zámky (nebo jiná synchronizační primitiva) uvnitř, aby to bylo transparentní směrem ven.

Naše datové struktury a sdílené objekty se musí napsat tak, aby nemohl nastat deadlock - ten může nastat například tak, že se zavolá jedna metoda, která zamyká a v ní se zavolá další, která zamyká ten samý zámek. To lze vyřešit tak, že public metody budou na začátku zamykat, na konci odemykat a mezi tím volat jen privátní metody, které budou dělat skutečnou činnost. Ty privátní metody už budou počítat s tím, že je všechno zamknuté a nebudou nic samy zamykat. Public metoda nesmí volat jinou public metodu, ale jen ty privátní, které vykonávají skutečnou práci, aby nedošlo k deadlocku. Public metody stejně budou jen zamykat a pak volat ty privátní, takže to není žádné omezení.

Semafory jsou udělané přes condition variables - kontroluje se proměnná a podle výsledku se volá wait() nebo signal(). Malý problém je, že pthread_cond_signal probudí jedno čekající vlákno, ale není specifikováno, které to bude, takže může přehazovat pořadí.

Při zamykání je nutné najít co nejlepší kompromis mezi minimalizací doby zakázání přístupu a příliš častým voláním kernelu. Zamykat by se tedy mělo na minimální nutnou dobu, ale zase je lepší někdy zamknout větší oblast kódu místo odemčení a téměř okamžitého následného zamčení. Každé zamknutí i odemknutí je potenciální volání kernelu, takže by se to nemělo dělat zbytečně moc často. Musí se dávat pozor na výskyt potenciálních race conditions a deadlocku, takže je lepší používat zámky spíš bezpečně než pro maximální paralelismus. Také by se ale nemělo stávat, aby vlákna moc dlouho čekala v zámcích. Ideální je mít co nejméně sdílených dat, ke kterým by přistupoval větší počet vláken najednou

Nesmíme zapomínat odemykat zámky v případě, že nějakou funkci opustíme v případě chyby (ať už exceptionou nebo klasicky přes return value) - to pomáhá řešit takzvaný Scoped Locking. To znamená, že každý objekt, který se musí zamykat, bude mít asociovaný lock a na začátku každé jeho metody se vytvoří lokální objekt, takzvaný guard, který dostane jako parametr konstruktoru ten zámek asociovaný s objektem a zamkne ho v něm - pak v destruktoru při libovolném opuštění metody (scope platnosti identifikatoru) se ten zámek zase odemkne. Tím se zaručí, že nikde nezůstane něco zamknutého. Malý problém, na který se musí dávat pozor, je to, že tohle nebude fungovat, pokud se v kritické sekci vlákno zruší např pomocí thread_exit(). V takovém případě se totiž nezavolá destruktor lokálních proměnných.

Užitečné je použití immutable objektů, to znamená objektů, jejichž stav nelze změnit po zkonstruování. To se týká i komponent, které takový objekt obsahují. Přístup k immutable objektům se nemusí synchronizovat a celkově je pak program bezpečnější. Techniku lze použít i pro objekty, které se mění jen zřídka v poměru k počtu read-only přístupů. V takové situaci se hodí strategie copy-on-write. Příkladem jsou kontejnery objektů, kde se mohou občas změnit ty objekty nebo může být nějaký přidán, apod.

Pro efektivnější synchronizaci přístupu k objektům bychom také mohli použít varianty instrukcí test_and_set (podle procesoru - např cmpxchg na intelu), které by se volaly přes assembler. Není to portabilní přístup, ale zase nám to umožní zamykat v userlandu, abychom nemuseli kvuli každému malému zamknutí volat kernel. Je možné, že pthread library na Linuxu implementuje zámky přesně takhle, ale jistotu nemáme - do zdrojáků pro pthread se mi dívat nechce.

Transaction Processing

Transakční zpracování bude rozděleno do několika objektů. Všechny bude bude obsahovat samostatná vrstva mezi databázovým storage systémem a zbytkem aplikace. Jejich úkolem bude správa transakci a zámků nad daty. Tato vrstva se bude ptát indexů na strukturu stromu XML dokumentu, protože musí vědět, jak vypadá strom ze DocDataId jednotlivých uzlů, ale ukládat si to nebude. Také nebude rozumět XML.

Mezi objekty pracující s transakcemi patří Resource Manager, DocGroupManager, Locking Manager, Transaction Manager, Recovery Manager a JournalManager. Resource Mamager, DocGroupManager a Transaction Manager budou přístupné vyšším vrstvám v celém systému. Tyto tři objekty budou splňovat rozhraní definované v schematu txfs-iface, zbytek není pro vyšší vrstvy důležitý. Všechny zmíněné objekty podílející se na transakcích mohou být volány paralelně více vlákny - paralelnost bude končit až u práce s diskem ve storage systemu.

Resource Manager

Resource Manager bude poskytovat přístup k vlastním datům. Při požadavku na nějaká data se bude muset vždy předávat TransactionId, DocId a DocDataId (nebo odpovídající Id pro indexy) dat, kterých se operace týká. Přítomnost transakce budou muset zajistit vrstvy nad Resource Managerem (případně DocGroup Managerem). ResourceManager se taky v metodách pro data dokumentu bude TransactionManageru ptát na hodnotu proměnné update_lock pro danou transakci a pak ji předá LockManageru při volání zamykací metody.

Transaction Manager

Transaction Manager se bude starat o správu transakcí, především bude poskytovat operace begin, commit a abort. Transaction Manager musí být přístupný vyšším vrstvám z toho důvodu, že uživatel by měl mít možnost sám si říct, kdy chce pustit nebo commitovat transakci. Transaction Manager také bude psát do logu záznam o beginu, abortu nebo commitu transakce. Při commitu a abortu se také musí říct LockManageru, aby odemknul všechny zámky pro tu transakci, tj aby dodržel Strict 2PL protokol. TransactionId musí být přidělována v rostoucí sekvenci - není možné přidělit některé dříve použité.

Pokud bude chtít uživatel sám pustit nějakou transakci, tak zavolá příslušnou metodu objektu TransactionManager, která vrátí TransactionId. To pak bude předávat spolu s každým požadavkem v rámci té transakce. V případě, že s dotazem žádné TransactionId nepošle, tak se pro ten dotaz vytvoří automaticky samostatná transakce. Uživatel/aplikace bude o abortu automaticky vytvořené transakce informován pomocí exceptiony. To znamená, že v případě transakce vytvořené uživatelem se jen vyhodí exception, ale v případě automatické vytvořené transakce se ta bude rovnou i abortovat. Transaction Manager bude tedy muset být viditelný pro uživatele a nadstavby nativní XML databáze, spolu s DOM interfacem a XPath processorem.

Lock Manager

LockManager spravuje všechny zámky a tak reprezentuje scheduler. Vlákna vykonávající operace v transakcích poběží s občasným čekáním na zámcích. Vlastně se dá říct, že scheduler je reprezentován hierarchickým a striktním dvoufázovým uzamykacím protokolem a tak zajišťuje uspořadatelnost (serializability).

Proces zamykání a odemykání bude vypadat tak, že Resource Manager (DocGroup Manager) požádají LockManager o zámek na určitý uzel. Volání LockManageru se vrátí až ve chvíli, kdy je zámek nad tím objektem přiřazen volající transakci. Ta transakce pak s tím objektem nějak pracuje přes Recovery Manager a po nějaké době (třeba až při commitu nebo abortu) řekne LockManageru, že už to může odemknout. V tomto okamžiku může LockManager přiřadit zámek na ten objekt jiné transakci. Resource Manager (DocGroup Manager) může LockManageru dát pokyn k odemknutí zámku až ve chvíli, kdy Recovery Manager potvrdí, že poslední operace nad tím objektem je hotová a zapsaná na disk - tedy v případě použití Strict 2PL. Při každém volání metody LockingManageru se bude muset zamykat nějaký mutex, aby se zámky pracovalo vždy maximálně jedno vlákno.

Lockmanager bude při každém zamknutí uzlu v document tree po indexech chtít všechny předky toho uzlu až do kořene včetně (ten seznam předků se bude předávat s požadavkem na zamknutí). Každého z těch předků postupně porovná se zamčenými uzly a zjistí, jestli může zamknout požadovaný uzel včetně zamčení předků

pomocí intention locks. Ten seznam předků se bude procházet od kořene k listům. Indexy tedy musí seznam předků předávat tak, aby se poznalo, který předek je kořen - prostě musí vrátit i vztahy mezi těmi uzly třeba ve formě pevného pořadí (první je kořen, poslední je požadovaný uzel). Tuhle funkčnost zajistí třída NodeAncestors. Kontrola všech předků se musí dělat při každé operaci s daty dokumentů, aby se korektně zamykalo. Pokud by došlo ke kolizi, tak se vlákno v Lock Manageru zablokuje a vrátí se, až se zámek uvolní a bude si ho moci vzít. Lock Manager bude jediné místo, kde se budou vlákna blokovat kvůli transakčním zámkům. To blokování vláken se zajistí pomocí jedné condition variable - pokud vlákno nemůže jít dál kvůli transakčnímu zámku, tak zavolá condvar_wait(). Při odemknutí nějakého zámku se všechny vlákna čekající na condvar probudí a zkontrolují, jestli nemohou pokračovat. Pokud ne, tak se zas uspí, jinak budou pokračovat. Tohle sice není moc efektivní, ale nic lepšího mě nenapadlo. Při zamykání dat z dokumentů bude LockManager také dostávat hodnotu proměnné update_lock pro danou transakci, aby si mohl správně vybrat mezi shared lock a update lock při zamykání pro čtení. Pokud se bude transakce znovu pokoušet zamknout objekt, na kterém už má zámek, tak se neprovede žádná akce a volání příslušné metody v LockManageru uspěje.

LockManager si bude seznam všech zamčených bloků dat ukládat pomocí několika hash tables. Bude existovat jedna hash table indexovaná přes DocDataId, která bude mít jako hodnotu seznam struktur (objektů) a prvkem každé té struktury bude LockType, LockId a TransactionId. Podobně budou existovat další hash tables indexované přes GroupId, IndexDataId a DocIndexDataId. Pak bude existovat ještě jedna hash table indexovaná přes TransactionId, jejíž hodnoty budou také struktury (objekty) obsahující ukazatele na čtyři seznamy. V jednom seznamu budou DocDataId dat, které má transakce zamknuté a zbylé tři seznamy budou mít podobnou funkci pro GroupId, IndexDataId a DocIndexDataId. Kromě toho budou ještě tři tabulky indexované přes DocId nebo IndexId nebo DocIndexId, jejichž hodnoty budou jen booleany indikující, jestli je nějaký blok dat v odpovídajícím souboru zamčený - to je nutné pro zamykání skupin tak, jak je popsáno dále v textu.

Při zamykání se přidá prvek seznamu do jedné z prvních čtyř zmíněných hash tables a připadně se nastaví flag v tabulkách indexovaných přes DocId, apod. Jeden prvek se také vloží do odpovídajícího seznamu v hash table indexované přes TransactionId. Při odemykání se projdou ty seznamy indexované přes TransactionId v příslušné tabulce a odemknou se všechny zámky uložené v ostatních hash tables. Pak se taky smažou ty seznamy zamčených dat pro jednu transakci.

Recovery Manager

Recovery manager se bude starat o commitování a abortování transakcí a také bude komunikovat s JournalManagerem. Musí umět reagovat na příkazy read, write, commit, abort a restart. Také přes něj půjde každý požadavek na storage system. Ke změnám dat se bude přistupovat mechanismem odložené aktualizace - reálné nevratné akce (zápis na disk) se budou provádět až těsně před commitem (ve stavu PC). Recovery manager bude dočasné změny dat provedené jednou transakcí posílat do žurnálu a při commitu je znovu přečte a pošle storage systému k uložení na disk. Pokud Recovery manager odmítne provést nějakou akci, kterou po něm chce transakce, tak vyhodí výjimku - automaticky vyrobená transakce se také abortuje. Stejně tak když storage system vrátí chybu pro nějakou akci, tak se pošle ven exception. Pokud RecoveryManager dostane příkaz ke commitu transakce, tak se z příslušné metody vrátí až ve chvíli, kdy bude mít od StorageManageru potvrzeno, že je všechno zapsáno na disk. Příkaz restart bude sloužit k obnově konzistentního stavu databáze po výpadku. To znamená, že se musí zrušit necommitované transakce a obnovit výsledky commitovaných (nebo ve stavu PC), to všechno podle záznamů v žurnálu. Pokud bude operace restart přerušená další systémovou chybou, tak následný běh restartu musí přivést databázi do stejného stavu, jako kdyby doběhl ten první - operace restart musí být idempotentní.

Při commitu RecoveryManager řekne JournalManageru, aby si zaznamenal stav PC pro danou transakci a aby vrátil všechny změny provedené tou transakcí. Nejprve se musí ptát na nové dokumenty, globální indexy a skupiny, protože ta transakce do nich mohla už zapisovat. Ty změny se mohou vracet třeba přes objekt, který bude mít metody pro přečtení všech těch změn. Každá změna může být objekt obsahující DocId, DocDataId a vlastní data. Až RecoveryManager dostane od StorageManageru potvrzení, že jsou změny zapsané na disku, tak řekne JournalManageru, ať ze žurnálu smaže ty dočasné změny. Při abortu RecoveryManager řekne JournalManageru, aby ty změny jen zahodil. Zápis změn na disk při commitu musí být atomický z pohledu ostatních transakcí.

V RecoveryManageru se při ukončení transakce (přes commit nebo abort) bude také volat metoda releaseDocId a jí podobné potom, co se všechny požadavky na zápis dat pošlou storage systemu. RecoveryManager by ale jednu z těchto metod měl zavolat jen v případě, že soubor odpovídající danému FileId nepoužívá i nějaká jiná transakce - tohle by si mohl pamatovat přes čítač nebo s pomocí JournalManageru. V těch metodách se totiž uvolní z paměti informace o tom souboru a taky se z cache vyhodí všechny bloky, které mu patří.

Journal Manager

JournalManager bude zprostředkovávat práci se žurnálem. Bude mít vyhrazen jeden adresář, do kterého bude ukládat soubory s dočasnými změnami, které mu pošle Recovery Manager. Pro každou transakci bude existovat samostatný soubor určený pomocí TransactionId, kde budou dočasné změny provedené odpovídající transakcí. TransactionId by mohlo být součástí názvu souboru, ale taky si bude JournalManager držet přiřazení mezi TransactionId a souborem. Soubor bude mít přesně definovanou strukturu. Záznamy v žurnálu se budou skládat z dat reprezentujících změnu a z identifikačních čísel těch dat. Čas uložení do žurnálu se nikam psát nemusí, protože pokud transakce vyrobí novou hodnotu pro nějakou položku v databázi, tak se ta stará zahodí. Nějak se musí reprezentovat i dočasně vytvořené skupiny a dokumenty. Změny v indexech se budou psát do stejného souboru jako změny v dokumentu. Pro každý soubor s dočasnými změnami bude existovat objekt nějaké třídy, který si bude pamatovat strukturu toho souboru. Hlavně bude mít tři hash table (pro dokumenty, globální indexy a dokumentové indexy), jejichž klíčem budou příslušná Id a hodnotou offset do žurnálového souboru.

Transakce vidí změny dat, které provedla, ještě před commitem, takže při čtení v rámci transakce se musí dívat do žurnálu, jestli tam není nová verze dat pro určité DocDataId. JournalManager bude ty změny psát okamžitě na disk pomocí Direct IO, ale také si je bude držet v paměti kvůli rychlejšímu čtení. Uložení žurnálu s dočasnými hodnotami na disku bude sice znamenat zpomalení oproti uložení v paměti, ale zase to umožní po havárii obnovit transakce, které byly v době pádu ve stavu PC.

Do žurnálového souboru se bude psát značka pro transaction begin (při vytvoření souboru a zapsání první změny) pro transakci s daným TransactionId, pak jednotlivé změny dat (bezprostředně po provedení) a ve stavu PC se zapíše další značka. Zároveň si do nějakého souboru zapíše, že tato transakce je ve stavu PC a pak při commitu to z toho souboru zas smaže. Změny v žurnálu budou ukládané v následujícím formátu. Na začátku bude jeden byte, jehož obsah určí, jestli se změna týka dokumentu, globálního indexu nebo dokumentového indexu. Pro změny může ten byte nabývat hodnot od 3 do 63 včetně. Samostatné hodnoty toho byte budou odpovídat značkám pro begin (1) a stav PC (2). Pak bude TransactionId a za ním DocId a DocDataId (nebo odpovídající Id pro indexy obou typů) a na konci bude nová verze příslušného bloku dat. Data smazaná jednou transakcí budou ve stejném souboru jako updaty a označené ty záznamy budou tak, že první byte bude mít hodnotu pro změnu dat stejné kategorie (dokument, dokumentový index, globální index) zvýšenou o 64. V tom záznamu pro smazaná data také samozřejmě nebude nová verze těch dat. Pro nové dokumenty, globální indexy a skupiny dokumentů vytvořené transakcí budou také existovat záznamy v žurnálovém souboru. Jako hodnoty prvního byte se v tomto případě mohou použít čísla od 128 dál.

JournalManager bude podporovat operace undo a redo - ty se budou volat v případě příkazu restart pro RecoveryManager. V případě systémové chyby (a vymazání cache) se znovu provedou všechny transakce, které byly v okamžiku chyby ve stavu PC, ale ještě nebyly commitované (jejich výsledky nebyly na disku). To se zjistí přečtením toho souboru se seznamem transakcí ve stavu PC. Každou transakci, pro kterou existuje v žurnálu soubor, dá do undo listu a pokud dále narazí na značku pro stav PC, tak ji přesune do redo listu. Až se projde celý žurnál, tak se změny provedené transakcemi v redo listu pošlou storage systemu a transakce v undo listu se zruší. Nakonec se upraví žurnál, aby neobsahoval zbytečné záznamy.

Transakce

Každý požadavek (dotaz, update) se vždy bude zpracovávat v rámci jedné transakce. Do jedné transakce se zahrnou změny dat dokumentů i indexových struktur. Systém bude podporovat jen ploché transakce s vlastnostmi ACID. Na začátku transakce by měl Transaction Manager na požádání (voláním begin_transaction) vygenerovat TransactionId, které se pak použije jako parametr požadavku na Resource Manager (nebo DocGroup Manager).

Poslední akce každé transakce musí být commit_transaction nebo abort_transaction. Pro updatovací transakce bude commit znamenat uložení na disk (respektive pokyn pro storage system, aby data uložil). Commit transakce se smí provést jen pokud jsou commitnuté všechny transakce, jejichž výsledky tato transakce už viděla. Tím se zabrání kaskádovým abortům, protože transakce pak nejsou závislé na necommitovaných transakcích, které mohou být zrušené. Všechny zmíněné požadované vlastnosti by měl zajistit strict two-phase locking protocol. Občas může být nutné použít techniku zpožděných commitů.

Před shutdownem databáze musí být ukončené všechny transakce a změny se musí zapsat na disk. To znamená, že nějakou dobu před shutdownem už by se neměly pouštět nové transakce.

Zamykání dat a metadat

Pro zamykání stromu reprezentujících XML dokumenty se nejvíc hodí hierarchical (multi-granularity) locking. To je z pohledu efektivity kompromis mezi přílíš častým zamykáním malých objektů (jednotlivých elementů) a dlouhodobým zamykáním velkých objektů (stromu s větším počtem elementů). Dlouhotrvající transakce obvykle budou zamykat objekty na vyšší úrovni, zatímco krátké budou zamykat malé objekty. Funguje to tak, že explicitní zamknutí nějakého uzlu určitým typem zámku implicitně zamkne stejným typem zámku celý podstrom toho uzlu.

Pro informaci o tom, že někde hlouběji v hierarchii má jedna transakce zamčený nějaký objekt, slouží intention locky. Zámek typu RIW (zvaný také update lock) znamená současnou aplikaci zámků R a IW - hodí se v situaci, když transakce zamkne hodně objektů pro čtení, prochází je a do vybraných později něco zapíše (po zamknutí pro zápis). Jeho cílem je nechat data zamknuté co nejdéle pro čtení, ale pak poskytovat možnost bezpečné změny na write lock (bez rizika deadlocku). Kompatibilní dvojice zámků jsou: (R,R), (R,IR), (IR,R), (IR,IR), (IR,IW), (IR,RIW), (IW,IR), (IW, RIW), (RIW,IR).

Před zamčením objektu A musí Lock Manager zaručit, že žádný předek objektu A nezamyká implicitně A nekompatibilním typem zámku. To se zaručí tak, že před zamknutím objektu A Lock Manager musí zamknout pomocí odpovídajícího intention locku všechny předky A od kořene hierarchie až k jeho bezprostřednímu předkovi.

Při konverzí zámku na jiný se uzel nakonec musí zamknout zámkem, který je stejně silný nebo silnější, než starý i nový. K tomu může sloužit tabulka konverzí. Její prvky se dají intuitivně určit podle síly zámků v dále popsaném uspořádání - malý chyták je, že při změně R na IW nebo IW na R se musí použít RIW lock. W je silnější než RIW, RIW je silnější než R a IW a R je silnější než IR. R a IR jsou v tomto uspořádání neporovnatelné s IW.

Protokol pro multigranularity locking:

1. 1) pokud X není kořen hierarchie, tak před nastavením R nebo IR na X musí mít transakce zámky IR nebo IW na rodiči X.
2. 2) pokud X není kořen hierarchie, tak před nastavením W nebo IW na X musí mít transakce zámek IW na rodičí X.
3. 3) před čtením X musí mít transakce zámek typu R nebo W na nějakém předchůdci X nebo přímo na X. Pro zápis do X musí mít zámek W na předchůdci nebo přímo na X.
4. 4) transakce nemůže uvolnit intention lock na X, pokud má zamčeného nějakého potomka X.

Tyto pravidla vlastně říkají, že zamykání probíhá v root-to-leaf order a odemykání v opačném pořadí.

Pro zajištění uspořadatelnosti se musí MGL (multigranularity locking) zkombinovat se Strict 2PL (two-phase locking) a deadlocky vyřeší správné použití update locku. Striktní dvoufázové zamykání je nutné proto, aby nemohl nastat unrepeatable read (porušení uspořadatelnosti). Tohle právě zařídí Strict 2PL protokol, který je i poměrně snadno implementovatelný. Zamknutí vyššího uzlu místo mnoha jeho potomků (lock escalation) neporuší Strict 2PL, protože ti potomci se sice jakoby odemknou před commitem, ale hned budou zamčeni implicitně po umístění zámku na vyšší uzel. V Lock manageru se jen přidá zámek na ten vyšší uzel a pak se mažou zámky na potomcích.

U MGL není problém zamykání malých objektů, ale složitější může být rozhodnutí, kdy zamknout nějaký uzel ve vyšší úrovni hierarchie. Tento problém se může vyřešit tak, že každý uzel zamčený nějakým typem zámku bude vědět, kolik má zamknutých potomků. Při určitém počtu zamčených potomků můžeme zkusit dát zámek na vyšší uzel (větší oblast). Tento princip se nazývá "Lock Escalation". Pokud se na vyšší uzel nepodaří zámek dát kvůli kolizím, tak se to neudělá a vlákno se neuspí, protože to by vedlo k deadlocku - jen se prostě nepovede dát ten zámek. To přesouvání zámků na vyšší uzel bude interní záležitost Lock Manageru, takže ostatní objekty nebudou mít možnost dát k něčemu takovému pokyn. Implementovat se to může tak, že při zamykání uzlu se Lock manager podívá, jestli otec toho uzlu nemá moc zamknutých potomků - pokud ano, tak zamkne toho otce v případě, že by vlákno nemuselo čekat na uvolnění nějakých zámků na otci. Přesouvání zámku na vyšší uzel zvyšuje efektivitu (menší počet zamknutí/odemknutí), ale v principu nevadí, když budou zámky na uzlech niž v hierarchii. Když bude v podstromě pracovat víc transakcí nebo by zámky kolidovaly, tak se nadřazený uzel nebude zamykat, v případě jedné pracující transakce nebo kompatibilních zámků ano.

Při použití multi-granularity (hierarchical) locking může deadlock nastat snad jen v případě, že dvě transakce zamknou stejný uzel pro čtení a později ho obě budou chtít zamknout na zápis. Problém se objeví v okamžiku, kdy chtějí zamknout ten uzel pro zápis a nemohou, protože ho má druhá transakce zamknutý pro čtení - v tu chvíli nastává deadlock. Tohle řeší RIW locks (také zvané update locks) - mechanismus pro prevenci deadlocků. Hlavní problém RIW locků je rozhodnutí, jestli pro čtení použít shared lock (R) nebo update lock (RIW). Při vytváření každé transakce (operace begin) se musí specifikovat hodnota flagu určující, jestli se pro všechny čtení v rámci té transakce bude používat shared lock nebo update lock. Třeba XPath processor bude vždy nastavovat flag na shared (nikdy nezapisuje), ale DOM bude častěji používat update lock. Defaultní použití update locku v rámci transakce bude znamenat omezení paralelismu, ale zase to zabrání deadlockům. Pokud ten flag DOM nebo XPath procesor nastaví špatně (vybere shared, ale pak bud chtít změnit shared lock na exclusive), tak hrozí výskyt deadlocku.

Zamykat se budou muset i indexové struktury (obou druhů - dokumentové i globální indexy), aby dvě vlákna zároveň nemodifikovaly index na disku neslučitelným způsobem. Na to bude nejvhodnější striktní dvoufázové zamykání s read/write zámky. Pro prevenci deadlocků můžeme použít strategii "wait or die" popsanou dále. Zamykat pomocí read-write locků se nejspíš budou muset i indexy v paměti, protože při nějakém updatu se změní v pamětí a pak i na disku. Není možné zamykat jen přístup na disk a v paměti to nechat odemknuté.

Pro zabránění deadlocku u dvoufázových zamykání indexů se bude provádět následující: transakce budou očíslované rostoucími čísly (TransactionId) a pokud se nějaká transakce bude pokoušet zamknout neslučitelným typem zámku objekt, který má zamknutý transakce s nižším číslem, tak se ta transakce s vyšším číslem rollbackuje. Budou se rollbackovat transakce s vyšším číslem, protože starší toho pravděpodobně udělala víc, takže by se víc dočasných výsledků zrušit. V případě, že transakce bude chtít zamknout objekt, který drží transakce s vyšším číslem, tak bude čekat. To odpovídá principu "wait or die". Je otázka, jestli se takhle nebude abortovat zbytečně moc transakcí (častý výskyt false positives na deadlocky).

Dobře se musí synchronizovat také tvorba a především mazání dokumentů a jejich skupin (přístupová práva k těmto operacím se řešit nebudou). Při přidávání nebo mazání dokumentu je nutné získat zámek (write lock) na celou skupinu, do které patří, aby si později příchozí transakce ani nemohly zjistit DocId pro ten dokument. Při přidávání a mazání skupiny se musí zamknout její předek ve stromu skupin. Bylo by ale nešikovné, kdyby se při každém požadavku na data přes ResourceManager muselo kontrolovat, jestli není zamknutá skupina, do které patří dokument s daty. Lepší bude, když se při zamykání dat z nějakého dokumentu poznačí, že je tento dokument

"zamčen" libovolným zámkem kvůli datům a při zamykání skupiny se atomicky zkontroluje, jestli jsou odemčené všechny dokumenty z té skupiny. Pokud budou odemčené, tak se zamkne ta skupina a všechny dokumenty v ní a poznačí se, že ty dokumenty jsou zamčeny kvůli skupině. Zamykání skupin může být pomalé, protože se bude dělat méně často než zamykání dat. Skupiny dokumentů tedy budou mít oddělenou hierarchii pro MGL od dat v dokumentech. Ty značky, že je dokument "zamčený" pro účely zamykání dat nebo zamykání skupin, se mohou ukládat v hashtable indexované přes DocId.

Logování

Persistentní logování a práce s log-soubory bude patřit do kompetence objektu třídy LogManager. Log messages se budou zapisovat vždy jen do jednoho souboru, který se bude každý den automaticky měnit. Logování bude využívat především TransactionManager, StorageManager a network server. Zaznamenávat se budou systémové zprávy (oznámení o startu a ukončení serveru), chybové zprávy (pro disk a síť), a operace begin, commit a abort u transakcí. Do logu se bude pouze přidávat. Operace zápisu zprávy do logu nebude součástí transakce, aby se záznamy nezrušily při rollbacku.

LogManager bude logy ukládat do souborů se jménem sestaveným ze zadaného prefixu a datumu nebo podobného identifikátoru času. Ten prefix a adresář pro logy se specifikuje v konfiguraci. V každém volání logovací metody se bude volat funkce gettimeofday, protože na začátek každého záznamu se vloží přesný čas jeho zápisu. Aktuální čas se vždycky uloží a při dalším volání se pak zjistí, jestli nezačal nový den a nemusí se vyměnit soubory. Množství logovaných zpráv se možná bude moci určovat v konfiguraci - mohou se zaznamenávat jen systémové a chybové zprávy nebo i transakční operace.

Ukládání XML dokumentů

Dokumenty se budou ukládat na disk pomocí databázového storage systému. Vyšší vrstvy (např reprezentace v DOM) budou storage systému říkat, že má načíst/uložit element s určitým DocDataId z/do nějakého dokumentu určeného pomocí DocId. Vrstva DOMu bude aplikacím poskytovat přístup k jednotlivým uzlům. Důležité je při ukládání zachovat XML dokument i s komentáři, entitami, processing instructions, apod.

Pro dokumenty nebudeme požadovat DTD ani XML Schema, protože nic takového nepotřebujeme. Kontrolu validity si může provádět nadstavba nebo klient sám a pro vyhodnocování dotazů nebo tvorbu indexů DTD také nepotřebujeme. Každý vkládaný dokument ale musí být well-formed - jinak ho musí náš XML parser odmítnout.

To, že nebudeme požadovat DTD, má jeden trochu nepříjemný důsledek pro defaultní hodnoty atributů uváděné právě v DTD. Náš XML parser a XPath processor ale o těch defaultních hodnotách atributů nebude vědět právě proto, že nebude číst DTD. Problém je, že někdo by se na tu defaultní hodnotu mohl ptát v XPath expression a dostal by pak nekorektní výsledek. Na druhou stranu to není žádná fatální chyba, protože specifikace XML říká, že non-validating parsers nemusí číst externí DTD a specifikace XPath výslovně uvádí, že v případě použití non-validating parseru nemusí defaultní atributy fungovat úplně správně. Nejspíš tedy defaultní hodnoty pro atributy nemusíme podporovat - ani v případě uvedení default value v internal DTD.

XML dokumenty mohou také obsahovat nějaká binární data, která ale budou zakodována do obyčejných znakových řetězců. Nejpoužívanější způsob je asi Base64 encoding, které kóduje do řetezců z 64 znaků (A-Za-z0-9+/) a na konci je ještě ’=’ - viz RFC 2045. Tím pádem by asi nemělo dělat problémy překodování do UTF-8 a zpět, protože použité znaky patří do basic ASCII (znaky s kódy 0-127) a tak mají ve všech charsetech stejný kód.

DOM

DOM reprezentující XML dokument podle XPath data modelu by se měl postavit nad indexovými strukturami a Resource Managerem. Indexů se bude ptát na Id potomků elementu a po Resource Manageru bude chtít data pro ty elementy podle DocDataId.

DOM je pouze interface pro přístup k objektům uloženým v dokumentu, takže neudržuje žádné vztahy mezi objekty - na strukturu dokumentu se ptá indexů. DOM by byl použitelný hlavně jako rozhraní pro aplikační programy, které chtějí přistupovat k objektům ve stromě a také pro modifikace dokumentů. DOM také může fungovat jako cache pro data z databáze, aby nemusel pořád volat storage system a transakce. Obsah databáze na disku je ale vůči DOMu autoritativní, takže při sporu má resource manager (storage system, recovery manager) vždy pravdu.

Dotazy na XML dokumenty

V databázi budou uloženy XML dokumenty, nad kterými se budou provádět dotazy v jazyce XPath. Všechny dotazy budou vracet node-sets obsahující prvky daného XML dokumentu. Implementovat budeme XPath 1.0, protože verze 2.0 je příliš složitá. Návrh bude takový, aby to šlo v budoucnosti rozšířit na XPath 2.0. Tu verzi 1.0 naimplementujeme pravděpodobně celou.

Procesor dotazů v XPath bude mít několik variant metody processXPathQuery(string XPathQuery), které budou vracet XMLNodeSet - detaily dále. Objekt třídy XMLNodeSet bude mít metody pro přístup k vráceným elementům, textu, apod. Ty elementy a další data z XML dokumentu by se mohly reprezentovat a vracet jako DOM objekty. Třída XMLNodeSet by měla mít i metodu toBytes nebo toString, která vrátí její obsah jako pole bytů. To bude nutné hlavně pro posílání dat přes síť. Dotaz na neexistující prvky v XML dokumentu by měl asi vrátit prázdnou množinu. Nemůže vrátit chybu nebo warning, protože XPath processor nebude mít přístup k schematu dokumentu v DTD nebo XML Schema.

Interface pro XPathProcessor bude mít následující podobu - popisuji zde jen metody, které by nadstavbové aplikace (včetně našeho thread poolu) nebo uživatelé knihovny mohli požadovat. Prakticky se jedná o varianty jedné základní metody "XMLNodeSet processXPathQuery(string query, XPathContext ctx, TransactionId trid)", kde se varianty liší přítomností či nepřítomností parametrů XPathContext a TransactionId. Parametr query musí v případě nepřítomnosti XPathContext obsahovat absolutní location path a v případě přítomnosti contextu může být location path i relativní - query se potom bude vyhodnocovat vzhledem k tomu contextu, který dodá nějaká nadstavba XStore (např XQuery processor). V tomto případě musí metoda také vyhazovat exception InvalidXPathContext, pokud bude context nevalidní. U variant metody processXPathQuery bez parametru TransactionId se ten dotaz interně vyhodnotí jako samostatná transakce, takže se uvnitř té metody musí zavolat tx_begin a tx_commit. Pokud bude parametr TransactionId přítomen, tak se dotaz vykoná v rámci (uživatelem vytvořené) transakce s dodaným TransactionId. Podpora XPathContextu je nutná hlavně kvůli možným rozšířením pro XQuery a parametr TransactionId je nutný pro to, aby si uživatel mohl vyrobit vlastní transakci přes TransactionManager a v jejím rámci provést víc dotazů. Instanci XPathProcessoru budou volat vlákna z thread poolu a různé testovací prográmky.

Všechny varianty metody processXPathQuery by měly vracet node-set zabalený do objektu, který by implementoval rozhraní s názvem XMLNodeSet. To rozhraní musí mít metodu toString, která jeho obsah převede do stringu (analogicky k metodě toString v jazyku Java). Serializace toho objektu do stringu se bude provádět před posíláním výsledku dotazu klientovi po síti. Kromě metody toString by měl interface XMLNodeSet obsahovat i metody pro přečtení každé node v tom node-set, apod.

Třida XPathContext by měla obsahovat cestu ke context node v daném dokumentu, context size, context position a variable bindings (viz specifikace XPath). Context size je velikost node-set, který byl vybrán v předchozím location stepu. Context position je pozice současné context node v celém contextu. Před prvním location step je context size obvykle 1, ale pak se mění podle toho, kolik nodes odpovídá tomu location stepu, atd. Pokud budeme v contextu podporovat i variable bindings a query bude obsahovat jméno proměnné, tak musíme substituovat hodnotu za jméno.

Dokument, ze kterého se mají brát data pomocí XPath, se bude specifikovat pomocí konstruktu dokument(<cesta k dokumentu>). Parametr toho konstruktu by popisoval cestu ve stromové struktuře skupin dokumentu. Při přidávání dokumentu by se ta cesta také musela zadat. V rámci XPath se musí implementovat i funkce jako substring(), number(), apod.

V dotazech v jazyce XPath mohou být některé znaky vyjádřeny pomocí character entities (<, ", &#cislo, apod), protože v XSLT a XQuery se XPath expressions někdy čtou z atributů. Před vyhodnocováním expression bychom asi měli tyhle entity expandovat. Pokud je entita jedna z <, >, ", ' a &, tak by asi stačilo nahradit ji odpovídajícím znakem. POkud to bude entita tvaru &#DecNum; nebo &#xHexNum;, tak bychom ji asi měli nahradit odpovídajícím znakem z iso-10646. Ten znak by mohl jít vyrobit tak, že se to číslo přiřadí do proměnné typu wint a to se pak může převést do utf-8. S tím taky souvisí otázka, jestli budeme všechny dotazy v XPath požadovat v utf-8 nebo ne.

Parser musí dotaz v XPath převést do nějaké interní reprezentace dotazu. Parsery XPath a XML by asi měly vstup validovat podle gramatiky. Měly by se také kontrolovat parametry dotazů, aby tam uživatel nedával něco, co by mohlo způsobit problémy - parser musí být co nejvíc neprůstřelný. K parsování XML se použije Martinův ročníkový projekt nebo už hotová a odzkoušená knihovna, např libXML2 nebo Xerces-C++.

Modifikace XML dokumentů

Modifikace se budou dělat přes DOM pomocí přidávání a ubírání uzlů ve stromu. XUpdate se nebude implementovat, protože to není oficiální standard W3C a modifikační konstrukce se stejně stanou součástí jazyka XQuery. Alternativou je přidávání a mazání celých dokumentů. Při vkládání nového dokumentu nebo modifikaci existujícího se musí nová data rozparsovat a převést do stromu, pak se musí jednotlivé objekty uložit na disk a nakonec se vyrobí indexy obsahující DocDataId přiřazené objektům storage systémem. Modifikační příkazy tedy budou vracet status úspěchu akce nebo nějaká Id, v závislosti na konkrétní akci.

Pro přidávání a mazání dokumentů (a skupin dokumentů) bude existovat samostatný objekt s názvem DocGroupManager, protože to nejde přes DOM ani přes XPathProcessor. Ten objekt bude umět dokument jen vytvořit - vlastní obsah dokumentu se musí vložit standardně po částech přes ResourceManager po zpracování XML parserem, apod. DocGroupManager bude také přístupný vyšším vrstvám stejně jako Resource Manager a Transaction Manager - bude to takové částečné administrační rozhraní. Také bude umět převádění cesty ve stromě skupin na DocId a GroupId, které se musí předávat StorageManageru. Převod bude využíván XPath parserem nebo kompilerem a DOMem před prvním přístupem k dokumentu v rámci transakce. Samotný převod se bude provádět pomocí nějaké hash table ve StorageManageru.

Indexové struktury

Indexy budou převádět cesty k elementům v dokumentu na DocDataId použitelné pro transaction manager a storage system. Indexovat se bude výskyt elementů a atributů, obsah atributů a také cesty v dokumentu. Cesty se budou indexovat hlavně takové, které odpovídají často používaným axis v XPath (rodič-potomek, apod). Důležitý je také index, který si bude pamatovat stromovou strukturu XML dokumentu - ten je nutný hlavně pro DOM. Obecně by indexy měly urychlovat vyhodnocování dotazu a měly by se navrhnout podle toho, co umí XPath. Například by urychlovaly provádění dotazů, které se ptají na elementy s určitou hodnotou atributu. Indexy by si mohly pamatovat také ty věci, které se píšou do hranatých závorek v dotazech v XPath. Také by asi měly ukládat typ všech prvků v dokumentu z pohledu XML-Infoset (datový model pro XPath), protože to je nutné pro vyhodnocování některých funkcí z XPath Core Function Library - např text() nebo node(). Indexy si budou pamatovat strom DocDataId jednotlivých objektů v XML dokumentu v různých tabulkách a budou rychle generovat DocDataId objektů, které chtějí po storage systému. Každá položka indexu by měla obsahovat DocDataId odpovídajícího kusu dat s jeho atributy nebo null (analogicky pro textový obsah elementů).

Oddělené ukládání objektů v XML dokumentu má za následek, že při požadavku na celý dokument nebo na větší počet elementů to pak musí vrstvy nad storage systémem poskládat do správného tvaru, co se týče pořadí elementů. Musí se tedy dodržet document order - to je jedna ze základních vlastností jazyka XML. Je důležité dodržet pořadí potomků u všech elementů (celý dokument musí vypadat stejně jako při vkládání) kvůli tomu, že aplikace ho pak může kontrolovat proti nějakému schématu v XML Schema, apod. Správné pořadí by mohlo být uložené v té indexové struktuře, která obsahuje informace o stromové struktuře dokumentu.

Všechny indexové struktury budou ukládané na disk, ale kromě toho budou taky v paměti, takže kromě transakční concurrency control, která řeší konzistenci na disku, se přístup k nim musí synchronizovat i pomocí mutexů v paměti.

V indexech by taky měly být uloženy hodnoty atributů ID a IDREF pro elementy, které takové atributy obsahují. V XPath Core Function Library je funkce id(), která vrací elementy s daným ID. Při přítomnosti té funkce v expression by se jen prošel index a vybral příslušný element.

Různé Alternativy

Reprezentace XML dokumentů ve storage systému jako strom

Odlišný přístup je ten, že by storage systém znal strukturu stromu objektů v dokumentu. To má jako hlavní výhodu to, že při prvním požadavku na objekt z dokumentu by se nahrál rovnou celý blok, kde jsou i sousední elementy podle stromové topologie. Při odděleném ukládání objektů by se mohlo stát, že by pokaždé chtěl objekt z úplně jiného bloku a to by zpomalovalo - aspoň při prvních několika přístupech na data jednoho dokumentu. Storage systém, který by musel znát tu strukturu stromu, by se pak dal dodělat v případě, že by ten náš vykazoval špatné výsledky pro operace nad běžnými dokumenty. Ani by se nemusely dělat velké změny v rozhraní. Jen by se do storage systému přidala funkce, kterou by zavolal transakční manažer a předal tak storage systému strom z Id všech objektů v dokumentu.

Tato reprezentace dat ve storage systému počítá s tím, že mu bude známá stromová struktura každého dokumentu. V tom případě bude obsah dokumentu rozdělen do bloků (alokačních jednotek filesystému) tak, že kořenový element a nejbližší potomci budou v jednom bloku, každý podstrom pak bude v dalším bloku a tak se půjde až k listům. Cílem je mít uzly stromu dokumentu, které jsou blízko u sebe, ve stejném bloku (stránce). To by ještě šlo upravit tak, že by se dynamicky měnilo rozdělení stromu celého dokumentu do bloků podle toho, jak se nejčastěji prochází tím stromem dokumentu. Bylo by vhodné, aby dvojice elementů ve vztahu "rodič - potomek", mezi kterými vede často využívaná hrana stromu, byly v jednom bloku. Při použití této reprezentace by storage systém mohl při požadavku na nějaký element rovnou vracet celý podstrom toho elementu. Tato operace by trvala kratší dobu než při použití reprezentace s oddělenými daty elementů.

Stromový způsob ukládání dokumentů by určitě byl náročnější na implementaci, ale zase by mohl být efektivnější a optimálnější pro XML dokumenty. Při častých updatech je rychlejší způsob s elementy bez vazeb, ale při méně častých updatech (pro statické obsahy dokumentů) je rychlejší strom. Stromové uspořádání má blízké elementy ve stejném bloku, takže může rychleji vracet požadovaná data. Při častém updatování by ale zpomalovalo udržování struktury stromu a rozdělení elementů do bloků.

Třetí a nejlehčí způsob ukládání XML dokumentů je uložit vždy dokument jako celek, při změne ho smazat a nahradit novým. Změny se tedy budou provádět jen v paměti. Tahle metoda ale požaduje načtení celého souboru do paměti.

Mechanismus ukládání zámků v LockManageru

Druhá možnost ukládání zámků je nepředávat různým metodám po cestě TransactionId, ale pointer na objekt Transaction (vyrobený TransactionManagerem během operace begin). Při každém zamčení do něj LockManager přidá další LockId a při commitu nebo abortu transaction manager vezme všechny ty LockId a postupně je nechá odemknout. Při použití této metody by objekt Transaction mohl ukládat i všechny dočasné změny - dá se říct, že by nahradil JournalManager.