Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Bartłomiej Budzyński
Nr albumu: 174782

Zwiększenie odporności systemu LoXiM na awarie – implementacja dziennika

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Maj 2007

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W niniejszej pracy autor opisuje działanie dziennika wycofań i powtórzeń z niespoczynkowymi punktami kontrolnymi jako metodę zwiększania bezpieczeństwa spójności danych w bazie danych i odporności systemu zarządzania bazami danych na awarie. Dodatkowo przedstawia wykonaną implementację dziennika w istniejącej bazie danych LoXiM.

Słowa kluczowe

dziennik wycofań i powtórzeń z punktami kontrolnymi, obiektowe bazy danych

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2 Database Management
H.2.4 Systems

Tytuł pracy w języku angielskim

Increased resilience in LoXiM system – implementation of log

Spis treści

Rozdział 1

Wprowadzenie

Nowoczesne systemy bazodanowe są programami coraz bardziej powszechnymi i stosowanymi do coraz bardziej odpowiedzialnych zadań. Jednym z podstawowych wymagań tych systemów jest bezpieczeństwo przechowywanych danych, także w obliczu awarii – również sprzętowych. W przypadku awarii dane systemu muszą być chronione.

Implementacja dzienników (*logs*) jest zasadniczą techniką zwiększającą odporność (*resilience*) [GMUW06]. Zawierają one informacje na temat historii zmian zachodzących w bazie danych i umożliwiają odtworzenie spójnego stanu danych w bazie danych.

W procesie odtwarzania (*recovery*) istotną cechą wpływającą na wydajność, jest możliwość uniknięcia sytuacji w której konieczna jest analiza dzienników z odległej przeszłości. Technika ułatwiająca to zadanie nazywa się metodą tworzenia punktów kontrolnych (*checkpointing*).

Archiwizacja umożliwia natomiast przetrwanie bazy danych także w sytuacji całkowitej utraty systemu i jego danych. Dzięki wykorzystaniu mechanizmu dzienników do tworzenia kopii zapasowej, spójny stan systemu można archiwizować bez przerywania jego pracy.

Praca ma następującą strukturę. W rozdziale 2 omówiono podstawowe zagadnienia z zakresu prowadzenia dzienników w systemach zarządzania bazami danych. W rozdziale 3 przedstawiono metodę tworzenia kopii archiwalnej bazy danych bez zatrzymywania przetwarzania transakcji. W rozdziale 4 zaprezentowano implementację dziennika wycofań i powtórzeń z niespoczynkowymi punktami kontrolnymi w systemie LoXiM. W rozdziale 5 dokonano podsumowania oraz przedstawiono propozycje kontynuacji badań opisanych w tej pracy. W dodatku A przedstawiono krótki opis instalacji oraz sposobu użytkowania załączonego oprogramowania. Dodatek B zawiera opis typów rekordów dziennika.

Rozdział 2

Prowadzenie dzienników

2.1. Awarie

Podczas pracy systemów zarządzania bazami danych, wiele zdarzeń może się nie powieść i może dojść do różnych rodzajów awarii, które z kolei mogą mieć skutek w rozspójnieniu danych w bazie danych. W zakresie pojawiających się problemów są zarówno błędy wprowadzania danych do systemu przez użytkownika jak i mechaniczne uszkodzenia komputerów i nośników danych. Poniżej lista często spotykanych zagrożeń dotyczących systemów zarządzania bazami danych.

błąd przy wprowadzaniu danych – ten typ awarii powinien być obsługiwany przez oprogramowanie użytkownika lub bazy danych (np. za pomocą wewnętrznych więzów spójności); jeżeli użytkownik wpisuje kod pocztowy i pominie jedną cyfrę, wówczas można sprawdzić poprawność; niestety nie wszystkie rodzaje tego typu błędów mogą zostać wykryte przez – np. przekręcenie cyfry w kodzie pocztowym jest niewykrywalne przez system.

uszkodzenia nośników – jest wiele metod zabezpieczenia nośników zarówno sprzętowych (np. schematy połączenia dysków typu RAID), jak i programowych (np. archiwizacja[1] czy redundantne kopie baz danych, rozprzestrzenione po różnych lokalizacjach).

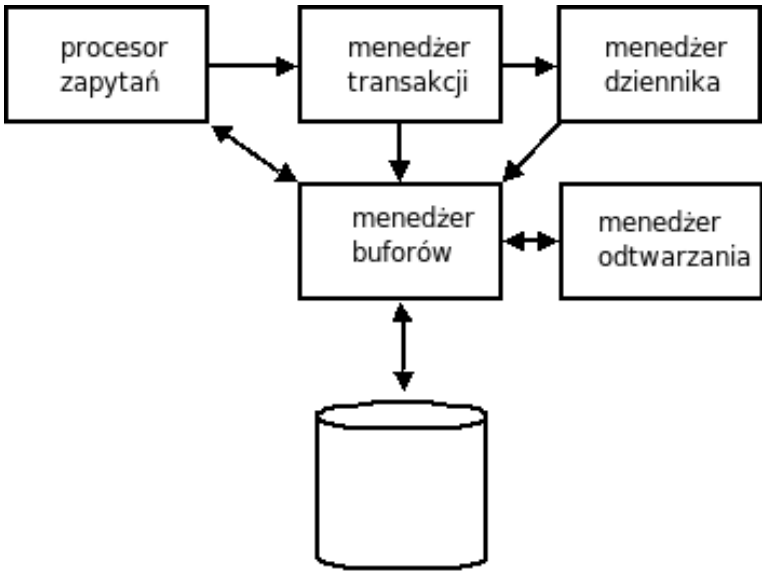
awarie katastrofalne – awarie polegające na kompletnym zniszczeniu nośników danych (np. pożar, zalanie czy kradzież); w takiej sytuacji można skorzystać z technik z poprzedniego punktu.

awarie systemowe – to sytuacje w których wykonywanie transakcji zostaje zatrzymane (więcej o transakcjach w rozdziale 2.2); typoy przykład takiej sytuacji to zanik zasilania w połowie wykonywania transakcji; w takiej sytuacji po odzyskaniu zasilania system powinien utrzymać spójny stan danych, przy możliwie małej utracie transakcji; sytuacja idealna polegać będzie na zachowaniu wszystkich danych uzyskanych z zakończonych transakcji i wycofaniu wszelkich zmian dokonanych przez niezakończone.

2.2. Transakcje

Transakcja jest jednostką wykonywania operacji na bazie danych. W systemie LoXiM językiem zapytań jest SBQL [Sub04]. Transakcje muszą wykonywać się automatycznie, to znaczy

[1] także z użyciem dzienników



Rysunek 2.1: schemat zależności między modułami programu

albo wykona się cała transakcja, albo nie wykona się żadna jej instrukcja. Poprawność wykonywania instrukcji zapewnia *menedżer transakcji*, który ponadto jest odpowiedzialny za szereg dodatkowych funkcji:

- wysyła polecenia do *menedżera dziennika* w celu zamieszczenia odpowiednich danych w dziennikach
- zapewnia równoległe przetwarzanie transakcji (szeregowanie)

Schemat zależności między modułami programu jest przedstawiony na rysunku 2.1.

Zakładamy, że baza danych składa się z pewnych „elementów". Elementy te można utożsamiać z różnymi bytami. Mogą to być: krotki relacji lub obiekty, relacje lub klasy, albo bloki dysków i strony. W celu pokazania mechanizmów, w dalszej części tego rozdziału będę utożsamiać elementy bazy danych z liczbami całkowitymi. W części implementacyjnej (rozdział 4) będę je utożsamiał głównie ze stronami na dysku, choć pewnych przypadkach z fragmentami stron.

Z bazą danych jest związany stan, określony przez wartości jej elementów. Zakładamy, że te stany są spójne gdy spełniają wszystkie więzy określone w schemacie bazy danych – zarówno jawne (np. klucze, więzy) jak i niejawne, które założył projektant.

Podstawowe założenia o transakcjach:

Zasada poprawności. *Jeśli spełnione są następujące warunki:*

- *transakcję wykonuje się osobno*
- *nie wystąpią żadne błędy przetwarzania*
- *transakcja zaczyna się w spójnym stanie*
- *oraz transakcja nie narusza więzów ukrytych*

to po zakończeniu transakcji stan bazy danych jest także spójny.

Ostatnie założenie jest właściwie założeniem natury bezpieczeństwa systemu – nie dajemy modyfikować użytkownikom ukrytych więzów spójności, chyba że mają prawo o nich decydować.

Dodatkowo:

1. Transakcje są *atomowe* – to znaczy albo wykonują się w całości, albo w ogóle się nie wykonują.

2. Transakcje wykonywane jednocześnie mogą generować stan niespójny bazy danych, chyba że zadba się o odpowiednią kolejność wykonywania operacji składowych[2].

Na rysunku 2.1 przedstawiony jest schemat współdziałania *menedżera transakcji* z bazą danych. Dane są najpierw wczytywane do buforów, skąd są pobierane do przestrzeni adresowej transakcji. Transakcja wykonuje operacje na nich i może dokonać zmiany ich wartości. Wartości te są następnie zapisywane do buforów, skąd mogą być pobrane i zmienione przez inną transakcję. W międzyczasie wartości buforów mogą (choć nie muszą) zostać zapisane na dysk – decyzja ta jest podejmowana przez *menedżera buforów*. Użycie dzienników polega na wymuszeniu aby zapisy odpowiednich wartości były przeprowadzane w odpowiednich momentach.

W celu prześledzenia algorytmów obsługi dzienników chciałbym wprowadzić następującą notację, opisującą operacje pierwotne w ramach transakcji:

INPUT(X) – kopiowanie bloku dysku zawierającego element X do bufora;

READ(X, t) – kopiowanie elementu X z bufora do przestrzeni adresowej transakcji do zmiennej lokalnej t; jeżeli nie ma wartości X w buforach, wówczas zostanie najpierw wykonane INPUT(X);

WRITE(X, t) – kopiowanie elementu X z przestrzeni adresowej transakcji ze zmiennej lokalnej t do bufora;

OUPUT(X) – kopiowanie bufora zawierającej X na dysk;

Przedstawione operacje pierwotne mają sens jeżeli element bazy danych jest mniejszy niż jeden blok na dysku.

Przykład 2.2.1 *Transakcja T składa się z dwóch kroków:*

1. $A := A * 2$;

2. $B := B * 2$;

Wykonanie transakcji T obejmuje przeczytanie A i B z dysku i przekazanie ich wartości do lokalnej przestrzeni adresowej transakcji, wykonanie mnożenia i zapisanie nowych wartości do buforów. Kolejne kroki transakcji zostały przedstawione w tabeli 2.1.

Jeżeli awaria nastąpi przed wykonaniem $OUTPUT(A)$ lub po wykonaniu $OUTPUT(B)$ to baza danych na dysku pozostaje spójna. Natomiast błąd pomiędzy $OUTPUT(A)$ i $OUTPUT(B)$ spowoduje rozspójnienie bazy danych.

[2]W swojej pracy zakładam, że zadba o to *menedżer transakcji*

Tabela 2.1: Kroki transakcji i ich wpływ na pamięć i dysk

operacja	t	Bufor(A)	Bufor(B)	Dysk(A)	Dysk(B)
READ(A, t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)		16	16	16	8
OUTPUT(B)		16	16	16	16

2.3. Dzienniki wycofań i powtórzeń

Dziennik stanowi ciąg zapisów, w którym każdy mówi o kolejnych zdarzeniach w transakcji. Czynności transakcji mogą się przeplatać, co uniemożliwia zapis historii transakcji po jej zakończeniu. Dzienniki (*logs*) pomagają zapewnić, że transakcje są zdarzeniami atomowymi. Podczas wykonywania transakcji rejestrujemy kolejne kroki wykonywanej transakcji. W przypadku awarii i rozspójnienia bazy danych umożliwia to odwrócenie zmian i przywrócenie stanu spójności.

Nieuszkodzone dzienniki w połączeniu z kopią archiwalną, umożliwiają także odtworzenie aktualnej wersji bazy danych sprzed awarii.

Na ogół, aby usunąć efekty awarii należy niektóre transakcje wykonać powtórnie (dzienniki powtórzeń[3]), a inne unieważnić (dzienniki wycofań[4]). Oba te podejścia mają pewne wady. Rejestrowanie wycofań wymaga zapisu danych natychmiast po zakończeniu transakcji, co może zwiększyć średnią liczbę dostępów do dysku. Rejestrowanie powtórzeń wymaga natomiast, aby wszystkie zmodyfikowane bloki przechowywać w buforach do momentu zatwierdzenia transakcji, co może zwiększyć średnią ilość potrzebnej pamięci. Obie metody mogą spowodować powstanie sprzecznych wymagań podczas punku kontrolnego. Więcej na ten temat w pracy [GMUW06].

W rozdziale 2.3.2 została opisana metoda rejestrowania wycofań i powtórzeń, która jest bardziej elastyczna, wymaga jednak przechowywania większej ilości danych w dziennikach.

2.3.1. Rekordy dziennika

Dziennik możemy sobie wyobrazić jako plik, do którego można dodawać (tylko) kolejne rekordy. Każdy rekord jest odpowiedzialny za inny typ zdarzenia podczas przetwarzania transakcji. Na potrzeby tej pracy będziemy stosować następujące typy rekordów:

$< START\ T >$ – ten rekord oznacza rozpoczęcie transakcji T;

$< COMMIT\ T >$ – oznacza, że transakcja T jest zakończona pomyślnie i nie będzie zmieniać bazy danych; wszystkie zmiany są zapamiętane na dysku;

$< ABORT\ T >$ – transakcja T nie powiodła się

[3] *ang. redo logs*
[4] *ang. undo logs*

Tabela 2.2: Kroki transakcji i ich wpływ na pamięć, dysk i dzienniki

operacja	t	Bufor(A)	Bufor(B)	Dysk(A)	Dysk(B)	Dziennik
						$< START\ T >$
READ(A, t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A, t)	16	16		8	8	$< T,\ A,\ 8,\ 16 >$
READ(B, t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	$< T,\ B,\ 8,\ 16 >$
FLUSH LOG						
OUTPUT(A)		16	16	16	8	
						$< COMMIT\ T >$
OUTPUT(B)		16	16	16	16	

$< T,\ X,\ v,\ w >$ – rekord aktualizacji: T - transakcja, X - identyfikator elementu, v - poprzednia wartość, w - nowa wartość; zawartość rekordu aktualizacji jest zależna od typu prowadzonego dziennika (por. [GMUW06]).

Pozostałe typy rekordów używane w dziennikach zostały opisane w rozdziale 2.3.2 i rozdziale 2.3.4. Podsumowanie wszystkich typów zastosowanych w implementacji zostało opisane w dodatku B.

2.3.2. Rejestrowanie wycofań i powtórzeń

Założenia które musi spełniać system rejestrowania zawierają się w następującej zasadzie:

Reguła 2.3.1 *Zanim na dysku zapisze się zmianę wartości elementu X, spowodowaną działaniem pewnej transakcji T, należy najpierw na dysk wprowadzić rekord aktualizacji* $< T,\ X,\ v,\ w >$[5].

Przykład 2.3.1 *W tabeli 2.2 przedstawiono kolejne kroki, które są wykonywane przy przetwarzaniu transakcji z przykładu 2.2.1. Pojawiła się tutaj także dodatkowa operacja o nazwie* FLUSH LOG*. Operacja ta jest wykonywana przez* menedżera transakcji *i ma na celu zasygnalizowanie* menedżerowi dzienników*, że należy zapisać zawartość buforów na dysk.*

Operacja dodania wartości $< COMMIT\ T >$ występuje w przykładzie po operacji *FLUSH LOG*. Oznacza to, że system zostawia dowolność co do wyboru momentu zapisu tego rekordu na dysk. Ten fakt ma trzy podstawowe konsekwencje:

- brak wymuszenia zapisu, zmniejsza średnią ilość zapisów na dysku
- może dojść do sytuacji w której użytkownik otrzyma informację o zatwierdzeniu transakcji, a rekord $< COMMIT\ T >$ nie będzie jeszcze zapisany na dysku; jeżeli w tym czasie nastąpi awaria, wówczas transakcja zostanie wycofana; można temu przeciwdziałać poprzez wymuszenie *FLUSH LOG* zaraz po $< COMMIT\ T >$

[5] [GMUW06] strona 922.

- wymusza w procesie odtwarzania wykonanie dwóch kroków (por. 2.3.3) ponieważ może wystąpić sytuacja w której zatwierdzone transakcje i zmiany bazy danych nie będą przechowywane na dysku, albo będą przechowywane na dysku pewne transakcje niezatwierdzone wraz z wprowadzonymi przez nie zmianami

2.3.3. Odtwarzanie za pomocą dziennika wycofań i powtórzeń

Podczas odtwarzania w rekordach aktualizacji dziennika mamy informacje umożliwiające zarówno wykonanie jak i wycofanie transakcji. Metoda odtwarzania jest dwufazowa i polega na:

- powtórzeniu wszystkich transakcji zatwierdzonych, zaczynając od najwcześniejszej
- wycofaniu wszystkich transakcji niezakończonych, zaczynając od najpóźniejszej

Przykład 2.3.2 *W przykładzie 2.3.1 odtwarzanie stanu bazy będzie zależne od momentu w którym nastąpiła awaria.*

- *Jeżeli rekord $< COMMIT\ T >$ został zapisany na dysk, wówczas transakcja zostanie uznana za zatwierdzoną. Oznacza to, że obu elementom na dysku należy przypisać wartość 16.*
- *Jeżeli awaria nastąpiła zanim rekord $< COMMIT\ T >$ trafił na dysk, wówczas transakcja zostanie uznana za niezatwierdzoną. Oznacza to, że należy wartości A i wartości B przypisać wartość 8.*

Awaria podczas odtwarzania

Jeżeli podczas odtwarzania stanu sprzed awarii system ulegnie awarii, wówczas należy rozpocząć odtwarzanie bazy danych od początku. Sytuacja taka jest możliwa, ponieważ odtwarzanie jest operacją idempotenta i przy powtórnym odtwarzaniu nie bierze się pod uwagę zmian wykonanych przy poprzednim odtwarzaniu. Wszystkie potrzebne wartości są i tak zapisane w dziennikach i przy odtwarzaniu nie korzysta się z wartości z bazy danych, wiec ich wartość jest bez znaczenia.

2.3.4. Punkty kontrolne w dzienniku wycofań i powtórzeń

Przy odtwarzaniu należy przejrzeć cały dziennik, co w przypadku bazy danych obsługującej dużo transakcji może być czasochłonne. W przypadku przetwarzania przez system pojedynczych transakcji, można by po wprowadzeniu rekordu $< COMMIT\ T >$ (lub lepiej tuż przed jego wprowadzeniem) czyścić cały dziennik, ponieważ dane dotyczące zatwierdzonych transakcji i zapisanych w bazie nie są potrzebne. Niestety w sytuacji przetwarzania kilku transakcji jednocześnie, takie działanie mogłoby spowodować usunięcie wpisów w dzienniku, dotyczących niezakończonych transakcji.

W celu uniknięcia takiej sytuacji można wprowadzić punkty kontrolne. Mechanizm wprowadzania punktów kontrolnych wygląda następująco:

- wstrzymanie akceptowania transakcji
- oczekiwanie na zakończenie rozpoczętych transakcji ($COMMIT$ lub $ABORT$)
- zapisanie dziennika

Tabela 2.3: Wprowadzenie punktów kontrolnych do dzienników

operacja	mechanizm punktów kontrolnych
$< START\ T_1 >$	
$< T_1,\ A,\ 4,\ 8 >$	
$< START\ T_2 >$	
	wstrzymanie akceptowania transakcji
$< T_2,\ B,\ 15,\ 16 >$	oczekiwanie na zakończenie rozpoczętych transakcji
$< T_2,\ C,\ 23,\ 42 >$	
$< T_1,\ D,\ 4,\ 8 >$	
$< COMMIT\ T_1 >$	
$< COMMIT\ T_1 >$	
	zapisanie dziennika
$< CKPT >$	dopisanie rekordu $< CKPT >$ i zapisanie dziennika
	wznowienie przetwarzania transakcji
$< START\ T_3 >$	
$< T_3,\ A,\ 15,\ 16 >$	
$< T_3,\ A,\ 23,\ 42 >$	
[...]	

- dopisanie rekordu $< CKPT >$ i zapisanie dziennika
- wznowienie przetwarzania transakcji

Dzięki zastosowaniu tej techniki możemy pominąć wszystkie wpisy, które znajdują się przez rekordem $< CKPT >$.

Przykład 2.3.3 *Tabela 2.3 obrazuje kolejne kroki przetwarzania transakcji i zawartość dzienników.*

2.3.5. Niespoczynkowe punkty kontrolne

Metoda opisana w punkcie 2.3.4 ma niestety pewną poważną wadę, polegającą na tym, że należy zatrzymać pracę systemu, na czas oczekiwania na zakończenie transakcji, co może trwać dość długo i niektórych zastosowaniach może być niedopuszczalne.

W związku z tym należy zastosować metodę niespoczynkowych punktów kontrolnych polegającą, której schemat przedstawiono poniżej:

1. Wprowadzenie do dziennika rekordu $< START\ CKPT(T_1,\ \ldots, T_k) >$ i zapisanie na dysku. $T_1,\ \ldots, T_k$ oznaczają transakcje aktywne.
2. Przetwarzamy normalnie transakcje oczekując zakończenia transakcji $T_1,\ \ldots, T_k$.
3. Gdy wszystkie transakcje z listy zostaną zakończone, dodanie do dziennika rekordu $< END\ CKPT >$ i zapisanie na dysku.

Szczególnym przypadkiem jest sytuacja, gdy system nie przetwarza żadnych transakcji i wówczas *menedżer dziennika* może wymusić zapisanie buforów na dysk. Do dziennika trafia $< START\ CKPT() >$ a zaraz potem $< END\ CKPT >$. Dla uproszczenia, taką sytuację wyróżnia się pojedynczym wpisem w postaci $< CKPT >$.

Korzystając z tego typu dziennika, po awarii postępować należy jak opisane zostało w punkcie 2.3.3. W zależności od tego który z rekordów, $< START\ CKPT(T_1,\ \ldots, T_k) >$ czy $< END\ CKPT >$, zostanie pierwszy odnaleziony, można wyróżnić dwa przypadki.

Jeżeli pierwszym odnalezionym rekordem jest $< END\ CKPT >$ oznacza to, że wszystkie niezakończone transakcje zaczęły się po poprzednim rekordzie $< START\ CKPT(T_1,\ \ldots, T_k) >$. W związku z tym wystarczy sprawdzać rekordy aż do momentu odnalezienia rekordu $< END\ CKPT >$ i zakończyć.

Jeżeli pierwszym rekordem jest $< START\ CKPT(T_1,\ \ldots, T_k) >$ oznacza to, że awaria nastąpiła w tym przedziale kontrolnym, czyli jedynymi transakcjami niezakończonymi są transakcje ze zbioru $T_1,\ \ldots, T_k$, a także wszystkie które napotkano przed rekordem $< START\ CKPT(T_1,\ \ldots, T_k) >$. W związku z tym należy sprawdzić dziennik wstecz wyłapując tylko transakcje z tej listy.

2.3.6. Algorytm odtwarzania

Poniżej został opisany dokładny algorytm odzyskiwania zawartości bazy danych na podstawie dziennika, z uwzględnieniem niespoczynkowych punktów kontrolnych. Algorytm jest zmodyfikowaną wersją z pracy [LBRS04], która nie uwzględnia dwóch przypadków w zależności czy do dziennika został zapisany rekord $< END\ CKPT >$ czy nie. W efekcie program przetwarzał, niepotrzebnie, dłuższy fragment dziennika.

W celu wykonania algorytmu potrzebujemy listę transakcji i ich statusu („do powtórzenia" lub „do wycofania") zmienną *o*ffset do zapamiętania położenia początkowego.

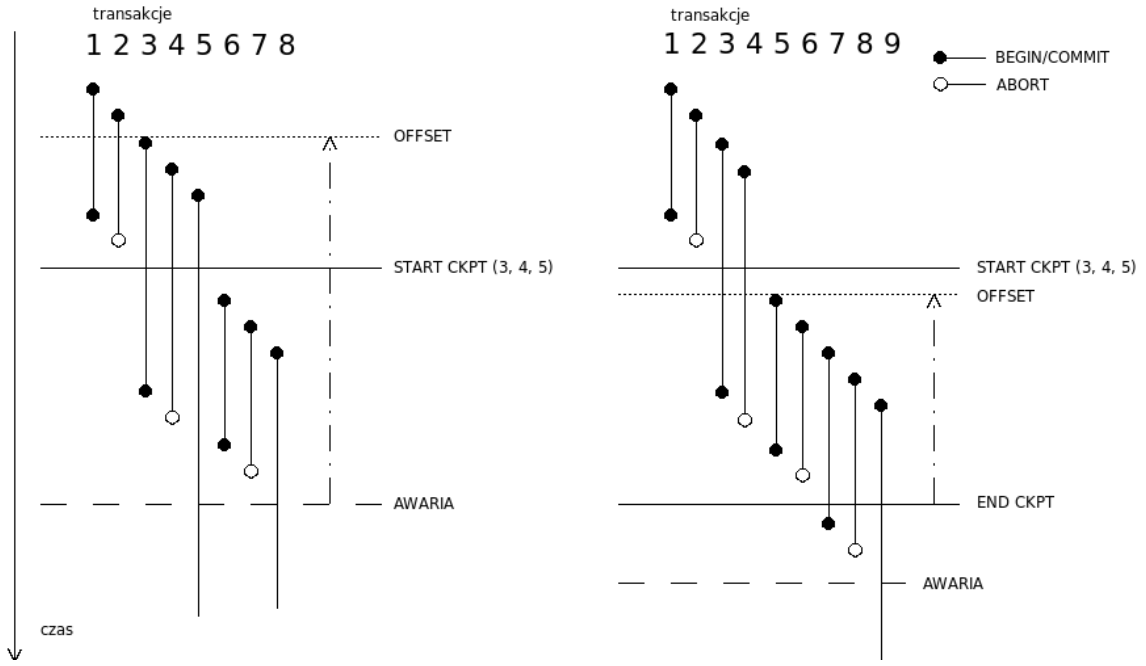
1. Celem pierwszego kroku algorytmu jest znalezienie pozycji startowej w dzienniku, od której zaczniemy odtwarzanie danych. W tym celu czytamy dziennik od końca:

   - jeżeli wystąpi $< END\ CKPT >$ to zapamiętujemy to zdarzenie i czytamy dalej
   - jeżeli wystąpi $< START\ CKPT(T_1,\ \ldots, T_k) >$, ale nie wystąpił wcześniej rekord $< END\ CKPT >$ oznacza to, że musimy się cofnąć aż do momentu rozpoczęcia pierwszej[6] transakcji z listy $T_1,\ \ldots, T_k$; zapamiętujemy położenie tego rekordu i kończymy przeglądanie
   - jeżeli wystąpi $< START\ CKPT(T_1,\ \ldots, T_k) >$, ale wystąpił wcześniej rekord $< END\ CKPT >$, oznacza to miejscem od którego rozpoczniemy analizę jest koniec rekordu $< START\ CKPT(T_1,\ \ldots, T_k) >$ i kończymy przeglądanie
   - jeżeli wystąpi jeden z rekordów: $< START >$, $< STOP >$, $< CKPT >$ lub dojdziemy do początku pliku, to zapamiętujemy to położenie i kończymy przeglądanie

2. Celem drugiego kroku jest podział transakcji na transakcje do powtórzenia (redo) i do wycofania (undo). W tym celu przeglądamy dziennik począwszy od miejsca znalezionego w kroku pierwszym aż do końca pliku. Jeżeli znajdziemy:

   - rekord typu $< START; CKPT(T_i) >$ to dodajemy do listy ze statusem „do wycofania"
   - rekord typu $< COMMIT\ CKPT(T_i) >$ to zmieniamy status tej transakcji na „do powtórzenia"
   - pozostałe rekordy ignorujemy

[6] w sensie czasu rozpoczęcia



Rysunek 2.2: Przykład działania algorytmu – odszukiwanie miejsca początkowego (*offset*).

3. Celem trzeciego kroku jest wycofanie odpowiednich transakcji. W tym celu wracamy do miejsca zapamiętanego w pierwszym kroku i zaczynamy przeglądać dziennik. Jeżeli znajdziemy rekord typu $< UPDATE\ T_i, pid^7, OLD^8, NEW^9 >$ i T_i jest na liście ze statusem „do wycofania" to zapisujemy wartość OLD do strony pid.

4. Czwarty krok jest analogiczny do trzeciego z tą różnicą, że teraz należy powtórzyć transakcje. W tym celu wracamy do miejsca zapamiętanego w pierwszym kroku i zaczynamy przeglądać dziennik. Jeżeli znajdziemy rekord typu $< UPDATE\ T_i, pid, OLD, NEW >$ i T_i jest na liście ze statusem „do powtórzenia" to zapisujemy wartość NEW do strony pid.

Dwa przykłady zastosowania tego algorytmy przedstawiono na rysunku 2.2. Przerywana linia zakończona strzałką określa przejście algorytmu po pliku, aż do znalezienia początku danych do interpretacji (*offset*).

2.3.7. Wydajność

Podstawową zaletą tego algorytmu jest proporcjonalnie niewielka ilość używanej pamięci, gdyż jedyną strukturą danych o zmiennej wielkości jest lista transakcji, która ma rozmiar $sizeof(int)+sizeof(bool) = 8b+1b = 9b$[10], czyli jeden megabajt pamięci pozwala przechować informację o ponad 116 tysiącach transakcji.

[7] identyfikator strony – właściwie składa się z identyfikatora pliku i identyfikatora strony

[8] stara wartość

[9] nowa wartość

[10] Wielkość struktury zajętej przez strukturę jest tak naprawdę zależna od maszyny, kompilatora i jego ustawień. W kompilatorach na procesor x86 występuje wyrównanie naturalne struktury, co w efekcie powoduje że faktyczny rozmiar w pamięci wynosi $sizeof(int) + sizeof(bool) = 8b + 8b = 16b$. Na maszynach typu ILP64 ta sama struktura będzie miała rozmiar $128b$, a w systemie w 128-bitowym ZLS aż $256b$.[CV04]

Wadą algorytmu jest natomiast potrzeba 4-krotnego przeglądania pliku w tym raz „od końca", który to proces jest dość wolny.

2.4. Podsumowanie

Metoda rejestrowania powtórzeń i wycofań jest najbardziej elastyczna (por. [GMUW06]) ponieważ wymaga tylko żeby rekord aktualizacji znalazł się na dysku zanim zostanie zapisana faktyczna modyfikacja w bazie danych. Nie ma natomiast ograniczeń dotyczących momentu zapamiętania rekordu zatwierdzenia transakcji ($< COMMIT >$). Poza tym metoda ta w łatwy sposób może zostać wykorzystana przy tworzeniu archiwum w sposób niespoczynkowy (patrz 3.2).

Odtwarzanie natomiast polega na powtórzeniu transakcji zatwierdzonych i wycofaniu niezatwierdzonych.

Podstawową wadą tej metody jest potrzeba rejestrowania wartości danych przed i po zmianie co może powodować szybkie powiększanie się plików dzienników, co z kolei wymaga odpowiedniego ich przycinania i skracania.

Rozdział 3

Archiwum

Opisane wcześniej dzienniki zabezpieczają system przed awariami systemu, ale tylko takimi w wyniku których tracona jest zawartość pamięci operacyjnej, natomiast dane na dysku – zarówno baza danych jak i dzienniki pozostają nienaruszone.

Niestety może się zdarzyć, że uszkodzeniu ulegnie nośnik fizyczny (dysk, macierz dyskowa) niszcząc bazę danych. Idąc za [GMUW06], istnieje możliwość odzyskania bazy danych na podstawie dziennika jeśli:

- dziennik nie uległ zniszczeniu, np. był na innym nośniku
- dziennik nie był niszczony po punkcie kontrolnym
- dziennik zawierał „nowe" wartości więc był typu „powtarzania" lub „wycofania i powtarzania"

Taki dziennik jednak zwiększa się szybciej niż baza danych, a więc nie ma sensu go przechowywać w nieskończoność.

W związku z tym stosuje się rozwiązania oparte o archiwum – czyli utrzymywanie oddzielnej kopii bazy danych w tzw. bezpiecznym miejscu. Kopia musi mieć następującą właściwość: odzyskana baza danych musi być spójna.

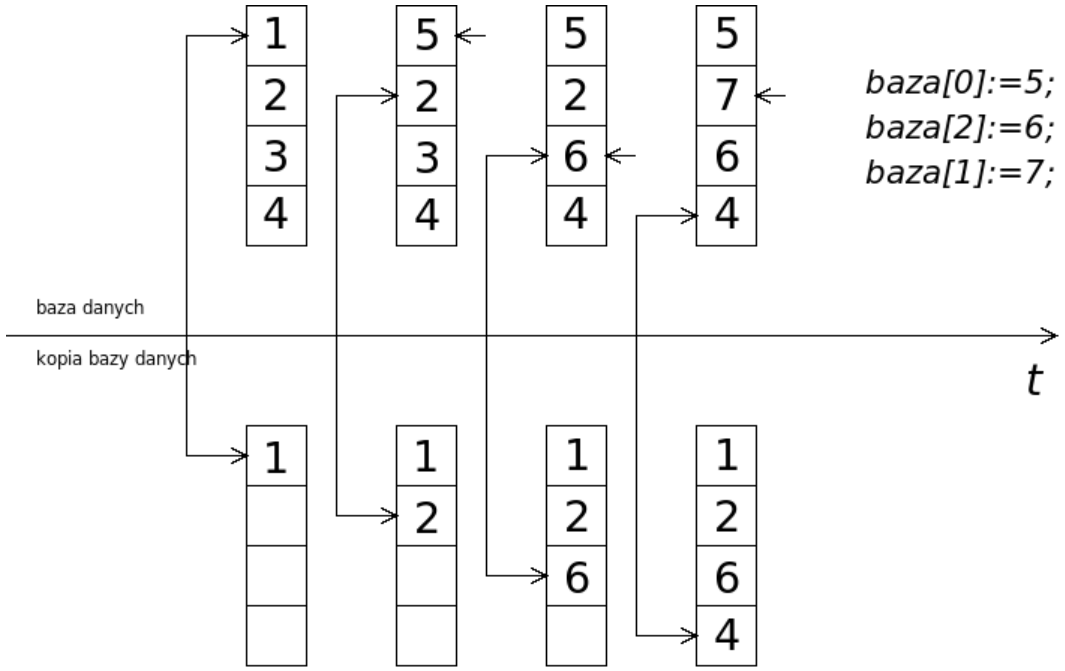
W praktyce sposoby tworzenia kopii archiwalnej dzielimy na te, które wymagają wyłączenia bazy danych (spoczynkowa) i te, które tego nie wymagają (niespoczynkowa).

3.1. Archiwizacja spoczynkowa

Przykładem archiwizacji spoczynkowej jest np. przegranie wszystkich plików bazy danych na inny nośnik. Taką kopię możemy wykonać podczas gdy baza danych jest nieaktywna. Jeżeli po wykonaniu takiego archiwum, baza danych ulegnie uszkodzeniu, wówczas będzie można odzyskać bazę danych z chwili wykonania kopii.

Aby przywrócić bardziej aktualny stan można zadbać żeby od czasu wykonania kopii zapasowej dziennik nie był niszczony i przetrwał awarię. Można także co pewien czas przesyłać aktualną wersję dziennika do bezpiecznej lokalizacji.

Jeżeli baza danych jest duża, wówczas wykonanie kopii zapasowej może trwać bardzo długo. Chcąc uniknąć takiej sytuacji można wykonywać kopię przyrostową – kopiować różnice od czasu wykonania ostatniej pełnej kopii lub ostatniej kopii przyrostowej. Wówczas można odtworzyć bazę danych na podstawie pełnego zrzutu, a następnie uaktualnić na podstawie kopii przyrostowych.



Rysunek 3.1: Przykład niespoczynkowej archiwizacji.

3.2. Archiwizacja niespoczynkowa

Metody archiwizacji opisane w punkcie 3.1 wymagają zamykania bazy danych na czas tworzenia kopii, co w przypadku pełnej kopii może nawet trwać wiele godzin.

Archiwizacja niespoczynkowa przypomina trochę tworzenie niespoczynkowych punktów kontrolnych opisanych w punkcie 2.3.5. Próbujemy tworzyć kopię bazy danych, ale w trakcie zrzutu (który może trwać wiele godzin) akceptujemy wszelkie zmiany które zachodzą w bazie i notujemy je w dzienniku.

Jeżeli trzeba będzie odtworzyć bazę danych po awarii, wówczas będziemy potrzebować wykonaną przez nas kopię, a także dziennik zmian które w niej zaszły lub powinny zajść do momentu skończenia zrzutu. Najpierw należało będzie przegrać bazę danych z powrotem, a następnie uaktualnić ją za pomocą dziennika.

Rysunek 3.1 zawiera przykład bazy danych zawierającej 4 elementy. Baza danych jest archiwizowana niespoczynkowo, a aktualne położenie procesu archiwizującego w pliku jest oznaczone strzałką w lewo. Podczas archiwizacji zachodzą zmiany w bazie danych wynikające z przetwarzania transakcji – zapisy w bazie danych są oznaczone strzałką w prawo.

Po zakończonej archiwizacji występują różnice pomiędzy aktualnym stanem bazy danych, a jej kopią. W szczególności jeżeli wszystkie zmiany zostały przeprowadzone w ramach jednej transakcji, to baza danych jest niespójna. Przed przywróceniem, zmiany które zawiera dziennik[1], znajdujący się w prawym górnym rogu rysunku, powinny zostać wprowadzone do bazy danych.

[1]jest to dziennik powtórzeń, ale oczywiście implementacja dziennika typu powtórzeń i wycofań jest też dobra jako że zawiera potrzebne dane

Tabela 3.1: Tworzenie kopii archiwalnej

operacja	mechanizm archiwizacji
$< START\ DUMP >$	
$< START\ CKPT(T_1, T_2) >$	
$< T_1, A, 1, 5 >$	rozpoczęcie kopiowania
$< T_2, C, 1, 5 >$	
$< COMMIT\ T_2 >$	
$< T_1, B, 2, 7 >$	
$< END\ CKPT >$	
[...]	zakończenie kopiowania
$< END\ DUMP >$	
[...]	

3.2.1. Tworzenie archiwum

Algorytm tworzenia kopii zapasowej bez zatrzymywania przetwarzania transakcji w bazie danych został opracowany na podstawie [GMUW06] i jest poprawny przy założeniu, że dzienniki są „powtórzeń" lub „powtórzeń i wycofań". W celu wykonania kopii należy:

1. wprowadzić do dziennika rekord $< START\ DUMP >$ oznaczający rozpoczęcie archiwizacji
2. zastosować punkt kontrolny typu $< START\ CKPT(T_1,\ \ldots, T_k) >$
3. wykonać zrzut
4. skopiować wystarczająco duży fragment dziennika niezbędny do odtworzenia bazy danych do bezpiecznej lokalizacji
5. wprowadzić do dziennika rekord $< END\ DUMP >$

Przykład 3.2.1 *Poniższy przykład obrazuje kolejne kroki tworzenia kopii archiwalnej i zawartość dzienników, w sytuacji przedstawionej na rysunku 3.1*

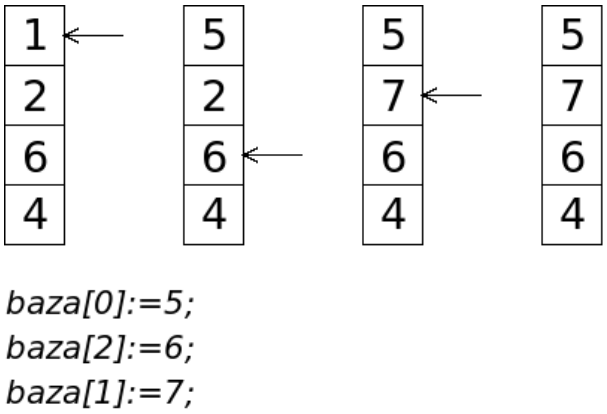
Pozostaje kwestia jaki jest najmniejszy fragment dziennika wystarczający do odzyskania bazy danych. Fragment taki powinien zaczynać się od punktu kontrolnego wstawionego w punkcie 2 a kończyć się w momencie zakończenia zrzutu.

Oczywiście w ten sposób wszystkie transakcje ze zbioru $T_1,\ \ldots, T_k$, które były aktywne w momencie tworzenia punktu kontrolnego w punkcie 2 będą musiały być wycofane. Przesunięcie początku zarchiwizowanego dziennika, do momentu rozpoczęcia się pierwszej[2] transakcji z tego zbioru umożliwia odtworzenie zatwierdzonych transakcji z tego zbioru.

Koniec dziennika możemy natomiast przesuwać w nieskończoność – bo czym większy fragment zostanie zarchiwizowany tym bardziej aktualna będzie baza danych po odtworzeniu. W przypadku tworzenia kopii archiwalnej ważnym argumentem jest skrócenie czasu wykonania kopii.

Podsumowując, z najlepszym miejscem zakończenia archiwizowania dziennika jest: najmniejszy zrzut obejmujący rekord $< END\ DUMP >$ i rekord $< END\ CKPT >$ który kończy punkt kontrolny utworzony w punkcie 2 algorytmu.

[2] w sensie czasu rozpoczęcia



Rysunek 3.2: Przykład odtwarzania archiwum uzyskanego metodą niespoczynkowej archiwizacji.

3.3. Odtwarzanie za pomocą archiwum i dziennika

Odtwarzanie za pomocą kopii bazy danych i dziennika powstałego w wyniku stosowania metody z punktu 3.2.1 jest dwufazowe:

1. najpierw należy odtworzyć (skopiować) bazę danych i dziennik z archiwum
2. wykonać algorytm odtwarzania z punktu 2.3.6

Przykład 3.3.1 *W przykładzie 3.2.1 wykonano kopię archiwalną na działającej bazie danych. Kopia ta składała się z dwóch elementów: bazy danych o umownej wartości* $< 1, 2, 6, 4 >$ *i dziennika w postaci rekordów* $< baza[0] := 5, baza[2] := 6, baza[1] := 7 >$.

Rysunek 3.2 przedstawia zastosowanie algorytmu z punktu 2.3.6 w celu odzyskania kopii archiwalnej czyli powtórzenie transakcji na istniejącej bazie.

Dzięki temu uzyskujemy początkową bazę danych z przykładu 3.2.1.

Rozdział 4

Implementacja

W trakcie implementacji postawiono szczególny nacisk na niezależność modułu *Logs* od innych modułów, chcąc w ten sposób umożliwić zastosowanie modułu w innym silniku bazodanowym lub ewentualną łatwą wymianę modułu w przyszłości. Do sprawdzania poprawności kodu użyto modułu `assert.h`, dodając asercje w krytycznych miejscach kodu.

Implementacja została wykonana w następujących etapach:

1. utworzenie modułu Logs (punkt 4.1)
2. tworzenie dziennika z punktami kontrolnymi (punkt 4.2)
3. odtwarzanie bazy danych na podstawie dzienników (punkt 4.3)

4.1. Dziennik

Podczas projektowania modułu, najważniejszą decyzją jest wybranie „elementu" do rejestrowania w dziennikach. Element ten musi być dostatecznie duży żeby cała zmiana obiektu była w nim zapisana i jednocześnie dostatecznie mały żeby trzymanie dzienników zmian nie było zbyt kosztowne. Co ciekawe element nie może być za duży – porównaj przykład 4.1.1.

Element można wybrać zarówno na poziomie logicznym (np. wartości atrybutów) lub fizycznym (np. strony, fragmenty plików, całe pliki).

Implementacja na poziomie fizycznym umożliwia zachowywanie zmienianych elementów bazy danych przez transakcje nie zastanawiając się co tak naprawdę zmieniane jest na poziomie logicznym (dane, indeksy czy struktury pomocnicze itp.). Z drugiej strony wymaga to od *menedżera dysku*[1] aby każda modyfikacja bazy danych była opatrzona informacją która transakcja dokonuje zmiany – identyfikator transakcji (*TransactionId*) musi być przekazywany za każdym razem z *menedżera transakcji* do strony dysku[2].

4.1.1. Blokowanie

Pierwotnie utożsamiono elementy bazy danych z stronami na dysku, chcąc w ten sposób uniknąć poważnych problemów z rejestrowaniem zmian w dzienniku i transakcjami. Niestety wybór ten okazał się niewłaściwy.

Baza danych LoXiM składa się z kilku plików o identycznym podziale na strony. Różne pliki wykorzystują różne mechanizmy blokowania stron lub ich fragmentów przez *menedżera*

[1] ang. *Store*
[2] ang. *Page*

transakcji. Plik zawierający dane (STORE_FILE_DEFAULT) jest blokowany na poziomie strony dysku, natomiast plik zawierający mapę (STORE_FILE_MAP) jest blokowany na poziomie fragmentu strony. Oznacza to, że dwie różne transakcje mogą modyfikować tą samą stronę.

Przykład 4.1.1 *Załóżmy, że strona ma 4 bajty i że pojedyncza strona jest elementem zapisywanym w dzienniku. W nawiasie pokazana jest zawartość strony na dysku. Następujący przebieg zdarzeń:*

operacja	***strona*** s_1	***rekord dodany do dziennika***
rozpoczęcie transakcji T_1	*0 0 0 0*	$< BEGIN\ T_1 >$
rozpoczęcie transakcji T_2	*0 0 0 0*	$< BEGIN\ T_2 >$
transakcja T_1 *zmienia bajt 0*	*1 0 0 0*	$< UPDATE\ T_1, s_1, 0\,0\,0\,0, 1\,0\,0\,0 >$
transakcja T_2 *zmienia bajt 1*	*1 2 0 0*	$< UPDATE\ T_2, s_1, 1\,0\,0\,0, 1\,2\,0\,0 >$
transakcja T_1 *jest zatwierdzana*	*1 2 0 0*	$< COMMIT\ T_1 >$

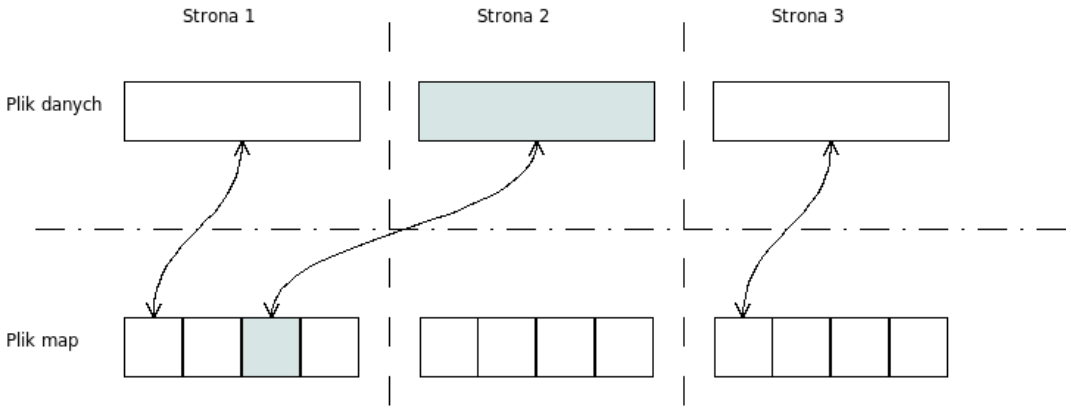
W tym miejscu praca systemu zostaje przerwana. Po ponownym uruchomieniu, przywrócenie spójności bazy będzie polegało na wycofaniu transakcji T_2 *i powtórzeniu transakcji* T_1 *(por. algorytm 2.3.6).*

operacja	***strona*** s_1
(stan początkowy po awarii)	*1 2 0 0*
wycofanie T_1 *na podstawie rekordu aktualizacji*	*0 0 0 0*
powtórzenie T_2 *na podstawie rekordu aktualizacji*	*1 2 0 0*

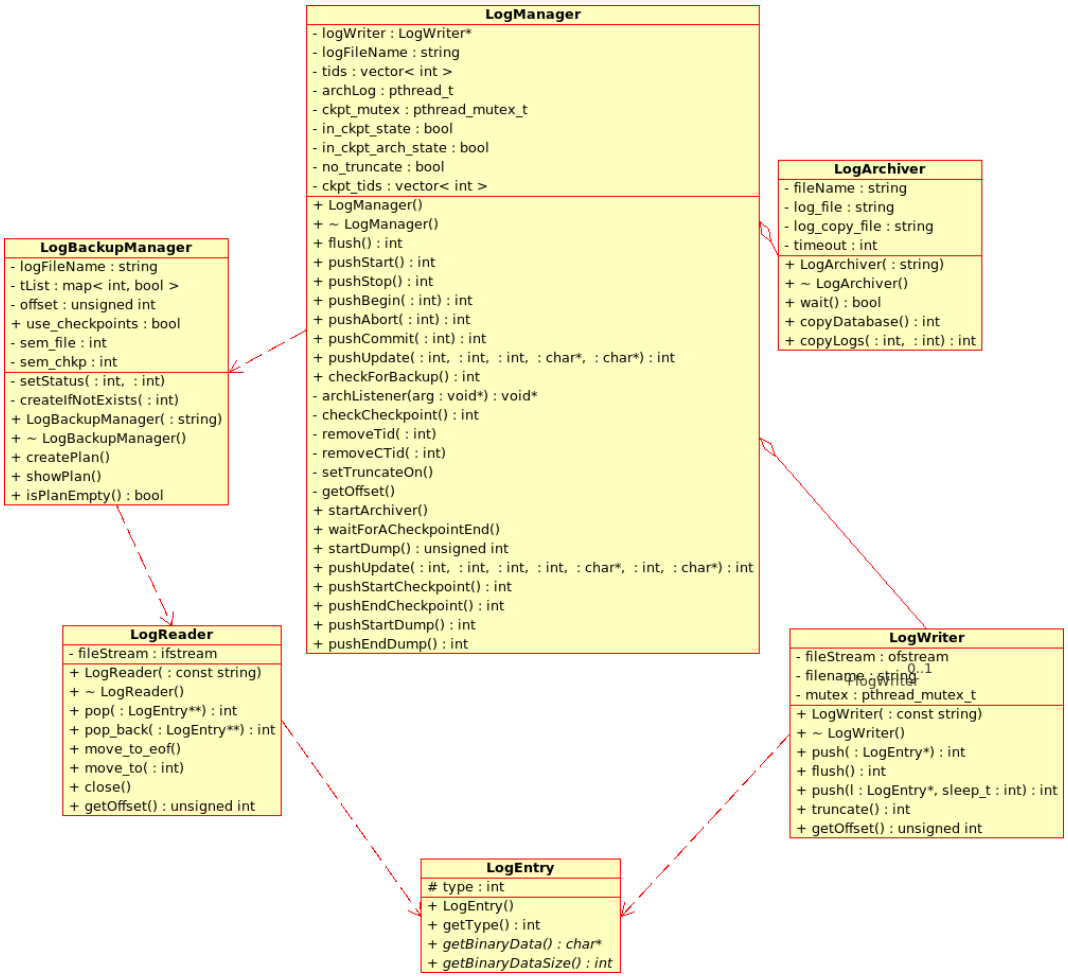
Jak widać wynik jest niewłaściwy, bo zawartość strony powinna wynosić 0200.

Mechanizm blokowania dostępu w pliku mapy jest konsekwencją mechanizmu blokowania w pliku z danymi. Zależności zostały przedstawione na rysunku 4.1. Blokowanie w pliku mapy nie jest wykonywane na poziomie strony lecz na poziomie jej fragmentu. Program LoXiM spełnia własność, że momencie gdy jakaś strona w pliku danych jest blokowana, to jest także blokowany odpowiedni fragment strony w pliku mapy.

Wniosek z przykładu 4.1.1 jest następujący: w przypadku pliku ze stroną, rozmiar elementu równy rozmiarowi strony dysku jest niewłaściwy. Konkretnie należy brać pod uwagę tylko



Rysunek 4.1: Zależność pomiędzy plikiem z danymi a plikiem map.



Rysunek 4.2: Diagram klas dla modułu Logs.

mniejszy fragment – ten który został zmieniony. W implementacji zastosowano rozróżnienie sposobu przechowywania danych w dziennikach w zależności od typu pliku – blokowanie na poziomie strony dysku lub jej fragmentu. Więcej szczegółów zostanie opisane w rozdziale 4.1.5.

W trakcie projektowania modułu Logs położono nacisk na zminimalizowanie interakcji z pozostałymi częściami systemu. Moduł składa się z klas obiektów przedstawionych na rysunku 4.2, a ich opis pojawi się w dalszej części rozdziału.

Podczas implementacji zastosowano testowanie oparte o testy automatyczne. Testy zostały przeprowadzone na poziomie pojedynczych klas i metod w tych klasach, a także testowano całe moduły i ostateczne funkcje. Te ostatnie zostały przetestowane za pomocą testów użytkownika, a nie automatycznych. Dodatkowo skorzystano z systemu asercji i biblioteki `assert.h`.

Komunikacja pomiędzy modułem a resztą systemu odbywa się za pomocą obiektu klasy *LogManager*. W systemie powinien występować jeden obiekt tego typu i cała komunikacja powinna się przez niego.

4.1.2. Klasa LogWriter

Klasa *LogWriter* jest odpowiedzialna za zapisywanie rekordów w pliku na dysku. Poniżej kod źródłowy nagłówka klasy.

```
class LogWriter
{
    private:
        ofstream fileStream;
        string filename;

        pthread_mutex_t mutex;
    protected:
    public:
        LogWriter(const string);
        ~LogWriter();

        int push(LogEntry* l, int sleep_t = 0);
        int flush();
        int truncate();
        unsigned int getOffset();
};
```

Oprócz konstruktora, każda klasa jest wyposażona w następujące metody:

flush() – wywołanie metody powoduje natychmiastowe zapisanie buforów pliku na dysku

truncate() – metoda powoduje przycięcie pliku i przejście na jego początek

getOffset() – wynikiem wywołania metody jest aktualna pozycja w pliku

Metoda **push(LogEntry* l)** jest najważniejszą metodą klasy i jest przeważnie wywoływana przez obiekt typu *LogManager*. Parametr to obiekt do zapisania na dysku. Wynikiem wywołania metody jest położenie w pliku po zapisaniu elementu. Metoda zapisuje obiekt na dysku korzystając z metod *getBinaryDataSize()* i *getBinaryData()*, a następnie zapisuje rozmiar rekordu.

Zapisanie wielkości rekordu na końcu ma na celu umożliwienie łatwego i wydajnego sposobu na przeglądanie plików od tyłu (por. 4.1.3).

Klasa wspiera wielowątkowość tak, że jeden obiekt tej klasy może być wywoływany przez kilka różnych wątków.

4.1.3. Klasa LogReader

Klasa *LogReader* jest odpowiedzialna za czytanie plików dziennika. Czyta plik i przekazuje do programu obiekty wypełnione odpowiednimi, przeczytanymi z pliku wartościami.

Czytanie od początku do końca

Klasa udostępnia dwie metody przeglądania pliku – pierwsza z nich to czytanie od początku do końca. Czytając w ten sposób program wykonuje kolejne kroki:

1. pobiera 4 bajty z pliku, co określa typ zapisanego obiektu
2. dalsze postępowanie jest zależne od typu tego obiektu.
3. po przeczytaniu całego obiektu wczytuje kolejne 4 bajty określające wielkość zapisanego rekordu, co może być wykorzystane do weryfikacji lub zignorowane

4. idź do kroku 1 i powtarzaj aż do końca pliku

Czytanie od końca do początku

Czytając od tyłu program zaczyna czytać od końca wykonując kolejne kroki:

1. cofa się o 4 bajty, wczytuje je i zapamiętuje (zmienna *record_len*) – co oznacza wielkość poprzedniego rekordu

2. cofa się do pozycji *pos* takiej, że $pos := aktualna\_pozycja - record\_len$

3. wczytuje 4 bajty z pliku, co określa typ zapisanego obiektu

4. dalsze postępowanie jest zależne od typu tego obiektu.

5. po przeczytaniu cofa się do pozycji *pos*

6. jeżeli $pos \neq 0$ idź do kroku 1

Właściwie klasa może w dowolnym momencie zacząć czytać od początku bądź od końca, pod warunkiem że aktualna pozycja pliku jest na końcu poprzedniego rekordu, za danymi określającymi jego wielkość.

Proces czytania od tyłu jest oparty o funkcje standardową *tellg()*, której implementacja pod Linuxem nie należy do najwydajniejszych. W efekcie czytanie od tyłu jest procesem relatywnie długotrwałym.

4.1.4. Rodzina klas LogEntry

Na potrzeby wygodnej implementacji wpisy (rekordy) dziennika zostały opakowane w klasy. Wszystkie klasy wywodzą się od klasy *LogEntry* która definiuje interfejs dla całej rodziny. Dokładny diagram klas został przedstawiony na rysunku 4.3. Pozostałe elementy modułu *Logs* przekazują pomiędzy sobą obiektów tych właśnie klas.

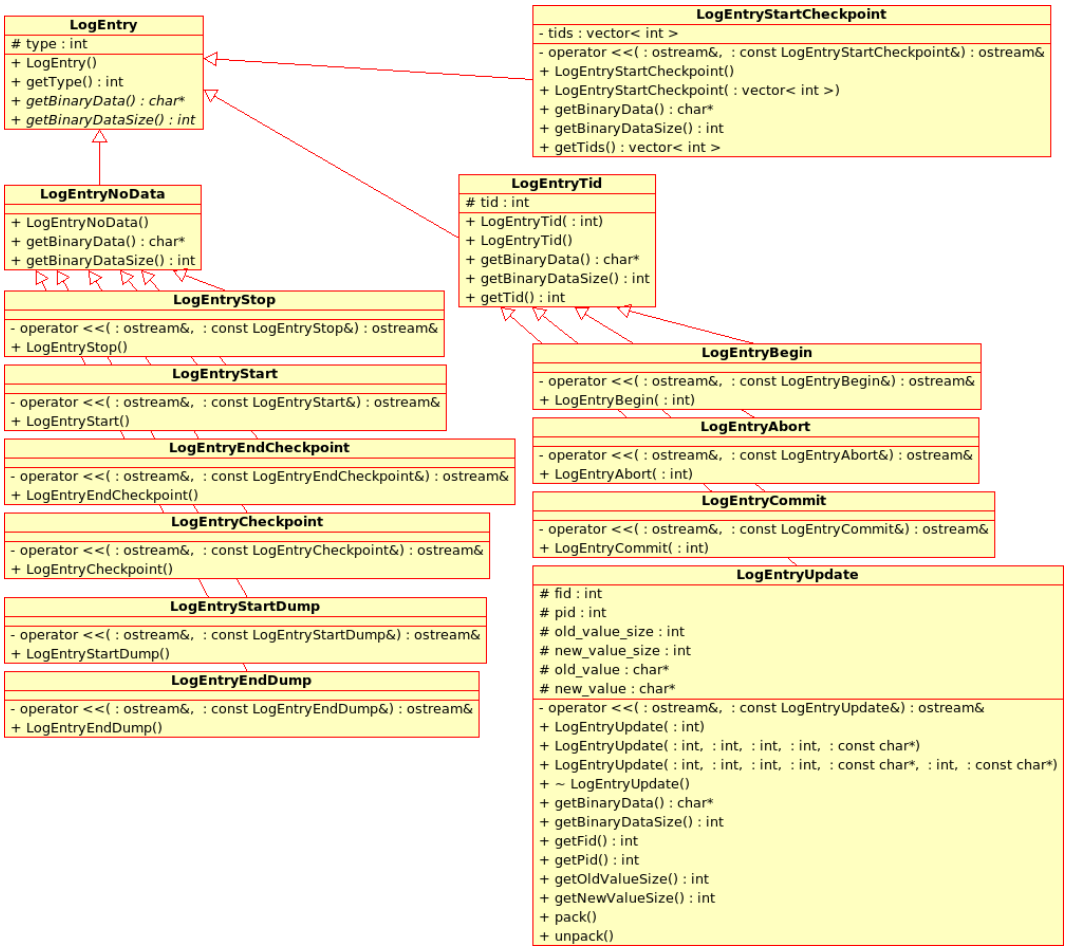
Klasą podstawową jest abstrakcyjna klasa *LogEntry* która definiuje interfejs dla klas pochodnych. Oprócz konstruktora, każda klasa pochodna musi być wyposażona w następujące metody:

getType() – metoda ta zwraca typ zgodnie z definicjami w pliku LogEntry.h

getBinaryDataSize() – metoda zwraca wielkość obiektu po serializacji do postaci binarnej

getBinaryData() – zserializowane dane z obiektu w postaci napisu gotowego do zapisu do pliku

Metody te są niezbędne dla obiektu klasy *LogWriter* która zajmuje się zapisywaniem do ich pliku. Poniżej kod źródłowy nagłówka klasy *LogEntry*.



Rysunek 4.3: Diagram klas dla elementów dziennika.

```
class LogEntry
{
    //operators
    friend ostream& operator<<(ostream&, const LogEntry&);

    private:
    protected:
        int type;
    public:
        //constructor/destructor
        LogEntry();

        //accessors
        int getType();

        virtual char* getBinaryData() = 0;
        virtual int getBinaryDataSize() = 0;
};
```

Metoda **getBinaryData()** dokonuje serializacji obiektu do postaci binarnej. Mapa takiej serializacji jest zależna od klasy i została przedstawiona w tabeli 4.1.5. Metoda serializacji jest dziedziczona przez klasy potomne.

W każdym przypadku pierwsze 4 bajty określają typ zapisanego obiektu. Po ich przeczytaniu obiekt klasy *LogReader* wie jakie znaczenie mają następne bajty.

4.1.5. Klasa LogEntryUpdate

Klasa LogEntryUpdate jest prawdopodobnie najciekawszą klasą w projekcie gdyż skupia w sobie cały mechanizm zapisywanie, wczytywania i odtwarzania danych. W przypadku obiektu tej klasy znaczenie kolejnych bajtów w pliku binarnym jest następujące:

tid – identyfikator transakcji (liczba typu *int*)

fid – identyfikator pliku (liczba typu *int*)

pid – identyfikator strony (liczba typu *int*); wraz z identyfikatorem pliku określają stronę w sposób unikalny w skali serwera

offset – przesunięcie

old_value_size – rozmiar danych dotyczących starej wartości (liczba typu *int*)

new_value_size – rozmiar danych dotyczących nowej wartości (liczba typu *int*)

old_value – dane dotyczące starej zawartości strony

new_value – dane dotyczące nowej zawartości

Klasa *LogEntryUpdate* jest wyposażona w metody *pack()* i *unpack()*, których celem jest zminimalizowanie ilości zapamiętywanych danych i optymalizacja zapamiętanych danych pod kątem problemu opisanego w przykładzie 4.1.1.

To czy plik jest blokowany na poziomie Manadżera Transakcji czy też nie obiekt klasy sprawdza za pomocą metody `isFrag()`. Jej kod sprowadza się do stwierdzenia na podstawie identyfikatora pliku czy dany plik jest czy nie jest blokowany na poziomie strony dysku.

Metoda pack()

Metoda ta powinna być wywołana za każdym razem przed zapisaniem danych do pliku dziennika. Metoda pakuje dane w zależności od typu pliku (blokowany czy nie). Jeżeli plik jest blokowany to należy mieć możliwość odtworzenia całej strony dysku przed i po zmianie. W związku z tym metoda pakuje dane pamiętając oryginalną wartość a także różnicę.

Obserwacje potwierdziły, że transakcje przeważnie zmieniają spójne bloki na stronie, w związku z tym różnica zapisywana jest w ten właśnie sposób, czyli metoda stara się znaleźć ten blok. Kolejne kroki metody:

1. porównuje starą i nową zawartość strony, bajt po bajcie od początku do końca, aż znajdzie różniący się elementów
2. porównuje starą i nową zawartość strony, bajt po bajcie od końca do początku, aż znajdzie różniący się elementów
3. przycina blok nowej wartości strony, zapamiętując tylko zmieniony fragment

Ta prosta technika pozwala za znaczne zmniejszenie ilości przechowywanych danych w dzienniku. Poniżej kod metody.

```
//samokontrola
assert(this->new_value_size==this->old_value_size);

//szukamy pierwszego znaku w ktorym stara i nowa wartosc sie roznia
for (int i = 0; i<this->new_value_size; i++) {
    if (this->old_value[i]!=this->new_value[i]) {
        this->offset = i;
        break;
    }
}

//szukamy ostatniego zmienionego znaku i pamietamy zawartosc
//pomiedzy pierwszym, a ostatnim zmienionym znakiem
for (int i = this->new_value_size - 1; i>=0; i--) {
    if (this->old_value[i]!=this->new_value[i]) {
        this->new_value_size = i - this->offset + 1;
        char* tmp = new char[this->new_value_size];
        memcpy(tmp, this->new_value+this->offset, this->new_value_size);
        delete this->new_value;
        this->new_value = tmp;
        break;
    }
}

//samokontrola
assert((this->offset+this->new_value_size) <= this->old_value_size);
```

Jeżeli jednak plik nie jest blokowany na poziomie Menedżera Transakcji, to należy pamiętać jeszcze mniejszy fragment strony – tylko zmienioną zawartość przed i po zmianie. Metoda pracuje wówczas w sposób opisany powyżej z tą różnicą, że przycina także fragment starej

wartości strony w przypadku w taki sam sposób (wielkość bloku i przesunięcie).

```
//nie musimy pamietac calej strony tylko fragment

//samokontrola
assert(this->new_value_size==this->old_value_size);

//szukamy pierwszego znaku w ktorym stara i nowa wartosc sie roznia
for (int i = 0; i<this->new_value_size; i++) {
    if (this->old_value[i]!=this->new_value[i]) {
        this->offset = i;
        break;
    }
}

//szukamy ostatniego zmienionego znaku i pamietamy zawartosc
//pomiedzy pierwszym, a ostatnim zmienionym znakiem
for (int i = this->new_value_size - 1; i>=0; i--) {
    if (this->old_value[i]!=this->new_value[i]) {
        //nowa wielkosc rekordu
        this->new_value_size = i - this->offset + 1;

        //przycinam nowa wartosc
        char* tmp = new char[this->new_value_size];
        memcpy(tmp, this->new_value+this->offset, this->new_value_size);
        delete this->new_value;
        this->new_value = tmp;

        //przycinam stara wartosc
        this->old_value_size=this->new_value_size;
        tmp = new char[this->old_value_size];
        memcpy(tmp, this->old_value+this->offset, this->old_value_size);
        delete this->old_value;
        this->old_value = tmp;
        break;
    }
}
```

Metoda unpack()

Celem metody `unpack()` jest przygotowanie zawartości rekordu aktualizacji do postaci wygodnej w odtwarzaniu. W przypadku plików nie blokowanych metoda właściwie nic nie robi, gdyż nic robić nie trzeba. W przeciwnym przypadku dokleja zmieniony fragment do starej wartości strony uzyskując w ten sposób pełną postać strony po zmianie przez transakcję.

```
void LogEntryUpdate::unpack() {
    if (!this->packed) return;

    if (this->isFrag()) {
        //nie trzeba odpakowywac, ale bedzie trzeba uwazac przy odtwarzaniu
        this->packed = false;
    } else {
        //odpakowujemy, czyli wpisujemy pelna zawatosc strony pod new_value
        char *tmp = new char[this->old_value_size];

        memcpy(tmp, this->old_value, this->old_value_size);

        for (int i=0; i<this->new_value_size; i++) {
            tmp[i+this->offset]=this->new_value[i];
        }

        delete[] this->new_value;

        this->new_value = tmp;
        this->offset=0;
        this->new_value_size=this->old_value_size;
        this->packed = false;
    }
}
```

W następnych wersjach można zastosować dalszą optymalizację ilości przechowywanych danych, także poprzez kompresję.

4.1.6. Klasa LogBackupManager

LogBackupManager to klasa odpowiedzialna za odzyskiwanie spójnego stanu bazy danych po awarii. Do jej zadań należy stwierdzenie stanu (spójny lub niespójny) na podstawie wpisów w pliku dziennika. W przypadku wykrycia niewłaściwego zakończenia pracy bazy danych, obiekt tej klasy odtwarza stan spójny wycofują i powtarzając odpowiednie transakcje. Działanie to zostało dokładnie opisane w punkcie 4.3.

4.2. Integracja

Komunikacja z modułami zewnętrznymi odbywa się za pomocą obiektu klasy *LogManager*. Klasa ta (podobnie jak klasa *LogWriter*) wspiera wielowątkowość i chroni swoje sekcje krytyczne za pomocą semaforów. Dzięki temu kilka wątków jednocześnie może korzystać z jednego obiektu *LogManager*. Sytuacja taka może mieć miejsce podczas tworzenia kopii archiwalnej bazy danych, gdy obiekt oprócz wykonywania swoich normalnych zadań (zlecanych przez wątek serwera), musi także wykonywać zlecenia procesu archiwizacji.

4.2.1. Rozpoczęcie i zakończenie transakcji

Interakcja z istniejącym SZBD została przedstawiona na rysunku 4.4. Obiekt serwera musi utworzyć obiekt *menedżera dzienników*, a także przekazać referencje do tego obiektu do

bajty	LogEntryNoData	LogEntryTid	LogEntryUpdate
4	(int)typ	(int)typ	(int)typ
8		(int)tid	(int)tid
12			(int)fid
16			(int)pid
20			(int)$offset$
24			(int)old_value_size
28			(int)new_value_size
28 + old_value_size			(char)old_value
28 + old_value_size+ new_value_size			(char)new_value

LogEntryStartCheckpoint	
4	(int)typ
8	(int)n
12	(int)tid_1
...	...
$(n+2)*4$	(int)tid_n

Tabela 4.4: Serializacja dla klas LogEntry.

menedżera transakcji i *menedżera dysku*. Dzięki temu *menedżer transakcji* może informować *menedżera dziennika* o rozpoczęciu transakcji (*pushBegin*) i jej zakończeniu (*pushAbort* lub *pushCommit*). Na rysunku przedstawiono, że *menedżer dysku* informuje menedżera dzienników o zmianach stron na dysku wywołanych przez aktualne transakcje (rekord aktualizacji – *pushUpdate*). Jest to jednak uproszczenie gdyż nie dzieje się tak bezpośrednio. Dokładny mechanizm został opisany w następnym rozdziale (4.2.2).

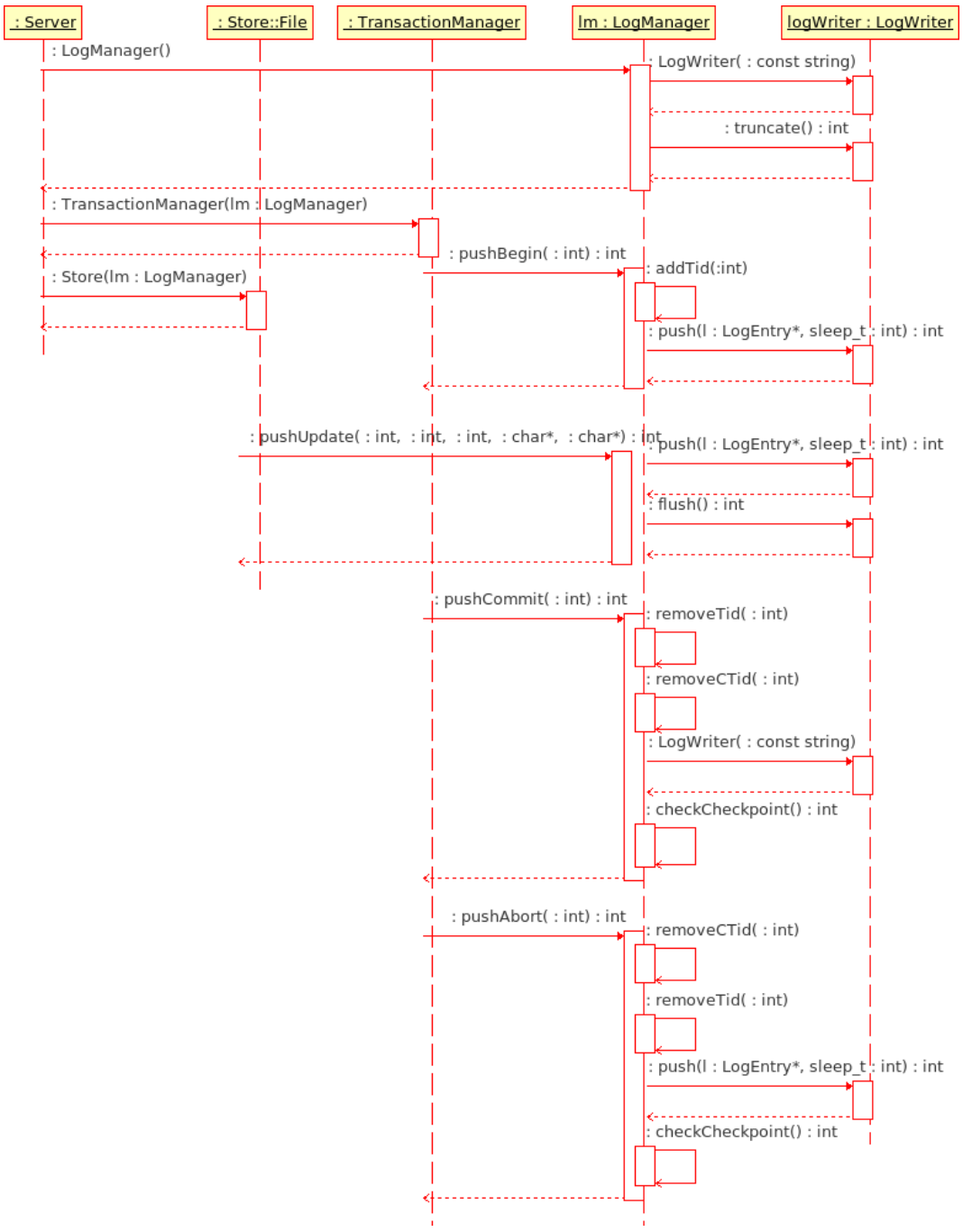
4.2.2. Rekord aktualizacji

Na poziomie fizycznym, baza danych systemu LoXiM składa się z kilku plików. Dostęp do tych plików jest zorganizowany poprzez kilka klas, tworzących różne warstwy. Rysunek 4.5 przedstawia interesujący nas fragment tej struktury. Strzałki określają wywołania zapisu, ponieważ te są interesujące z perspektywy zapisu do dziennika. Strzałki są opisane odpowiednimi nazwami metod wywoływanymi przez obiekty.

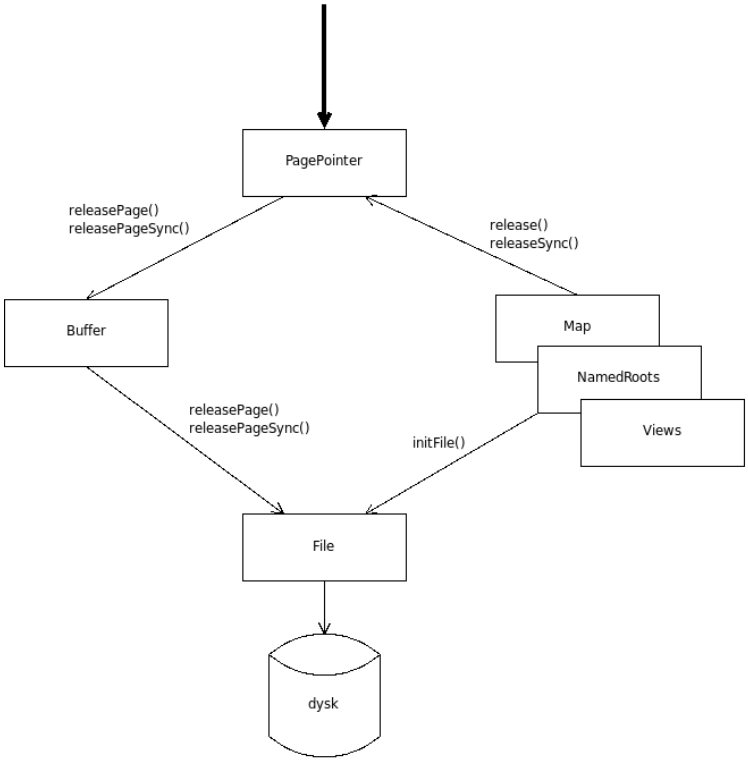
Klasa *File* zajmuje się bezpośrednim dostępem do dysku. Bezpośredni dostęp do tej klasy ma tylko klasa bufora i klasy *Map*, *NamedRoots* i *Views*. Przy czym te trzy ostatnie korzystają z niej bezpośrednio, tylko podczas tworzenia nowych plików. Klasa *Buffer* jest natomiast wywoływana tylko z klasy *PagePointer*.

Cała komunikacja pomiędzy systemem, a plikiem na dysku jest zorganizowana przez klasę *PagePointer*. Klasa udostępnia metody umożliwiające przypinanie i odpinanie stron z dysku. Podczas ich odpinania (metody *release()* i *releaseSync()*) sprawdzane jest czy strona została zmieniona i jeżeli tak, to w pliku dziennika tworzony jest odpowiedni rekord aktualizacji.

Żeby wszystko działało poprawnie, przy każdym wywołaniu metody *release()* lub *releaseSync()* w klasie *PagePointer* musi być określony numer transakcji wywołującej to zdarzenie.



Rysunek 4.4: Diagram sekwencji dla obiektów tworzących dziennik.



Rysunek 4.5: Struktura klas dostępu do plików na dysku w systemie LoXiM

Poza tym obiekt tej klasy, musi znać obiekt *menedżera dzienników*.

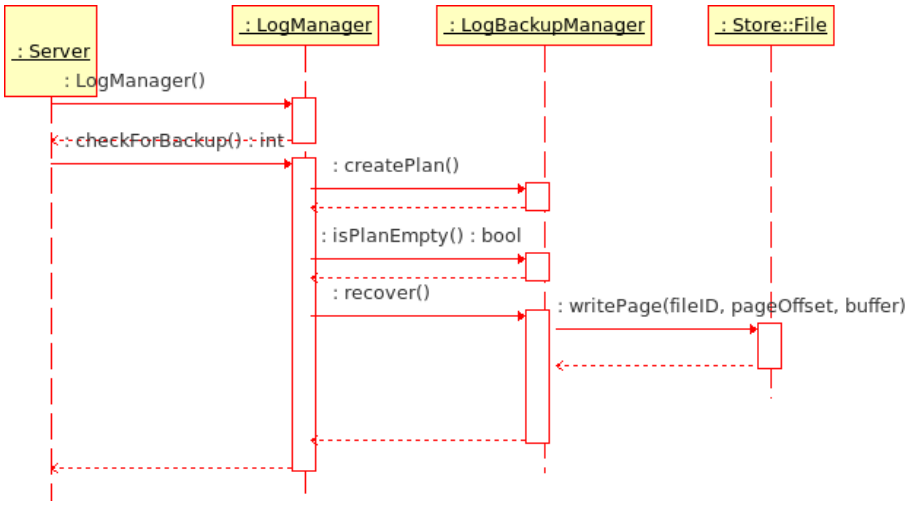
4.2.3. Tworzenie punktów kontrolnych

Podczas swojej pracy *menedżer dzienników* przechowuje listę aktualnie przetwarzanych transakcji (zmienna *tids*). Transakcje są dodawane do tej listy podczas wywołania metody *pushBegin* i usuwane podczas wywołania metod *pushAbort* i *pushCommit*.

Jeżeli *menedżer dzienników* jest w trakcie tworzenia punktu kontrolnego to metody *pushAbort* i *pushCommit* dodatkowo usuwają transakcję z listy *ckpt_tids*. Lista ta przechowuje informacje o transakcjach które są otwarte od czasu rozpoczęcia tworzenia punktu kontrolnego. Jest ona tworzona w momencie rozpoczęcia tworzenia punktu kontrolnego poprzez skopiowanie zawartości listy *tids*.

Gdy *menedżer dzienników* otrzyma informacje o zakończeniu transakcji, wywoływana jest metoda *checkCheckpoint()*. Do jej zadań należy sprawdzenie czy można zakończyć przetwarzanie punktu kontrolnego (pusta lista aktualnie przetwarzanych transakcji) i wstawia do dziennika rekord $< END\ CKPT >$. W przeciwnym przypadku podejmowana jest decyzja o utworzeniu punktu kontrolnego. Szansa utworzenia punktu kontrolnego jest wprost proporcjonalne do czasu logicznego[3] który upłynął od utworzenia poprzedniego punktu kontrolnego i odwrotnie proporcjonalna do liczby aktualnie otwartych transakcji. Strategię tę opisuje wzór $\frac{\Delta t}{|k_T|} > W$, gdzie W to parametr ustalany przez administratora bazy danych i dostosowany do specyfiki lokalnej bazy danych. W ogólności, czym większa wartość parametru W tym rzadziej tworzone są punkty kontrolne. Jeżeli warunek jest spełniony, to wywoływana jest metoda *pushStartCheckpoint* która kopiuje listę aktualnie przetwarzanych transakcji i zapisuje rekord

[3]ilość rekordów wpisanych do pliku dziennika



Rysunek 4.6: Diagram sekwencji dla procesu odtwarzania bazy danych po awarii.

$< START\ CKPT(...) >$ do dziennika.

Szczególnym przypadkiem jest sytuacja gdy lista aktualnie przetwarzanych transakcji jest pusta. Wówczas do dziennika wstawiany jest rekord typu $< CKPT >$ i dokonuje się przycięcia dziennika (lub nie – zależnie od ustawień).

4.3. Odtwarzanie

Proces odtwarzania po awarii, a także odtwarzania z kopii archiwalnej wygląda identycznie. Procedura odtwarzania jest uruchamiana automatycznie przez obiekt serwera zaraz po utworzeniu obiektu menedżera dzienników. W przypadku odtwarzania z kopii archiwalnej należy uprzednio przegrać plik bazy danych i plik dziennika usuwając poprzednią wersję.

Proces odtwarzania przebiega w sposób przedstawiony na rysunku 4.6. Po uruchomieniu, obiekt serwera inicializuje po kolei różne obiekty. Po inicializacji obiektu *menedżera dysku* i *menedżera dzienników* odpytuje tego ostatniego o stan dzienników (*LogManager::checkForBackup()*). *Menedżera dzienników* tworzy obiekt klasy *LogBackupManager* i prosi go o ułożenie planu odtwarzania (*metoda LogBackupManager::createPlan()*), który stara się ułożyć taki plan na podstawie aktualnych dzienników. Jeżeli po analizie okaże się, że serwer został zamknięty poprawnie to ułożony plan nie zawiera żadnych transakcji do wycofania ani do powtórzenia. Plan może być pusty jeżeli:

- serwer został zamknięty poprawnie – dziennik zakończony jest rekordem typu $< STOP >$
- serwer nie został zamknięty poprawnie, ale w chwili upadku wszystkie dane były zsynchronizowane – na przykład dziennik zakończony rekordem $< CKPT >$

W tym wypadku serwer może kontynuować inicjację obiektów, w celu włączenia się do normalnej pracy.

Jeżeli plan nie jest pusty, to zawiera listę transakcji z podziałem na te do odtworzenia i wycofania. *Menedżer dzienników* prosi obiekt *LogBackupManager* o wykonanie odtwarzania (metoda *LogBackupManager::recover()*). Odtwarzanie przebiega w dwóch krokach: w pierwszym odpowiednie transakcje są wycofywane, w drugim odpowiednie transakcje są powtarzane – zgodnie z wcześniej utworzonym planem. Pliki bazy danych są modyfikowane za pomocą metody *File::writePage(...)*.

W procesie odtwarzania mogą pojawić się następujące przypadki szczególne:

1. należy zapisać wartość zmienioną przez transakcję na stronie pliku, której inicjacja nie została zapisana na dysku; w takim wypadku program tworzy odpowiednią (jeden lub więcej) stron wypełnionych „zerami" i zmienia jej wartość

2. należy wycofać inicjację strony; wówczas program wymazuje zawartość i zostawia stronę pełną wartości „zerowych"

Po skończonym odtwarzaniu serwer może wznowić normalną pracę.

4.4. Przykład awarii

4.4.1. Wprowadzenie

W rozdziale tym przedstawiono krok po kroku przykładową awarię, a także proces odzyskiwania. Przykład został zbudowany na bazie podstawowego scenariusza testowego umożliwiającego przetestowanie podstawowej i pełnej funkcjonalności modułu Logs. Wszystkie dane pochodzą z faktycznie przeprowadzonego testu.

4.4.2. Przykład 1 – CREATE

W celu przeprowadzenia testu utworzono pustą bazę danych. W jednym oknie uruchomiono program serwera (Listener), w drugim program klienta, a w trzecim podglądano zawartość pliku dziennika poleceniem `./logmaster --cat`.

Krok 1 – Logowanie

Logowanie przeprowadzono na domyślne dane administratora.

Stan konsoli klienta:

```
[bart@brachypelma SBQLCli]$ ./SBQLCli
SBQLCli ver 1.1
Ctrl-d to exit
Login: root
Password:
<Connection::deserialize> tworze obiekt VOID
<Connection::deserialize> tworze obiekt BOOL (true)
<Connection::deserialize> tworze obiekt VOID

 >
```

Stan pliku dziennika:

```
[bart@brachypelma Server]$ ./logmaster --cat
<START>
<BEGIN 0>
<COMMIT 0>
<CKPT>
```

Na początku dodano rekord $< START >$ określający włączenie serwera. Rekord ten jest przydatny w przypadku gdy nie jest włączona opcja auto-przycinania dziennika, gdyż przy analizie będzie wiadomo, że wcześniej (przez tym rekordem) nie ma żadnych przydatnych danych.

Następnie wykonano jedną transakcję w celu uwierzytelnienia użytkownika 'root'. Transakcja była zapytaniem które nie zmieniło stanu bazy. W związku z tym pomiędzy rekordem $< BEGIN >$ a $< COMMIT >$ nie ma żadnych rekordów aktualizacji.

Po skończonej transakcji *menedżer dziennika* sprawdza listę aktualnie przetwarzanych transakcji. Jest ona pusta więc można zakończyć dziennik rekordem $< CKPT >$. Gdyby włączona była opcja auto-przycinania, to w tym miejscu dziennik został by przycięty.

Krok 2 – Rozpoczęcie transakcji

```
 > begin;
 \ /
<SBQLCli> query: begin;
<SBQLCli> adding last result as a parameter $ans = void
<Connection::deserialize> tworze obiekt VOID
void
 >
```

Stan pliku dziennika:

```
[bart@brachypelma Server]$ ./logmaster --cat
<START>
<BEGIN 0>
<COMMIT 0>
<CKPT>
<BEGIN 1>
```

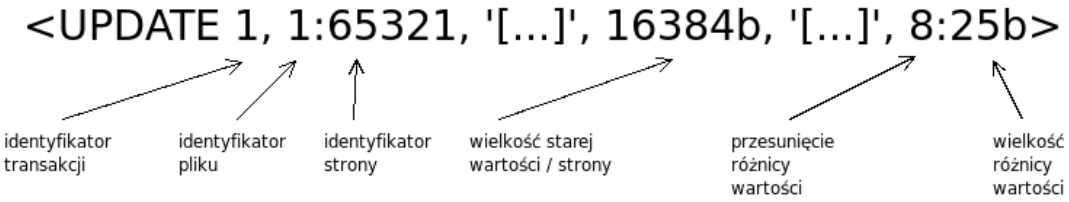
Menedżer transakcji zgłasza rozpoczęcie transakcji co jest odnotowywanie w dzienniku.

Krok 3 – Utworzenie obiektu

```
 > create 1 as obiekt;
 \ /
<SBQLCli> query: create 1 as obiekt;
<SBQLCli> adding last result as a parameter $ans = void
<Connection::deserialize> tworze obiekt BAG
<Connection::deserialize> tworze obiekt REFERENCE
{ ref(1) }
 >
```

Stan pliku dziennika:

```
[bart@brachypelma Server]$ ./logmaster --cat
<START>
<BEGIN 0>
<COMMIT 0>
<CKPT>
<BEGIN 1>
```



Rysunek 4.7: Opis kolumn wydruku rekordu aktualizacji.

```
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:65296, '[...]', 16384b, '[...]', 8:9b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 12:10b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:37b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:25b>
```

Polecenie powoduje zapisanie obiektu w bazie danych. Powoduje to zapisanie w dzienniku serii rekordów aktualizacji, zawierających dane stron przed i po zmianie. Kolejne kolumny wydruku rekordu aktualizacji zostały opisane na rysunku 4.7.

Krok 4 – Zakończenie transakcji

```
 > end;
 \ /
<SBQLCli> query: end;
<SBQLCli> adding last result as a parameter $ans = { ref(1) }
<Connection::deserialize> tworze obiekt VOID
void
 >
Ctrl-C
[bart@brachypelma SBQLCli]$
```

Stan pliku dziennika:

```
[bart@brachypelma Server]$ ./logmaster --cat
<START>
<BEGIN 0>
<COMMIT 0>
<CKPT>
<BEGIN 1>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:65296, '[...]', 16384b, '[...]', 8:9b>
```

```
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 12:10b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:37b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:25b>
<COMMIT 1>
<CKPT>
<END>
```

Menedżer transakcji kończy transakcję. Jak zwykle po zakończeniu, *menedżer dziennika* sprawdza listę aktualnie przetwarzanych transakcji. Jest ona pusta więc można dodać do dziennika rekord $< CKPT >$ lub go przyciąć.

Następnie rozłączono się z serwerem, a także wydano mu polecenie zakończenia pracy. Do dziennika został wprowadzony rekord $< END >$ określający prawidłowe zakończenie pracy. Przy następnym uruchomieniu obiekt klasy *LogBackupManager* trafi na ten rekord w pierwszej kolejności, przeglądając dziennik od końca. Na tej podstawie oceni, że proces naprawy bazy danych nie musi być uruchamiany.

Analiza poprawności na poziomie binarnym

Analiza poprawności tworzenia dziennika przebiegała następująco:

1. serwer został ustawiony na zapis natychmiastowy (synchroniczny), bez korzystania z mechanizmu zapisu opóznionego – każde wywołanie metody `releasePage()` bylo kierowane do metody `releasePageSync()`

2. każdy zapis w klasie `File` serwera był przechwytywany – praca systemu byla zatrzymywana tuż przed i zaraz po zapisie

Tabela 4.5 przedstawia kolejne kroki wykonywane w ramach dwóch konsoli: serwera i testowej. Zgodnie z wymaganiem z rozdziału 2.3.2 tuż przed zapisem transakcji na dysku, rekord dziennika znajduje się na dysku. Po wywołaniu polecenia `compare` widać, że suma kontrolna[4] odpowiedniej strony w bazie danych jest zgodna z „starą" wartością w rekordzie aktualizaji w dzienniku. Wywołanie po zapisie, pokazuje zgodność nowej wartości strony (po zapisie na dysku) z „nową" wartością w rekordzie aktualizaji w dzienniku.

Zamiast kolejnych wywołań polecenia `compare` można skorzystać z gotowego skryptu `cyclic_compare`. Więcej o narzędziach w rodziale A.4.

[4]Suma kontrolna MD5

Tabela 4.5: Analiza poprawności tworzenia dzienników na poziomie binarnym.

Konsola serwera	**Konsola testowa**

`Server: [Listener.MAIN] Running port ----> 6543`
`[...]`
`Log: [LogManager.pushUpdate] --> start.`
`Log: [LogManager.pushUpdate] --> (tid=0, fid=65296, pid=1)`
`Log: [LogManager.pushUpdate] --> flush.`
`Log: [LogManager.pushUpdate] --> end.`
`Store: File: ------------> fileid:offset:len 65296:16384:16384`
`Store: File: Before write (65296:16384:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 1`
	`baza c8faf446cec3c612d65a16f1ca24ac3e`
	`old  c8faf446cec3c612d65a16f1ca24ac3e`
	`new  b21decb5e57857e7d5efd0d9069c8897`

`Store: File: After write (65296:16384:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 1`
	`baza b21decb5e57857e7d5efd0d9069c8897`
	`old  c8faf446cec3c612d65a16f1ca24ac3e`
	`new  b21decb5e57857e7d5efd0d9069c8897`

`PagePointer::releaseSyncing page (65296, 0)`
`Log: [LogManager.pushUpdate] --> start.`
`Log: [LogManager.pushUpdate] --> (tid=0, fid=65296, pid=0)`
`Log: [LogManager.pushUpdate] --> flush.`
`Log: [LogManager.pushUpdate] --> end.`
`Store: File: ------------> fileid:offset:len 65296:0:16384`
`Store: File: Before write (65296:0:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 2`
	`baza 718bc2f722a523e57123aa5086f7077b`
	`old  718bc2f722a523e57123aa5086f7077b`
	`new  dee0e515a30f76baae033b4d0d5dc0e2`

`Store: File: After write (65296:0:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 2`
	`baza dee0e515a30f76baae033b4d0d5dc0e2`
	`old  718bc2f722a523e57123aa5086f7077b`
	`new  dee0e515a30f76baae033b4d0d5dc0e2`

dalej podobnie...	

4.4.3. Przykład 2 – powtórzenie

Do przeprowadzenia przykładu załóżmy, że serwer z poprzedniego przykładu uległ awarii po dodaniu do pliku dziennika rekordu $< COMMIT >$. W celu spreparowania takiego dziennika można użyć dziennika z przykładu w rozdziale 4.4.2 przyciętego za pomocą narzędzia `dd`[5]. W efekcie otrzymamy dziennik postaci:

```
[bart@brachypelma Server]$ ./logmaster --cat
<START>
<BEGIN 0>
<COMMIT 0>
<CKPT>
<BEGIN 1>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:65296, '[...]', 16384b, '[...]', 8:9b>
```

[5]Dziennik należy wyświetlić za pomocą narzędzia `./logmaster` użytego z parametrem `-show-offset`

```
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 12:10b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:37b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:25b>
<COMMIT 1>
```

Podczas analizy dziennika widać, że serwer nie został zamknięty poprawnie, gdyż dziennik nie jest zakończony rekordem $< END >$. Analizując dalej znajdujemy rekord $< COMMIT >$ co oznacza, że transakcja 1 powiodła się i należałoby ją powtórzyć. Kończymy analizę po osiągnięciu rekordu $< CKPT >$, nie znajdując transakcji do powtórzenia ani wycofania. Poniżej fragment dziennika serwera.

```
LogBackupManager: [LogBackupManager.LogBackupManager] -> start
LogBackupManager: [LogBackupManager.createPlan] -> start
LogBackupManager: [LogBackupManager.createPlan] -> end
LogBackupManager: [LogBackupManager.recover] -> start
LogBackupManager: [LogBackupManager.recover] -> undoing...
LogBackupManager: [LogBackupManager.recover] -> redoing...
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65312)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65312)
LogBackupManager: [LogBackupManager.recover] -> end
```

Analiza przykładu na poziomie binarnym

Analizę procesu odtwarzania (powtórzenia transakcji) bazy danych została przeprowadzona analogicznie jak w rozdziale 4.4.4, z tą różnicą że zamiast starych wartości są wpisywane nowe.

4.4.4. Przykład 3 – wycofanie

Do przeprowadzenia przykładu załóżmy, że serwer z poprzedniego przykładu uległ awarii przed dodaniem do pliku dziennika rekordu $< COMMIT\ 1 >$ (lub w innym momencie przed,

ale po wstawieniu pierwszego rekordu aktualizacji). W celu spreparowania takiego dziennika można użyć dziennika z przykładu w rozdziale 4.4.2 przyciętego za pomocą narzędzia `dd`. W efekcie otrzymamy dziennik postaci:

```
[bart@brachypelma Server]$ ./logmaster --cat
<START>
<BEGIN 0>
<COMMIT 0>
<CKPT>
<BEGIN 1>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:65296, '[...]', 16384b, '[...]', 8:9b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 12:10b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 0:1, '[...]', 16384b, '[...]', 20:1b>
<UPDATE 1, 1:1, '[...]', 16384b, '[...]', 8:16373b>
<UPDATE 1, 1:65296, '[...]', 16384b, '[...]', 8:20b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:37b>
<UPDATE 1, 1:65312, '[...]', 16384b, '[...]', 8:25b>
<COMMIT 1>
<CKPT>
```

Po uruchomieniu serwera i przeanalizowaniu przez niego dziennika zostanie uruchomiona procedura odzyskiwania, jako że serwer nie został zamknięty poprawnie. Analizę tradycyjnie kończymy na rekordzie $< CKPT >$. Jak widać od tego czasu została rozpoczęta jedna transakcja. Nie wiadomo jednak czy się zakończyła, w związku z tym należy ją wycofać. Poniżej fragment dziennika procesu serwera.

```
Log: [LogManager.checkForBackup] --> start.
LogBackupManager: [LogBackupManager.LogBackupManager] -> start
LogBackupManager: [LogBackupManager.createPlan] -> start
LogBackupManager: [LogBackupManager.createPlan] -> end
LogBackupManager: [LogBackupManager.recover] -> start
LogBackupManager: [LogBackupManager.recover] -> undoing...
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (0, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 1)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65296)
```

```
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65312)
LogBackupManager: [LogBackupManager.recover] -> (fid, pid) = (1, 65312)
LogBackupManager: [LogBackupManager.recover] -> redoing...
LogBackupManager: [LogBackupManager.recover] -> end
```

Analiza przykładu na poziomie binarnym

Tabela 4.6 przedstawia fragment procesu odtwarzania. Jak widać w wyniku wycowania stare wartości bazy danych są na nowo wpisywane do bazy danych. Zawartość bazy danych w kroku 1 i 3 jest zgodna z nową wartością. Zgodność ta jest przypadkowa[6].

Tabela 4.6: Analiza poprawności odtwarzania dzienników na poziomie binarnym.

Konsola serwera	**Konsola testowa**

`Server: [Listener.MAIN] Running port ----> 6543`
`[...]`
`Log: [LogManager.checkForBackup] --> start.`
`LogBackupManager: [LogBackupManager.LogBackupManager] -> start`
`LogBackupManager: [LogBackupManager.createPlan] -> start`
`LogBackupManager: [LogBackupManager.createPlan] -> end`
`LogBackupManager: [LogBackupManager.recover] -> start`
`LogBackupManager: [LogBackupManager.recover] -> undoing...`
`LogBackupManager: [LogBackupManager.recover] -> \`
`(fid, pid, offset, size) = (65296, 1, 0, 16384)`
`Logs: File: ------------> fileid:offset:len 65296:16384:16384`
`Logs: File: Before write (65296:16384:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 1`
	`baza b21decb5e57857e7d5efd0d9069c8897`
	`old  c8faf446cec3c612d65a16f1ca24ac3e`
	`new  b21decb5e57857e7d5efd0d9069c8897`

`Store: File: After write (65296:16384:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 1`
	`baza c8faf446cec3c612d65a16f1ca24ac3e`
	`old  c8faf446cec3c612d65a16f1ca24ac3e`
	`new  b21decb5e57857e7d5efd0d9069c8897`

`LogBackupManager: [LogBackupManager.recover] -> \`
`(fid, pid, offset, size) = (65296, 0, 0, 16384)`
`Logs: File: ------------> fileid:offset:len 65296:0:16384`
`Logs: File: Before write (65296:0:16384)`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 2`
	`baza dee0e515a30f76baae033b4d0d5dc0e2`
	`old  718bc2f722a523e57123aa5086f7077b`
	`new  dee0e515a30f76baae033b4d0d5dc0e2`

`Logs: File: After write (65296:0:16384`
 	
Oczekiwanie na klawisz Enter...
	`[bart@brachypelma tools]$ ./compare 2`
	`baza 718bc2f722a523e57123aa5086f7077b`
	`old  718bc2f722a523e57123aa5086f7077b`
	`new  dee0e515a30f76baae033b4d0d5dc0e2`

dalej podobnie...	

[6]Może nie do końca przypadkowa, po prostu serwer został wyłączony w miarę gładko, a nie na przykład w połowie zapisu

Rozdział 5

Podsumowanie

Systemy zarządzania bazami danych mają w obecnych czasach coraz bardziej odpowiedzialne zadania i pojawiają się w każdej niemal dziedzinie życia. Awaryjność systemów bazodanowych spowodowała, że większość użytkowników traktuje kartkę papieru jako najpewniejszą metodę przechowywania danych. Stosowanie spójnej i przemyślanej polityki ochrony bazy danych dobrze zintegrowanej z jej wewnętrznymi mechanizmami może znacznie poprawić bezpieczeństwo danych.

Podstawową metodą stosowaną obecnie w większości poważnych i liczących się systemów bazodanowych jest metoda dzienników binarnych, polegająca na rejestrowaniu zmienianych danych. Z trzech podstawowych stylów prowadzenia dzienników: „rejestrowanie wycofań", „rejestrowanie powtórzeń" oraz „rejestrowanie wycofań i powtórzeń" ta ostatnia wydaje się najskuteczniejsza i najbardziej elastyczna. Metoda ta jest najbardziej skuteczna – pozwala zarówno na wycofywanie i powtarzanie transakcji, a także najłatwiejsza w implementacji – nakłada minimalną ilość reguł o których należy pamiętać przy implementacji.

Wadą tej metody jest większa ilość miejsca potrzebnego do przechowywania dzienników w stosunku do pozostałych metod. W krótkim czasie może to doprowadzić do sytuacji w której pliki dziennika są wielokrotnie większe niż pliki faktycznej bazy danych. Dodatkowo analizowanie tak wielkich dzienników może być bardzo czasochłonne. Przeciwdziałać temu można dzięki użyciu metody punktów kontrolnych i auto-przycinania.

Punkty kontrolne ograniczają rozmiary dzienników do przeanalizowania. Auto-przycinanie występuje w miejscu spoczynkowych punktów kontrolnych, które są specyficznym przypadkiem niespoczynkowych punktów kontrolnych lub w chwili poprawnego zamknięcia pliku bazy danych.

Implementacja zaprezentowana w rozdziale 4 umożliwia zastosowanie dzienników „rejestrowanie wycofań i powtórzeń" z niespoczynkowymi punktami kontrolnymi w istniejącej obiektowej bazie danych LoXiM. Zaimplementowany moduł jest zaprojektowany obiektowo w sposób umożliwiający wymianę poszczególnych obiektów na inne implementacje, które będą korzystać z innych metod lub strategii.

W trakcie implementacji i testowania mechanizmu dzienników według projektu opisanego w tej pracy, pojawiły się pewne zagadnienia które mogą stanowić podstawę do rozpoczęcia dalszych prac w tej dziedzinie. Poniżej sugestie dotyczące dalszego rozwoju modułu Logs.

Dalsze prace mogłyby polegać na zoptymalizowaniu ilości zapamiętywanych danych (np. poprzez pamiętanie tylko różnic) i kompresje "w locie" za pomocą publicznie dostępnych algorytmów. Przykładowo kompresja opisanego dziennika (z rozmiaru 262762b) algorytmem `gzip` spowodowała zmniejszenie pliku do około 805b, a algorytmem `bzip2` do 364b. Badanie zostało przeprowadzone na pustej bazie danych, tzn. oprócz wstawionego obiektu pozostałą

zawartość plików stanowiły „zera". W związku z tym w przypadku bazy „produkcyjnej" sytuacja może wyglądać mniej optymistycznie, albo wręcz może się okazać że ta droga kompresji jest zupełnie nieopłacalna.

Przy wyborze strategii „checkpointingu" posłużyłem się „zasadą Occama" [WO27] i wybrałem najprostszą strategię z parametrem. W szczególności, wystarczająca duża wartość parametru może ograniczyć stawianie punktów kontrolnych tylko do sytuacji w której lista aktualnie przetwarzanych transakcji jest pusta. Dalsze prace mogłyby polegać na zbadaniu tej strategii w rzeczywistej bazie danych, a także zoptymalizowaniu jej lub stworzeniu lepszej.

Test opisany w punkcie 4.4 pokazał, że przy zapisie pojedynczego obiektu do bazy danych *menedżer dysku* dokonuje zmian na 13 stronach. Wydaje się, że liczba zmienionych stron nie musi być aż tak duża. Być może należałoby dokonać badań w kierunku optymalizacji ilości zapisów. Zmniejszenie ilości zapisów wpłynęłoby pozytywnie na wydajność zarówno bazy danych jak i samego modułu Logs.

Dopełniająca metodą ochrony danych w bazie danych jest system tworzenia kopii archiwalnej. Tworzenie kopii archiwalnej niespoczynkowo ma następujące zalety: nie wymaga zatrzymania pracy systemu bazodanowego i tworzy jej spójną kopię. Dalsze prace mogłyby też polegać na przygotowaniu wydajnego systemu tworzenia i odzyskiwania zapasowych kopii metodą przyrostową. Tworzenie kopii przyrostowych jest mniejszym obciążeniem dla systemu komputerowego przechowującego bazę danych, a także wymaga mniejszej ilości miejsca na dysku do ich przechowywania.

Dodatek A

Opis załączonego oprogramowania

Na płycie CD załączonej do pracy magisterskiej znajduje się implementacja obiektowej bazy danych LoXiM. Poniżej opisano zawartość katalogów. Najnowsza wersja systemu LoXiM jest dostępna pod adresem `http://loxim.mimuw.edu.pl/`.

/praca – katalog zawiera pracę magisterską w postaci pliku TEX, a także wszystkie rysunki zawarte w pracy w postaci plików .dia i .xmi z programów Dia i Umbrello. Dodatkowo zawiera plik PDF.

/praca/materialy – katalog zawiera pliki z materiałami pobranymi z Internetu

/praca/prezentacja – katalog zawiera pliki z materiałami dotyczącymim prezentacji tematu pracy magisterskiej z końca roku 2006; niektóre materiały i koncepcje mogą być nieaktualne

/praca/tools – dodatkowe narzędzia testowe; narzędzia zostały opisane w punkcie A.4

/LoXiM – katalog zawiera implementację systemy LoXiM zintegrowaną z dziennikami zaprojektowanymi i zaimplementowanymi w pracy magisterskiej

/LoXiM/Logs – moduł Logs

/LoXiM/Logs/logmaster – źródła programu logmaster

Implementacja przeznaczona jest dla systemu Linux. W celu kompilacji należy przegrać pliki na dysk, a następnie w katalogu ∼/LoXiM uruchomić polecenie *make*. Jeżeli proces kompilacji przebiegnie poprawnie powinny pojawić się następujące pliki:

- ∼/Server/Listener – plik uruchamialny serwera
- ∼/SBQLCli/SBQLCli – plik uruchamialny klienta
- ∼/Logs/logmaster/logmaster – narzędzie opisane w rozdziale A.3

W celu testów należy uruchomić na jednej konsoli program serwera, a na innej program klienta.

A.1. Konfiguracja

Moduł logs posiada możliwości ustawienia parametrów konfiguracyjnych. Część z nich można ustawić na etapie kompilacji, a część w pliku konfiguracyjnym serwera przez jego uruchomieniem.

A.1.1. Opcje kompilacji

Opcje kompilacji należy ustawić w plikach źródłowych przed rozpoczęciem procesu kompilacji:

Logs/LogManager.h

CHECK_DIFFERENCE – jeżeli stała jest zdefiniowaną to *menedżer dzienników* porównuje wartość stron przed i po; sprawdzanie to nie jest obowiązkowe gdyż zajmuje się tym klasa *PagePointer*;

TRUNCATE_LOGS – jeżeli stała jest zdefiniowana to *menedżer dzienników* dokonuje auto-przycinania pliku dziennika; generalnie opcja powinna być włączona, ale na czas testów mechanizmu logowania trzeba ją wyłączyć;

A.1.2. Plik Server.conf

Plik *Server.conf* jest głównym plikiem konfiguracyjnym serwera. W pliku tym znajduje sie specjalna sekcja dotycząca ustawień modułu Logs. W ramach tej sekcji można ustalić następujące parametry:

logspath – określa położenie plików dziennika; domyślnie jest to wartość /tmp/szbd_logs

checkpointing_w – określa parametr strategii stawiania punktu kontrolnego opisanej w punkcie 4.2.3; domyślna wartość to 5;

A.2. Dokumentacja

Dokumentacja modułu Logs została przygotowana w kodzie zgodnie wytycznymi programu Doxygen (http://www.doxygen.org/). Uruchomienie polecenia *make doc* w głównym katalogu projektu powoduje utworzenie plików HTML z dokumentacja. W szczególności rozdział na temat modułu Logs można znaleźć w pliku ∼/Documentation/API/html/namespaceLogs.html.

A.3. logmaster

Na potrzeby testowania działania modułu Logs powstał program o nazwie *logmaster*. Jego pierwotną funkcją było czytanie zawartości plików dziennika, które są przechowywane na dysku w postaci binarnej. Program korzysta bezpośrednio z klas modułu Logs. Poniżej wydruk ekranu pomocy wyświetlanego przez program po wywołaniu bez parametrów. Użyto języka angielskiego ze względu na umiędzynarodowienie projektu LoXiM.

```
logmaster v. 1.0.1
Usage: logmaster <action> [OPTIONS]
Actions:
-?, --help         Display this help and exit.
--cat              Display log file
--tac              Display log file reverse order
--show-plan        Display recovery plan (if any)
--create           Create random log file
--create-test      Create random test log file
--create-hc-test   Create hardcoded test log file
```

```
--dd-old            Display dd command for old value in last update record
--dd-new            Display dd command for new value in last update record

Options:
-f=<filename>       Log file name (default: /tmp/sblogs)
-v=<level>          Verbosity level (0,1,2, default: 0)
-c=<number>         Number of records created (default: 1000)
-t=<number>         Number of concurent "threads" to create test log file (default: 5)
-cp=<0|1>           Use checkpoint while building plan (default: 0)
-i=<numer>          Show i-th update record with --dd-<old|new> command (default: -1)
-autocheck          Autocheck created log file (default: 0)
-show-offset        Display log record offset (default: 0)
-show-dd            Display dd command (default: 0)
```

Program przymuje dwa typy parametrów: akcje i opcje. Akcje określają zadanie do wykonania, natomiast opcje parametryzują wykonanie poszczególnych akcji. Program może wykonać następujące akcje:

`--cat` – wypisywanie pliku od początku do końca; dostępne opcje: -f, -v, -autocheck, -show-offset

`--tac` – wypisywanie pliku od końca do początku; dostępne opcje: -f, -v, -autocheck, -show-offset

`--show-plan` – wypisywanie planu odtwarzania, czyli lista transakcji do powtórzenia lub wycofania; dostępne opcje: -f, -v, -autocheck, -cp

`--create` – utworzenie pliku dziennika z zupełnie losowo poukładanymi rekordami; dostępne opcje: -f, -v, -autocheck, -c, -t

`--create-test` – utworzenie pliku dziennika zawierającego losowe transakcje, które zaczynają się i kończą w sensowny sposób; dostępne opcje: -f, -v, -autocheck, -c, -t

`--create-hc-test` – utworzenie pliku dziennika ustalonej w budowie, zawierającego zestaw transakcje do przetestowania mechanizmu budowy planów odtwarzania; dostępne opcje: -f, -v, -autocheck

`--dd-old` – buduje polecenie `dd` wyświetlające starą wartość dla rekordu aktualizacji; przydatne dla narzędzi testujących;

`--dd-new` – buduje polecenie `dd` wyświetlające nową wartość dla rekordu aktualizacji; przydatne dla narzędzi testujących;

Powyższe akcje mogą zostać sparametryzowane za pomocą następujących opcji:

`-f=<filename>` – nazwa pliku który program ma czytać lub do którego ma zapisać wynik

`-v=<level>` – ilość informacji o tym co dokładnie robi program; przyjmuje wartość od 0 do 2, gdzie 0 to mało informacji; opcja przydatna przy testowaniu; wartość domyślna 0

`-c=<number>` – ilość tworzonych rekordów w pliku; wartość domyślna 1000

`-t=<number>` – ilość jednocześnie uruchomionych wątków zapisujących rekordy w pliku; przydatne przy testowaniu wielowątkowości; wartość domyślna 5

`-cp=<0|1>` – czy używać punktów kontrolnych przy analizie dziennika; 0 - nie, 1 - tak; wartość domyślna 0

`-autocheck` – sprawdza poprawność i spójność wewnętrzną rekordów

`-show-offset` – wypisuje położenie (*offset*) rekordu w pliku; ta opcja bardzo ułatwia obróbkę dzienników za pomocą Linuxowego polecenia `dd`.

`-show-dd` – wypisuje polecenie `dd` umożliwiające podejrzenie starej i nowe zawartości rekordu aktualizacji; przydatne dla narzędzi testujących;

Poniżej przykładowe wywołanie narzędzia z parametrami, a także fragment jego wyniku.

```
[bart@brachypelma logmaster]$ ./logmaster --cat -show-dd -show-offset
0 : <START>
8 : <BEGIN 0>
20 : <UPDATE 0, 65296 : 1 , '[...]', 16384 b, '[...]', 0 : 16384 b>
   old value: dd bs=1 if=/tmp/sblogs count=16384 skip=48
   new value: dd bs=1 if=/tmp/sblogs count=16384 skip=16432
32820 : <UPDATE 0, 65296 : 0 , '[...]', 16384 b, '[...]', 0 : 16384 b>
   old value: dd bs=1 if=/tmp/sblogs count=16384 skip=32848
   new value: dd bs=1 if=/tmp/sblogs count=16384 skip=49232
65620 : <UPDATE 0, 1 : 0 , '[...]', 16384 b, '[...]', 0 : 16384 b>
   old value: dd bs=1 if=/tmp/sblogs count=16384 skip=65648
   new value: dd bs=1 if=/tmp/sblogs count=0 skip=65648
   new value: dd bs=1 if=/tmp/sblogs count=16384 skip=82032
   new value: dd bs=1 if=/tmp/sblogs count=0 skip=82032

[...]
```

A.4. Narzędzia testowe

W katalogu `/praca/tools` znajdują się dwa skrypty testowe.

A.4.1. compare

Skrypt ten jest ściśle zależny od programu `logmaster`. Jako parametr wywołania przyjmuje jeden argument określający kolejny rekord aktualizacji w pliku dziennika. Przykładowe wywołanie poniżej, wypisuje na ekranie kolejno:

1. sumę md5[1] fragmentu lub strony odpowiadającej pierwszemu rekordowi aktualizacji w dzienniku
2. sumę md5 starej wartości zapisanej w rekordzie aktualizacji
3. sumę md5 nowej wartości zapisanej w rekordzie aktualizacji

```
[bart@brachypelma tools]$ ./compare 1
baza d7ea2a692c5814268255f8a7599160b5
old  c8faf446cec3c612d65a16f1ca24ac3e
new  b21decb5e57857e7d5efd0d9069c8897
```

[1] za pomocą linuksowego polecenia md5sum

A.4.2. cyclic_compare

Jego funkcją jest podwójne wywołanie w pętli skryptu `compare`. Jest to bardzo przydatne przy testowaniu zapisów w dzienniku i plikach bazy danych, co zostało opisane w rozdziale 4.4.4. Poniżej kod skryptu.

```
#!/bin/bash

MAX=‘./logmaster --cat | grep UPDATE | wc -l‘

for i in ‘seq 1 $MAX‘; do
    echo Rekord $i:
    ./compare $i
    read

    echo Rekord $i:
    ./compare $i
    read
done
```

```
#!/bin/bash

MAX=‘./logmaster --cat | grep UPDATE | wc -l‘

for i~in ‘seq 1 $MAX‘; do
    echo Rekord $i:
    ./compare $i
    read

    echo Rekord $i:
    ./compare $i
    read
done
```

Dodatek B

Typy rekordów dziennika

Implementacja wykorzystuje następujące typy rekordów dziennika. W zależności od opcji kompilacji (np. włączenie lub wyłączenie trybu „przycinania") pewne typy rekordów mogą się nie pojawiać.

typ rekordu	**parametry**	**opis**
$< START >$	brak	określa rozpoczęcie zapisywania do dziennika po włączeniu serwera; przy odtwarzaniu ma identycznie znaczenie jak $< CKPT >$; nie występuje przy włączonej opcji przycinania, gdyż dziennik jest w tym miejscu przycinany
$< STOP >$	brak	określa zakończenie pracy serwera; jeżeli występuje na końcu pliku, po włączeniu serwera, oznacza to że serwer został poprawnie wyłączony
$< BEGIN(T_i) >$	identyfikator transakcji	rozpoczęcie transakcji o podanym identyfikatorze
$< UPDATE(T_i, f_{id}, p_{id}, v_{old}, v_{old_{size}}, v_{new}, v_{new_{size}}, v_{offset}) >$	identyfikator transakcji, identyfikator pliku i strony, poprzednia wartość strony (i wielkość), nowa wartość strony (i wielkość), przesunięcie	rekord aktualizacji służy do przechowywania informacji o zmianie dokonanej przez transakcję w bazie danych – stanie przed i po; więcej w rozdziale 2, a także na rysunku 4.7
$< ABORT(T_i) >$	identyfikator transakcji	przerwanie transakcji o podanym identyfikatorze; na etapie odtwarzania należy taką transakcję wycofać
Ciąg dalszy na następnej stronie...		

Tabela B.1 – ciąg dalszy z poprzedniej strony

typ rekordu	**parametry**	**opis**
$< COMMIT(T_i) >$	identyfikator transakcji	zakończenie transakcji o podanym identyfikatorze; na etapie odtwarzania należy taką transakcję powtórzyć
$< START\ CKPT(T_1, ..., T_n) >$	lista identyfikatorów transakcji	określa moment rozpoczęcia punktu kontrolnego dla wymienionych transakcji; więcej w rozdziałach 2.3.5 i 2.3.4
$< END\ CKPT >$	brak	zakończenie punktu kontrolnego
$< CKPT >$	brak	pojawia się w sytuacji gdy *menedżer transakcji* nie przetwarza żadnych transakcji; jeżeli włączona jest opcja „przycinania" to dziennik w tym miejscu jest przycinany i rekord ten nie występuje
$< START\ DUMP >$	brak	określa miejsce rozpoczęcie niespoczynkowej archiwizacji; więcej w rozdziale 3
$< END\ DUMP >$	brak	określa moment zakończenia niespoczynkowej archiwizacji; dziennik jest archiwizowany pomiędzy punktem rozpoczęcia i zakończenia archiwizacji

Bibliografia

[GMUW06] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, *Database Systems: The Complete Book*, Pearson Education, Inc., publishing as Prentice Hall, 2006

[Sub04] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych, 2004

[CJD95] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Company, Inc., 1995

[TP] Tadeusz Pankowski, *Odtwarzanie bazy danych*, http://www.put.poznan.pl/~pankowsk/

[LBRS04] Prajakta S. Kalekar, Shruti P. Mahambre, Anirudha U. Bodhankar, *Log Based Recovery System*, Indian Institute of Technology, Powai, Mumbai, 2004

[TCCB04] Thomas Connolly, Carolyn Begg, *Database Systems: A Practical Approach to Design, Implementation and Management*, Addison-Wesley Publishing Company, Inc., 2004

[WRS98] W. Richard Stevens, *Programowanie zastosowań sieciowych w systemie Unix*, Wydawnictwa Naukowo-Techniczne, 1998

[KS00] Khalid Sayood, *Introduction to Data Compression*, Morgan Kayfmann Publishers, 2000

[BRJ01] G. Booch, J. Rumbaugh, I. Jacobson, *UML przewodnik użytkownika*, Wydawnictwa Naukowo-Techniczne, 2001

[WO27] Wilhelm Occam, *Summa logicae*, (przed 1327), Paryż 1448, Bolonia 1498, Wenecja 1508, Oxford 1675

[CV04] John Lakos *C++. Projektowanie systemów informatycznych. Vademecum profesjonalisty*, Wydawnictwo Helion, 2004

[EBBC03] Eric M. Burke, Brian M. Coyner *Java Extreme Programming Cookbook*, O'Reilly Publishing Company, Inc.,2003

[LCTH03] Lisa Crispin, Tip House *Testing Extreme Programming*, Addison-Wesley Publishing Company, Inc., 2003

[BGH83] P. A. Berenstein, N. Goodman, V. Hadzilacos *Recovery algorithms for database systems*, Proc. 1983 IFIP Congress, North Holland, Amsterdam

[BGH87] P. A. Berenstein, N. Goodman, V. Hadzilacos *Concurency Control and Recovery in Database Systems*, Addison-Wesley Publishing Company, Inc., 1987

[Gr78] J. N. Gray *Notes on database operating systems*, Springer-Verlag, 1978

[GMJB81] J. N. Gray, P. R. McJones, M. Blasgen *The recovery manager of the System R database manager*, Computing Surveys, 1981

[GR93] J. N. Gray, A. Reuter *Transacion Processing: Concepts and Techniques*, Morgan-Kaufman, San Francisco, 1993

[KH98] V. Kumar, M. Hsu *Recovery Mechanism in Database Systems*, Prentice-Hall, Engelwood Cliff, 1998

[HR83] T. Haerder, A. Reuter *Principles of transaction-oriented database recovery – a taxonomy*, Computing Surveys, 19