Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Piotr Turski
Nr albumu: 209301

Implementacja indeksów w systemie LoXiM

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Kwiecień 2008

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 pracy przedstawiono implementację indeksów dla półstrukturalnej bazy danych LoXiM. Opisane zostały wszelkie niestandardowe rozwiązania, których wymagała implementacja. Były to między innymi rezygnacja z dziennika zmian, leniwe wycofywanie transakcji, "bąbelkowe" usuwanie z B^+drzewa, odejście od standardowych strategii buforowania. Zaimplementowano także pewne metody optymalizacji zapytań wykorzystujące niejawne użycie indeksów. Poza szczegółowym przedstawieniem zastosowanych rozwiązań, w wielu miejscach dodatkowo przedstawiono krótki przegląd alternatywnych możliwości.

Słowa kluczowe

LoXiM, indeksy, półstrukturalna baza danych, B+drzewo, Stack Based Approach, wersjonowanie

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.3. Information Storage and Retrieval
H.3.1. Content Analysis and Indexing

Tytuł pracy w języku angielskim

An implementation of indexes in the LoXiM system

Spis treści

Wprowadzenie

Indeksy są strukturami przyspieszającymi wykonywanie pewnych zapytań w bazach danych. Można na nie patrzeć jak na szczególnego rodzaju zbuforowane zapytanie, pozwalające szybko odnajdywać obiekty na podstawie wartości klucza wyszukiwania. Indeksy przechowują pary postaci $\langle klucz, wskaźniki \rangle$ [BMS]. Ta jednoaspektowość wyszukiwania umożliwia ich bardzo efektywną organizację [S].

W ramach laboratorium z Systemów Zarządzania Bazami Danych powstaje implementacja obiektowej bazy danych o nazwie LoXiM. Jest ona oparta na podejściu stosowym SBA i posiada półstrukturalny model danych. Celem pracy było zaimplementowanie indeksów w systemie LoXiM.

Podczas implementacji pojawiły się trzy rodzaje problemów. Pierwszy można nazwać "architektonicznym". Pomimo, że większość zagadnień składowych (jak transakcyjność, współbieżność, buforowanie czy operacje na B^+drzewach) jest znana i dobrze opisana od strony architektury, to rozpatrywanie ich osobno nie oddaje wszystkich problemów pojawiających się w trakcie implementacji. Wynikają one z silnego powiązania ze sobą części składowych i wzajemnych kolizji jakie w takich sytuacjach powstają. Wymagało to opracowania zmian w standardowych algorytmach, tak aby mogły one poprawnie ze sobą współpracować. Druga klasa problemów spowodowana jest przejściem z bazy relacyjnej na półstrukturalną. W tym przypadku trzeba było na nowo zdefiniować większość wymagań. Co właściwie chcemy indeksować? Jak radzić sobie z brakiem zdefiniowanej struktury i typów indeksowanych obiektów? Kolejny rodzaj problemów związany był z implementacją LoXiMa i przyjętych w niej rozwiązaniach.

Powyższe przyczyny sprawiły, że prezentowana implementacja znacznie modyfikuje niektóre ze standardowych rozwiązań. Najważniejsze zmiany to rezygnacja z dziennika zmian, leniwe wycofywanie transakcji, "bąbelkowe" usuwanie z B^+drzewa, odejście od standardowych strategii buforowania (LRU, MRU, clock, FIFO [RG]).

W rozdziale 1 przedstawiono podstawowe zagadnienia dotyczące indeksów. Rozdział 2 opisuje problemy charakterystyczne dla indeksów w półstrukturalnym modelu danych. W rozdziale 3 przedstawiona jest, w sposób ogólny, cała koncepcja budowy modułu indeksów i jej podstawowe założenia. W kolejnych rozdziałach omawiane są elementy składowe, czyli moduł zarządzający indeksami i odpowiedzialny za komunikację z resztą LoXiMa (rozdz. 4), sposób implementacji transakcyjności (rozdz. 5), realizacja indeksowania danych różnego typu (rozdz. 6), zmiany w stosunku do standardowych algorytmów operujących na B^+drzewie (rozdz. 7), realizacja niezależnego i odmiennego od reszty LoXiMa buforowania (rozdz. 8), implementacja współbieżności i unikanie zakleszczeń (rozdz. 9). Rozdział 10 opisuje sposób niejawnego wykorzystania indeksów. W dodatkach znajduje się krótki opis stworzonych testów jednostkowych, składnia umożliwiająca zarządzanie indeksami i jawne ich wywoływanie oraz przykładowy scenariusz pracy z niejawnymi wywołaniami indeksów.

Rozdział 1

Klasyfikacja indeksów

Fizycznie indeks jest strukturą opisującą lub organizującą rekordy dyskowe tak, aby przyspieszyć pewne operacje wyszukiwania danych. Umożliwia on szybkie odnajdywanie rekordów spełniających zadane kryteria nałożone na klucz wyszukiwania [RG].

Ze względu na dużą różnorodność indeksów, istnieje wiele różnych metod klasyfikacji indeksów. Najpopularniejsza opiera się na strukturach wykorzystywanych do budowy indeksu. Zostanie ona szczegółowo przedstawiona w kolejnych podrozdziałach [GUW]. Jednak abstrahując od konkretnych struktur indeksom można się przyjrzeć z nieco wyższego poziomu. Ze względu na organizację pliku danych, indeks może być [BMS]:

- zewnętrzny - struktura indeksu przechowywana jest poza plikiem danych. Tworzony jest specjalny plik indeksu, na którym odbywa się wyszukiwanie zadanych kluczy. Rezultatem takiego wyszukiwania jest adres (bądź adresy) właściwego rekordu (lub strony dyskowej) znajdującego się w pliku danych. Kolejność rekordów w pliku danych jest dowolna. Na pliku danych może być założonych wiele indeksów zewnętrznych. Indeks zewnętrzny może być [RG]:
  - pogrupowany - rekordy danych uporządkowane są w ten sam sposób co pozycje indeksu.
  - niepogrupowany - rekordy danych uporządkowane są inaczej niż pozycje indeksu.
- wewnętrzny - plik danych jest zawarty w pliku indeksu. Indeks określa położenie każdego z rekordów. Na pliku danych może być założony tylko jeden indeks wewnętrzny. Taki indeks jest pogrupowany z definicji.

Ze względu na ilość przechowywanych informacji indeks może być:

- gęsty - każdemu rekordowi danych odpowiada jakaś pozycja w indeksie. Dzięki temu część zapytań może zostać zrealizowana bez odwoływania się do pliku danych. Każdy indeks niepogrupowany musi być gęsty.
- rzadki - pozycje indeksu odpowiadają tylko niektórym rekordom z pliku danych. Najczęściej każdy z takich rekordów jednoznacznie identyfikuje jakąś stronę dyskową. Taki indeks jest mniejszy, jednak nie nadaje się na źródło danych do realizacji strategii *tylko-indeks*.

1.1. B^+drzewa

B^+drzewa są zrównoważonymi drzewami poszukiwań, zoptymalizowane do przechowywania węzłów w pamięci o dostępie blokowym (dyski twarde). B^+drzewo składa się z korzenia,

warstwy pośredniej i liści (gdy jest mało rekordów, sam korzeń może być liściem). Każdy blok indeksu odpowiedzialny jest za pewną przestrzeń wartości klucza, dzieli ją na przedziały i wskazuje kolejne bloki odpowiedzialne za każdy z tych nowych przedziałów. Liście natomiast wskazują na rekordy posiadające dane klucze.

Zaletą B^+drzew jest dobry pesymistyczny czas wyszukiwania oraz aktualizacji indeksu po dodaniu lub usunięciu rekordu. Indeksy te mogą być także wykorzystywane do realizowania zapytań zakresowych.

1.2. Indeksy haszujące

Funkcja haszująca każdej wartości klucza przypisuje jeden z n kubełków. W celu wykorzystania tego mechanizmu do indeksowania, przyporządkowuje się kubełkom adresy bloków. Dzięki temu wiadomo, gdzie wstawiać i szukać rekordu o danej wartości klucza. W takim wypadku dostęp do danych wymaga stałej liczby odwołań do dysku. W każdym kubełku można efektywnie przechowywać pewną określoną ilość danych. Gdy danych przybywa, operacje na tym kubełku stają się coraz mniej efektywne. Gdy w dużej liczbie kubełków jest zbyt wiele danych, indeks przestaje być efektywny.

Indeksy haszujące mogą być efektywniejsze niż B+drzewa jednak nie realizują wyszukiwań zakresowych. Efektywność indeksu zależy od sposobu obsługi nadmiaru danych w kubełku.

- haszowanie statyczne - liczba kubełków ustalana jest podczas zakładania indeksu i nie zmienia się. Gdy dane z jednego kubełka nie mieszczą się w jednym bloku, kubełek staje się listą bloków. Operacje na indeksie wymagają przeglądania takiej listy, a więc wielu dostępów do dysku. Gdy średnia liczba bloków przypadająca na kubełek staje się zbyt duża, należy od nowa zbudować indeks mający większą liczbę kubełków.
- haszownie dynamiczne - liczba kubełków zmienia się wraz z ilością danych.
  - haszowanie rozszerzalne - liczba kubełków zmienia się tak, aby na jeden kubełek przypadał co najwyżej jeden blok danych. Przechowywana jest tablica wskaźników do bloków danych. Kilka kubełków może współdzielić ten sam blok danych. Dostęp do danych wymaga odnalezienia wskaźnika w tablicy, a następnie pobrania właściwego bloku danych. Gdy liczba kubełków okazuje się zbyt mała, powiększa się ją dwukrotnie. Wadą tej metody jest to, że liczba kubełków (a więc i rozmiar tablicy wskaźników) zależy od zagęszczenia wartości funkcji haszującej w dowolnym przedziale. Gdy w indeksie znajduje się wiele wpisów o podobnej wartości funkcji haszującej, tablica nadmiernie się rozrasta, mimo że pozostałych kluczy jest mało i wiele kubełków jest pustych bądź prawie pustych. Taka tablica może zajmować wiele bloków. Co więcej, potencjalnie długotrwały proces jej powiększania wstrzymuje jakiekolwiek operacje na indeksie.
  - haszowanie liniowe - średnia liczba rekordów danych na jeden kubełek nie przekracza pewnej ustalonej wartości. Dopuszczalne są bloki nadmiarowe ale ich średnia liczba wynosi nie więcej niż jeden na kubełek. Gdy liczba kubełków jest zbyt mała aby spełnić te założenia, tworzony jest kolejny kubełek.

1.3. Indeksy wielowymiarowe

Aby opisać indeksy wielowymiarowe, najpierw należy ustalić, jakimi informacjami jesteśmy zainteresowani. Najpowszechniejsze zapytania dla danych wielowymiarowych to:

- zapytania z częściowym dopasowaniem - dla zadanych wartości kilku wymiarów, odnaleźć wszystkie punkty spełniające te ograniczenia.

- zapytania o zakres - odnaleźć wszystkie punkty znajdujące się w zadanym zakresie wartości w kilku wymiarach.

- zapytania o najbliższych sąsiadów - dla zadanego punktu, znaleźć jego k najbliższych sąsiadów.

- zapytania o lokalizację - dla zadanego punktu, znaleźć obszary, wewnątrz których punkt się znajduje.

Różne rodzaje struktur wspomagają wykonywanie części z przedstawionych powyżej zapytań. Najpopularniejsze z nich.

- plik siatkowy - na k-wymiarową przestrzeń nakłada się siatkę. W ten sposób otrzymujemy k-wymiarową macierz k-wymiarowych obszarów. Każdy z obszarów reprezentuje jeden kubełek. Kubełki mogą mieć bloki nadmiarowe. Aby odnaleźć kubełek właściwy dla danego punktu, trzeba znać obszar, do którego punkt należy. W celu efektywnego wyznaczenia właściwego obszaru, dla każdego wymiaru należy utworzyć indeks przechowujący linie nałożonej siatki. Gdy w kubełkach znajduje się zbyt dużo danych, można dołożyć dodatkowe linie do siatki i rozbić już istniejące kubełki. Jednak wybór właściwego miejsca dla nowej linii jest trudny. Ten indeks zazwyczaj sprawdza się przy zapytaniach o zakres i najbliższego sąsiada. Problemem może być duża liczba pustych kubełków.

- podzielone funkcje haszujące - dla każdego z wymiarów stosowana jest inna funkcja haszująca. Właściwy kubełek wybierany jest na podstawie sklejenia wyników tych funkcji. Indeks umożliwia realizację zapytań z częściowym dopasowaniem.

- k-d drzewa - są to drzewa wyszukiwań binarnych. Każdy poziom drzewa odpowiada za wyszukiwanie względem jednego z wymiarów. Wymiary, po których odbywa się wyszukiwanie, powtarzają się cyklicznie na kolejnych poziomach. Liście są blokami obejmującymi tyle rekordów ile tylko się w nich zmieści. Indeks umożliwia wykonywanie zapytań z częściowym wyszukiwaniem, zapytań o zakres oraz o najbliższego sąsiada.

- drzewa ćwiartek - k-wymiarowa przestrzeń zostaje podzielona na k-wymiarowe kostki. Jeśli wszystkie rekordy z kostki v mieszczą się w jednym bloku, to v jest liściem. W przeciwnym przypadku v jest węzłem wewnętrznym, a jego potomkowie to kostki powstałe po podziale v na ćwiartki. Drzewa ćwiartek są szczególnym rodzajem k-d drzew. Obszar dzielony jest zawsze na cztery równe podobszary, co nieco ułatwia implementacje, lecz może powodować gorsze zrównoważenie drzewa. Drzewa ćwiartek umożliwiają analogiczne operacje jak k-d drzewa.

- R-drzewa (drzewa obszarów) - k-wymiarowa przestrzeń jest podzielona na k-wymiarowe obszary (niekoniecznie rozłączne i niekoniecznie pokrywające całą przestrzeń), z których każdy może posiadać podobszary. Informacje o obszarach przechowywane są po kilka w bloku (podobnie jak w B^+drzewie). Każdy z takich węzłów dzieli obszar poszukiwań na wiele podobszarów i kieruje przeszukiwanie do właściwego z nich. R-drzewa dobrze nadają się do zapytań o lokalizację. Umożliwiają wyszukiwanie obiektów niepunktowych.

- indeksy bitmapowe - zbiór wektorów bitowych. Długość każdego z tych wektorów jest równa liczbie indeksowanych rekordów. Wektorów jest tyle, ile różnych wartości indeksowanego pola. Każdy z wektorów odpowiada za inną wartość indeksowanego pola. Jeśli indeksowane pole rekordu o numerze v jest równe k, to w wektorze odpowiadającym wartości k na pozycji v znajduje się 1. Jeśli pole jest różne od v, to w wektorze znajduje się 0. Wykonując na wektorach bitowe operacje *AND* i *OR* możemy otrzymać wektor informujący, które rekordy odpowiadają zadanym kryteriom wyszukiwania. Indeks bitmapowy umożliwia wykonywanie zapytań z częściowym dopasowaniem oraz zapytań zakresowych. W celu zmniejszenia rozmiaru wektorów stosuje się kompresję.

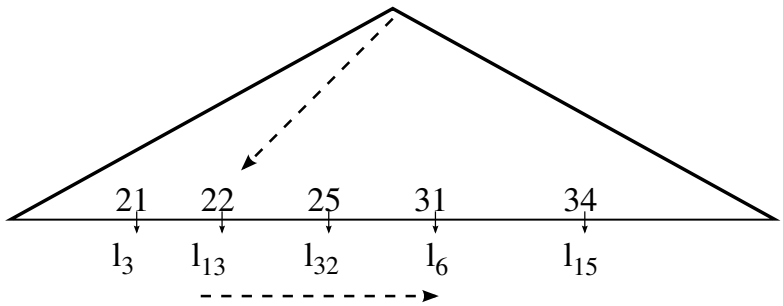
Rozdział 2

Indeksy w bazie półstrukturalnej

W bazach relacyjnych z jedną krotką związana jest jedna wartość indeksu. Indeksując pracowników po wieku i mając następującą tabelę, indeks przyporządkowuje wartościom pola *age*

```
int4        id (primary key),
varchar(20) name,
number(3)   age,
...
```

identyfikator wiersza, w którym ta wartość występuje. W bazie półstrukturalnej sytuacja wygląda nieco inaczej. Baza składa się obiektów postaci $\langle id, name, v\rangle$, gdzie *id* jest unikatowym identyfikatorem obiektu, *name* jest nazwą obiektu, a v jest zawartością obiektu i może być wartością prostą (liczbą lub napisem), zbiorem innych obiektów lub identyfikatorem innego obiektu [S]. Trzeba się więc zastanowić, co właściwie chcemy indeksować. Można zażądać, aby indeksowane były dowolne ścieżki (np. `emp.department.address.street`), jednak w ogólnym przypadku są duże kłopoty z efektywną aktualizacją tego typu indeksów (nazywanych ścieżkowymi). Prezentowana implementacja jest pierwszą próbą włączenia indeksowania do systemu LoXiM i skupia się tylko na indeksowaniu korzeni po ich polach. Indeks będzie zatem przyporządkowywał wartościom indeksowanego pola identyfikator obiektu, który zawiera to pole. Implementacja ma umożliwiać zapytania zakresowe, oparta jest więc na B$^+$drzewach. Załóżmy, że chcemy indeksować korzenie o nazwie *emp* po polu o nazwie *age* (pracowników po ich wieku). W bazie półstrukturalnej sytuacja jest nieco bardziej skomplikowana, gdyż



Rysunek 2.1: Wyszukiwanie przy użyciu B$^+$drzewa. Przerywana linia oznacza kierunek wykonywania zapytania o pracowników w wieku od 22 do 31 lat. Zwrócone zostaną trzy obiekty: $l_{13},\ l_{32},\ l_6$

indeksowane obiekty mogą mieć różną strukturę. Zakładając, że obiekt o identyfikatorze l_1

jest korzeniem, rozważmy takie obiekty:

1. $< l_1,\ "emp",\ \{< l_2,\ "age",\ 25\} >$
2. $< l_1,\ "emp",\ \{< l_2,\ "age",\ 25 >,\ < l_3,\ "age",\ 30 >\} >$
3. $< l_1,\ "emp",\ \{< l_2,\ "age",\ 25 >,\ < l_3,\ "age",\ "trzydzieści" >\} >$
4. $< l_1,\ "emp",\ \{< l_3,\ "age",\ \{< l_4,\ "years", 25 >\} >\} >$
5. $< l_1,\ "emp",\ \{\} >$
6. $< l_1,\ "emp",\ 5 >$
7. $< l_1,\ "emp",\ l_2 >$

Mimo, że część powyższych obiektów być może nie ma sensu w przypadku przechowywania wieku pracowników, to jednak wszystkie są całkowicie poprawnymi obiektami i nic nie stoi na przeszkodzie, aby użytkownik dodał je do bazy. W jaki sposób indeks powinien obsługiwać takie obiekty? Możliwych było kilka rozwiązań:

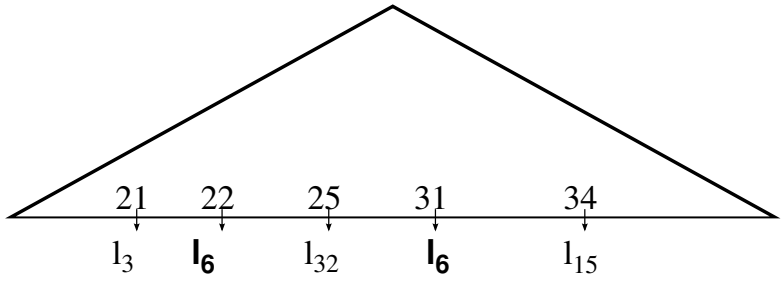
1. Zwracać błąd, gdy nastąpi próba wstawienia do indeksu danych niewłaściwego typu. Jest to wyjście najprostsze z perspektywy programisty.

2. Sprawić, aby wszystkie te typy były ze sobą porównywalne. Należałoby wybrać sposób porównywania - każdy obiekt musiałby zwracać swoją "wagę" (napis, liczbę), która mogłaby posłużyć do porównania dwóch dowolnych obiektów. Bardzo trudno byłoby jednak dobrać odpowiednią funkcję wagi, która dałaby oczekiwany porządek.

3. Każdy typ indeksować osobno. Pytanie o wszystkich pracowników starszych niż 20 lat już na początku odrzucałoby wszystkie obiekty *emp* niezawierające podobiektu *age* będącego liczbą. Wymagałoby to dla każdego indeksu budowania kilku drzew.

O wyborze zadecydowały względy praktyczne. Załóżmy, że mamy w bazie wszystkie korzenie wymienione powyżej. Co będzie wynikiem zapytania `emp where age > 20`? Takie zapytanie zwróci błąd. Ze względu na silną nieregularność danych, takich zapytań nie należy stosować, gdy nie jesteśmy pewni, jakie obiekty znajdują się w bazie [S]. Takie zapytanie miałoby sens, gdyby w bazie każdy pracownik miał podobną strukturę. Z tego powodu wybrana została opcja pierwsza.

Podczas zakładania indeksu należy podać jego typ (typ pól, po których będzie indeksowanie). Może to być typ int, double lub string. Oprócz typu należy także podać nazwę korzeni, które będą indeksowane oraz nazwę pola, po którym korzenie będą indeksowane. W omawianym przypadku wygląda to tak: `create int index on emp(age)`. Przyjęto, że indeks zostanie założony tylko, jeśli każdy korzeń o nazwie *emp* będzie posiadał dokładnie jeden podobiekt o nazwie *age*, który jest liczbą. Można by zezwolić, aby indeksowane pola pojawiały się wielokrotnie wewnątrz jednego obiektu (np. indeksować pracowników po wszystkich imionach). Jednak podczas wyszukiwania należałoby uważać, aby tego samego obiektu nie zwracać wielokrotnie. Zakładając, że *emp* o identyfikatorze l_1 jako jedyny ma dwa imiona, wówczas proste przeglądanie indeksu zwróciłoby ten obiekt dwukrotnie (rys. 2.2). Spamiętywanie wszystkich dotychczasowych obiektów, przy dużych zbiorach danych, byłoby kosztowną operacją. Zatem próba założenia indeksu, gdy któryś z obiektów nie spełnia wyżej wymienionych kryteriów, zakończy się błędem. Podobnie, jeśli do już istniejącego indeksu nastąpi próba dodania "nielegalnego" obiektu, zakończy się ona niepowodzeniem.



Rysunek 2.2: Obiekt l_6 z dwoma podobiektami o nazwie age

Nie są to jedyne operacje, po których w indeksie mogłyby pojawić się dane niewłaściwych typów. Korzeń może stać się nieprawidłowy także podczas jego modyfikacji. Może wtedy nastąpić próba usunięcia lub zmodyfikowania typu podobiektu, po którym następuje indeksowanie (lub próba dodania kolejnego takiego pola). Taka operacja nie powinna się powieść. Modyfikując korzeń, wiemy jakie podobiekty znajdą się w nim po tej operacji. Sprowadza się to zatem do problemu dodawania korzeni. Usuwanie podobiektów jest zupełnie innym problemem. Podczas dodawania można przejrzeć zawartość korzenia i stwierdzić czy jest ona prawidłowa (czy dla każdego indeksu założonego na tym korzeniu istnieje pole wymagane przez ten indeks). W LoXiMie API aparatu wykonawczego przekazuje jedynie informację, że usuwany jest obiekt o danym identyfikatorze logicznym. Nie byłby to żaden problem, gdyby nie fakt, że skład danych nie przechowuje w obiektach wskaźników do ich obiektów nadrzędnych. Prezentowana praca miała jak najmniej ingerować w inne części LoXiMa. Koszt zmiany struktury obiektów przechowywanych w składzie byłby zbyt duży, nie zdecydowano się zatem na tego typu modyfikacje. Nie było więc możliwości stwierdzić czy obiekt jest podobiektem korzenia, a tym bardziej czy jest podobiektem indeksowanego korzenia. Co za tym idzie, nie wiadomo było, czy można go legalnie usunąć. Udało się jednak znaleźć rozwiązanie niewymagające jakichkolwiek zmian w składzie danych. Pomysł opiera się na tym, że zanim aparat wykonawczy wyda polecenie skasowania lub modyfikacji jakiegoś podobiektu, to najpierw musi wyjąć ten obiekt z jego ojca. Wystarczy więc w trakcie wyjmowania obiektu z korzenia przekazać informacje o nazwie korzenia do podobiektu. I wystarczy je przekazać tylko do bezpośrednich podobiektów korzenia, nie ma potrzeby przekazywać tej informacji jeszcze niżej w głąb struktury. W ten sposób choć obiekty w bazie nie wiedzą, który obiekt jest ich ojcem, to w trakcie wykonywania programu informacja ta jest dostępna. Na tej podstawie można stwierdzić czy indeksowanie odbywa się właśnie po podobiekcie, który próbujemy usunąć i co za tym idzie, czy należy zabronić usuwania tego obiektu.

Czasami jednak wymaganie podobiektu określonego typu może być złagodzone. Pewne typy mimo, że różne, możemy chcieć porównywać ze sobą nawzajem. Wówczas typ podawany podczas zakładania indeksu mówi, w jaki sposób wartości będą interpretowane. Jednym z założeń pracy było umożliwienie utworzenia indeksu typu string, także na obiektach typu int i double. Wszystkie takie wartości mają być traktowane jako napisy i porównywane leksykograficznie.

Rozdział 3

Architektura

Poniżej przedstawiono podmoduły składające się na implementację indeksów. Zostaną one szczegółowo opisane w kolejnych rozdziałach.

- menadżer indeksów - fasada [GHJV] odpowiedzialna za komunikację między modułem indeksów a pozostałą częścią LoXiMa. Zarządza listą indeksów znajdującą się w składzie, buduje indeksy, sprawdza legalność dodawania, modyfikacji i usuwania obiektów z indeksu (rozdz. 2), odbudowuje indeksy po awarii. Odpowiada także za poprawną współbieżność w dostępie do struktur reprezentujących indeksy, ale nie za współbieżność samych operacji na B^+drzewach.

- menadżer buforów - synchronizuje operacje wejścia-wyjścia, sprowadza węzły z dysku i zrzuca węzły brudne. Przechowuje w pamięci wczytane węzły i wybiera, które mają być z niej usunięte w pierwszej kolejności. Dostarcza nowe węzły, zarządza węzłami usuniętymi.

- optymalizator zapytań - jeśli jest to możliwe, przekształca zwrócone przez parser drzewo składniowe na takie, które korzysta z istniejących indeksów.

- menadżer transakcji - określa, które wpisy znajdujące się w węzłach są widoczne dla transakcji aktualnie wykonującej zapytanie, jak również które można usunąć, bo już nigdy nie będą widoczne.

- B^+drzewo - dodawanie, usuwanie, wyszukiwanie wpisów i zapewnienie współbieżności tych operacji.

W LoXiMie występują już moduły o nazwie menadżer buforów, menadżer transakcji jak również optymalizator zapytań. Implementacja indeksów zawiera analogiczne moduły całkowicie niezależne od tych, znajdujących się w LoXiMie. W niniejszej pracy nazwy te zawsze będą dotyczyły modułów związanych z implementacją indeksów (chyba, że będzie explicite powiedziane inaczej).

3.1. Skład danych

Jednym z założeń pracy było, aby operacje na indeksach nie były odnotowywane w dziennikach. Powodem takiej decyzji była wydajność. Każda zmiana strony dyskowej ze składu danych jest zapisywana do dzienników a to stanowi dodatkowy koszt. Indeksy jako dane redundantne nie muszą być w ten sposób zabezpieczane. Baza powinna być optymalizowana

pod prawidłową pracę, więc w wypadku awarii można pozwolić sobie na zbudowanie indeksów od nowa. Jedyne informacje podlegające zapisom do dzienników (znajdujące się w składzie) to lista założonych indeksów. Przechowywana jest tak samo jak zwykłe obiekty w LoXiMie. Dane te znajdują się wewnątrz jednego korzenia, którego nazwa jest konfigurowalna (na wypadek konfliktu nazw). W tym korzeniu znajdują się obiekty v_i, z których każdy definiuje jeden indeks. Nazwa obiektu v_i oznacza nazwę indeksu. Każdy z obiektów v_i zawiera trzy podobiekty. Podobiekt o nazwie *root* i wartości będącej nazwą indeksowanych korzeni, podobiekt o nazwie *field* i wartości będącej nazwą pola, po którym następuje indeksowanie, podobiekt o nazwie *type* i wartości liczbowej oznaczającej typ indeksu (int, double, string) (rys. 3.1).

$< l_1,\ "index\_metadata",\ \{$
$\quad < l_2,\ "emp\_age",$
$\quad\quad \{< l_3,\ "root","emp" >, < l_4,\ "field","age" >, < l_5,\ "type", 1 >\}$
$\quad >,$
$\quad < l_6,\ "emp\_name",$
$\quad\quad \{< l_7,\ "root","emp" >, < l_8,\ "field","name" >, < l_9,\ "type", 0 >\}$
$\quad >$
$\} >$

Rysunek 3.1: Przykładowe metadane przechowywane w składzie

Dane, które nie podlegają zapisywaniu do dzienników przechowywane są w oddzielnych plikach. Jeden z nich zawiera węzły wszystkich B$^+$drzew. Drugi zawiera metadane niezbędne do prawidłowej pracy indeksów, utworzone podczas pracy systemu i nadal potrzebne po jego ponownym uruchomieniu. Są to:

- dane menadżera buforów - adres pierwszego i ostatniego węzła na liście węzłów usuniętych oraz adres kolejnego niewykorzystanego jeszcze węzła (rozdz. 3.2).
- dane B$^+$drzew - adres pierwszego i ostatniego węzła na liście każdego z indeksów (rozdz. 3.2) oraz adresy korzeni drzew.
- dane menadżera transakcji - ilość wpisów pozostawionych w poszczególnych indeksach przez transakcje wycofane (rozdz. 5.2).

W trakcie startu systemu menadżer dzienników odpytywany jest o stan systemu. Jeśli poprzednio baza nie została prawidłowo zamknięta, następuje skasowanie plików zawierających węzły oraz metadane i wczytanie ze składu listy indeksów (odtworzonych już przez niezależny system dzienników w LoXiMie). Następnie ustawiane są początkowe wartości wszystkim innym metadanym (kolejny usunięty węzeł, kolekcja wycofanych transakcji itp.) i indeksy zostają zbudowane od nowa.

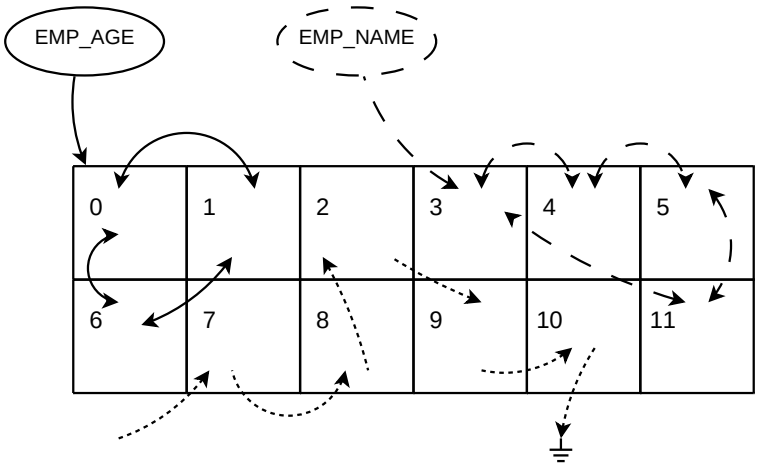
3.2. Węzły

B$^+$drzewo zawiera dwa rodzaje węzłów: węzły wewnętrzne i liście. W węzłach wewnętrznych przechowywane są klucze oraz wskaźniki na dzieci. W liściach przechowywane są klucze i logiczne identyfikatory zaindeksowanych korzeni oraz wskaźnik do sąsiedniego węzła z prawej strony. Dodatkowo w liściach znajdują się niezbędne metadane dotyczące samego wpisu, wymagane do zapewnienia poprawnej transakcyjności (rozdz. 5), takie jak znaczniki czasowe i identyfikator transakcji.

Węzły wszystkich indeksów znajdują się w jednym pliku. Indeksy rosną i maleją, więc muszą mieć możliwość pozyskania nowych węzłów jak i oddania już niepotrzebnych. Choć w literaturze można znaleźć opinie, że węzłów z B^+drzewa nie trzeba usuwać [GUW], to jednym z założeń tej pracy było zaimplementowanie również tej operacji. O ile pobieranie nowych węzłów jest oczywiste (wystarczy powiększyć plik) o tyle usuwanie węzłów wprowadza pewne komplikacje. Najprostszym wyjściem byłoby "zgubienie" takiego węzła. Byłby już nie do odzyskania (bez defragmentacji polegającej na przeglądaniu i przesuwaniu całego pliku indeksu) ale pozostałe operacje pozostałyby proste. To jednak nie jest akceptowalne rozwiązanie, szczególnie jeśli weźmiemy pod uwagę możliwość skasowania całego indeksu. Zatem musi być możliwość ponownego użycia usuniętego węzła. Najprościej zrobić to w chwili gdy jakiś indeks prosi o kolejny węzeł. Wtedy zamiast powiększać plik węzłów wystarczy zwrócić węzeł usunięty. Ponieważ ta metoda umożliwia swobodną wymianę węzłów pomiędzy różnymi indeksami przestrzeń dyskowa jest efektywniej wykorzystywana. Ten pomysł wymaga przechowywania kolekcji węzłów usuniętych. Wystarczy tutaj lista jednokierunkowa. Zawsze pobierany będzie pierwszy węzeł z tej listy. Każdy z usuniętych węzłów będzie wskazywał na kolejny węzeł na liście usuniętych [SQLite]. Musimy tylko przechowywać (także pomiędzy kolejnymi uruchomieniami systemu) adres pierwszego z tych węzłów.

Kolejnym problemem jest usuwanie całego indeksu. Musimy wiedzieć, które strony mają zostać zwolnione. W tym celu konieczne jest posiadanie kolekcji wszystkich bloków używanych przez indeks. Chcielibyśmy więc wszystkie węzły należące do danego indeksu połączyć w listę (analogicznie jak w przypadku węzłów usuniętych). To jednak nie wystarczy. Mając taką listę nie jesteśmy w stanie usunąć węzła z drzewa gdy okaże się, że znajduje się on w środku listy. Potrzebna jest lista dwukierunkowa. Gdy indeks jest usuwany, wówczas cała lista węzłów indeksu dołączana jest (w czasie stałym) na koniec listy węzłów usuniętych. Dlaczego na koniec a nie na początek? Bo początek listy może być przechowywany w buforze (jeśli niedawno został usunięty) aby przyspieszyć dostarczanie nowego węzła. To oczywiście sprawia, że musimy znać adresy ostatniego węzła zarówno na liście indeksu jak i na liście węzłów usuniętych.

Zatem zarządzanie węzłami sprowadza się do przesuwania ich z jednej listy do drugiej. Takie rozwiązanie sprawia, że plik indeksów nigdy się nie zmniejsza ale też nigdy nie rośnie jeśli nie jest to konieczne.



Rysunek 3.2: Listy węzłów o adresach od 0 do 11. Strzałki kropkowane oznaczają listę węzłów usuniętych. Pozostałe strzałki to listy dwóch różnych indeksów.

Każde dodanie i usunięcie z takiej listy wymaga wczytania zarówno poprzednika jak i na-

stępnika. Być może pewną optymalizacją mogłoby być wyłączenie samych list poza węzeł. Może udałoby się w ten sposób obniżyć ilość operacji wejścia/wyjścia. Wtedy jednak należałoby efektywnie wczytywać i buforować dodatkowe struktury reprezentujące te listy.

Każdy węzeł składa się z nagłówka i znajdujących się za nim właściwych danych. Nagłówek składa się z adresu węzła, informacji czy węzeł jest liściem czy nie, ilości wolnego miejsca, poziomu węzła w drzewie (rozdz. 8) oraz adresu węzła następnego i poprzedniego na liście.

Struktura danych właściwych to leżące jeden za drugim kolejne wpisy, zawierające klucz oraz identyfikator logiczny wskazywanego obiektu. Budowa samego wpisu zależy od typu danych skojarzonego z danym indeksem i jest szczegółowo opisana w rozdziale 6.

3.3. Klucze zmiennej długości

Oprócz danych typu int i double w węzłach mogą także znaleźć się dane typu napisowego. Zawsze trudniej składuje się dane zmiennej długości, występują bowiem problemy związane z ich usuwaniem, modyfikowaniem i odzyskiwaniem miejsca po nich [RG]. Jednym z założeń pracy była obsługa takich właśnie kluczy. Co prawda nałożone zostało ograniczenie na maksymalną długość takiego napisu ale nieakceptowalne było przeznaczanie tej samej maksymalnej przestrzeni także na krótkie wpisy.

Tradycyjne podejście zakłada, że węzeł może zawierać co najwyżej pewną maksymalną liczbę wpisów. Należy wtedy pilnować, żeby w każdym momencie węzeł zawierał przynajmniej połowę z tej maksymalnej liczby. Zrezygnowano jednak z tej koncepcji na rzecz traktowania węzła jako ciągu bajtów. W tym podejściu należy pilnować aby węzeł w każdej chwili był przynajmniej w połowie zapełniony, nie zwracając w ogóle uwagi na liczbę wpisów, które zawiera [SQLite, K]. Klucze nie mogą być jednak zbyt długie. W skrajnym przypadku klucz mógłby nie zmieścić się w jednym węźle i należałoby tworzyć listę węzłów na potrzeby jednego klucza. Nawet klucze mieszczące się w węźle ale dostatecznie długie spowodują degenerację drzewa i zbyt dużą jego głębokość. W praktyce podczas wyszukiwań nie stosuje się porównywania długich napisów gdyż jest to nieefektywne. Z tych powodów maksymalna długość klucza została ograniczona w taki sposób aby każdy węzeł zawierał przynajmniej kilkadziesiąt wpisów.

Rozdział 4

Menadżer indeksów

Menadżer indeksów jest fasadą [GHJV] nałożoną na całą implementację indeksów tak, aby zapewnić prostotę komunikacji i odpowiednią izolację od pozostałej części LoXiMa. Jego zadania to:

- wczytanie listy indeksów (ze składu) i ich metadanych (z pliku) podczas uruchamiania LoXiMa
- przebudowanie indeksów w razie awarii
- wykrywanie i reagowanie na zdarzenia istotne dla pracy indeksów
- synchronizacja dostępu do struktur reprezentujących indeksy

Współbieżność operacji na B^+drzewach zostanie szczegółowo omówiona w rozdziale 9. Osobną sprawą jest natomiast współbieżność dostępu do samych drzew, ich tworzenie, usuwanie i odpytywanie. Tworzenie nowego indeksu jest operacją potencjalnie czasochłonną (w bazie może być dużo danych do zaindeksowania). W trakcie takiej operacji nie powinno być możliwe ani ponowne dodanie indeksu o tej samej nazwie, ani dodanie kolejnego indeksu na tym samym korzeniu i polu. Nie można też zezwolić, aby jakiekolwiek zapytanie użyło indeksu przed jego pełnym zbudowaniem. Podobnie, nie można rozpocząć usuwania indeksu, zanim nie zakończą się na nim wszystkie wyszukiwania. Nie ma jednak powodu, aby zabraniać w tym czasie dodawania lub usuwania innych indeksów i absolutnie nie można w tym czasie wstrzymać wyszukiwań na innych indeksach.

Problemy te zostały rozwiązane w następujący sposób. Podczas dodawania nowego indeksu atomowo sprawdzane jest czy w ogóle może on zostać założony (czy nie ma indeksu o takiej nazwie i czy ten korzeń i to pole nie jest już indeksowane). W tym celu na kolekcji indeksów zakładana jest blokada w trybie do zapisu. Jeśli wszystkie warunki są spełnione, dodawana jest nowa struktura reprezentująca indeks z oznaczeniem, że nie jest on jeszcze gotowy do użycia. Następnie zwalniana jest blokada założona na kolekcji indeksów. Dopiero później rozpoczyna się fizyczne budowanie indeksu. Jeśli indeks nie może być dodany, zostanie zgłoszony odpowiedni błąd. Gdy w trakcie budowania zostanie wydane ponowne polecenie dodania indeksu lub wyszukiwania w nim, zostanie znaleziona struktura przechowująca indeks, ale oznaczona jako niedostępna i zostanie zwrócony błąd. Po zbudowaniu indeksu reprezentująca go struktura jest oznaczana jako gotowa do użytku. Od tego momentu transakcje mają dostęp do zawartości indeksu. Usunąć indeks można dopiero, gdy ma się pewność, że żadna transakcja z niego nie korzysta. W tym celu trzeba zliczać, ile transakcji wykonuje operacje na danym B^+drzewie. Gdy zostaje wydane polecenie usunięcia indeksu, atomowo odnajdywana jest reprezentująca ten indeks struktura i zostaje oznaczana jako niedostępna. Następnie

oczekuje się, aż zakończona zostanie ostatnia operacja na indeksie (nowe operacje nie będą rozpoczynane, bo indeks jest niedostępny). Dopiero wtedy następuje usunięcie indeksu.

Każda transakcja zakłada blokadę na kolekcji indeksów tylko na czas odnalezienia właściwego indeksu i poinformowaniu go, jaką operację zacznie wykonywać. Nie ma w tym miejscu oczekiwania na jakiekolwiek inne zdarzenia. Dzięki temu nie występuje problem zakleszczenia czy zagłodzenia.

Należało jeszcze wybrać miejsca wpięcia indeksów do LoXiMa. Trzeba było wybrać istotne zdarzenia, o których moduł indeksów musi być powiadamiany, jeżeli ma poprawnie pracować. Te zdarzenia to:

- dodawanie nowego korzenia - być może będzie to korzeń indeksowany i trzeba będzie uaktualnić indeksy.
- usuwanie korzenia
- modyfikacja obiektu - być może modyfikowany jest indeksowany korzeń i po modyfikacji pole, po którym indeksujemy będzie miało inną wartość. LoXiM nie udostępnia odrębnego API odpowiedzialnego za modyfikację korzeni i obiektów zagnieżdżonych. Taka modyfikacja może się nie powieść, jeżeli po modyfikacji indeksowany korzeń miałby nieprawidłową (z punktu widzenia indeksów) zawartość.
- usuwanie obiektu - nie można pozwolić na usunięcie podobiektu, po którym indeksujemy.
- rozpoczęcie transakcji - skoro indeksy nie polegają na składzie, muszą same realizować wszelkie aspekty transakcyjności. W tym celu muszą znać stan wszystkich transakcji w systemie.
- zatwierdzenie transakcji
- wycofanie transakcji

We wszystkich tych miejscach dodano wywołania odpowiednich metod modułu indeksów. Informacje o tych zdarzeniach są wystarczające do prawidłowego uaktualniania zawartości indeksów i wybierania wpisów widocznych dla poszczególnych transakcji (rozdz. 5).

Rozdział 5

Transakcyjność

Operacje na indeksach, tak jak wszystkie inne operacje na bazie danych, muszą podlegać transakcjom. Kontrola transakcyjności może opierać się na zamkach lub znacznikach czasowych [GUW2]. Jednak skoro zrezygnowano z odnotowywania w dziennikach operacji na indeksach (rozdz. 3.1), to nie ma możliwości wycofania transakcji i przywrócenia stanu sprzed jej rozpoczęcia bez użycia znaczników czasowych i wersjonowania. Co więcej, indeks jest strukturą mającą przyspieszać wykonywanie zapytań, transakcje nie powinny na nim czekać, aż inna transakcja się zakończy. Z tych powodów w przedstawianej implementacji zastosowane zostało wersjonowanie.

Ponieważ wszelkie dane będą trzymane poza składem, należy zaimplementować funkcjonalność, którą w ten sposób stracimy. Nie musimy się jednak martwić awariami. Jeśli takowa nastąpi, indeks zostanie przebudowany na podstawie poprawnych danych znajdujących się w składzie. Co więc należy zaimplementować? Zgodnie z warunkami *ACID* [BMS] cechy poprawnej transakcji to:

- atomowość - transakcja wykona się w całości albo w ogóle. Chociaż nie rozpatrujemy przypadku awarii, to i tak musimy poprawnie obsługiwać wycofane transakcje. Zmiany wprowadzone przez te transakcje nie mogą być widoczne dla jakiejkolwiek innej transakcji.

- spójność - po wykonaniu transakcji nie zostaną naruszone więzy integralności, system pozostanie spójny. Ponieważ indeksy są danymi redundantnymi wystarczy, aby wszelka kontrola integralności została wykonana na poziomie składu. Należy jednak uważać na spójność w obrębie samego indeksu. W indeksie może znajdować się kilka wersji tego samego obiektu (wstawionych zarówno przez transakcje aktywne jak i wycofane) i zawsze należy wybrać tę, która powinna być pokazana danej transakcji.

- izolacja - transakcja pracuje na bazie, jaką zastała w chwili rozpoczęcia i nie widzi zmian dokonywanych przez inne transakcje. Ten aspekt transakcyjności został całkowicie utracony w chwili rezygnacji ze składu i wymagał zaimplementowania od nowa.

- trwałość - zmiany dokonane przez zatwierdzone transakcje pozostają na stałe widoczne w systemie. Istotą B^+drzewa jest fakt składowania go w pamięci trwałej. Ponieważ nie przejmujemy się awariami, trwałość nie wymaga szczególnej uwagi. Należy jedynie upewnić się, że przed zamknięciem systemu wszystkie węzły zostaną zrzucone na dysk.

W następnych rozdziałach wygodnie będzie używać następujących definicji:

Definicja 5.1 *Poprzez $b(t_i)$ i $e(t_i)$ oznaczmy moment, odpowiednio, rozpoczęcia i zakończenia transakcji t_i.*

Definicja 5.2 *Transakcję t_i nazwiemy* wcześniejszą *niż t_j, wtedy gdy $b(t_i) < b(t_j)$.*

5.1. Wersjonowanie

Jak dokładnie zostało zaimplementowane wersjonowanie? Przede wszystkim żadna modyfikacja wpływająca na indeks nie powoduje oczekiwania na zakończenie jakiejkolwiek transakcji. Każda zmiana jest wstawiana jako kolejny wpis. Zarówno dodanie indeksowanego obiektu do bazy, modyfikacja, jak i usunięcie realizowane jest jako dodawanie kolejnych wpisów do B^+drzewa. Wpisy te od razu (bez oczekiwania na zakończenie transakcji) mogą być odczytane przez wszystkie inne transakcje. Podstawą poprawnej izolacji jest przesiewanie wpisów, które powinny być widoczne dla aktualnie wyszukującej transakcji. Które wpisy należy odsiać? Transakcja t_i powinna widzieć zmiany dokonane przez transakcję t_j, wtedy i tylko wtedy, gdy t_j została zatwierdzona zanim rozpoczęła się t_i.

Do tego celu wystarczyłoby, żeby do każdego wpisu dodać informację o czasie zatwierdzenia transakcji. Niestety nie jest to możliwe, bo wpisy są wstawiane wcześniej niż następuje zatwierdzenie transakcji. W chwili zatwierdzenia mogą być w węźle, który został już zrzucony na dysk, więc późniejsze nadawanie wpisom czasu zatwierdzenia transakcji byłoby bardzo kosztowne. Baza powinna być optymalizowana pod zatwierdzanie transakcji, więc to rozwiązanie nie jest do zaakceptowania.

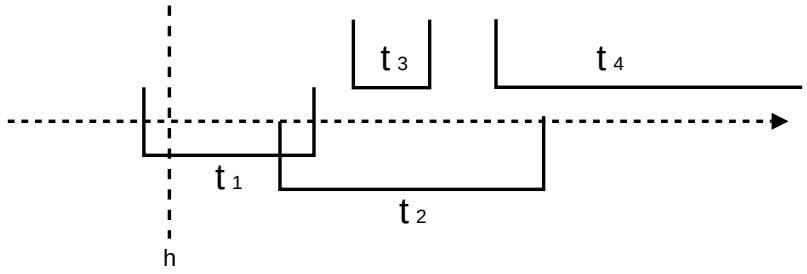
Jedyne dane jakie możemy podać w trakcie wstawiania wpisu to identyfikator transakcji, z którą wpis jest związany oraz czas wstawienia wpisu. Zatem podczas wyszukiwania danych w indeksie są to jedyne informacje, na podstawie których należy określić, czy wpis ma być widoczny dla transakcji, czy nie.

W ten sposób natychmiast można sprawdzić, czy wpis należy do transakcji bieżącej. A co z transakcjami zatwierdzonymi? Najprostszym rozwiązaniem byłoby trzymać odwzorowanie przyporządkowujące transakcji czas jej zatwierdzenia. Wtedy widząc kolejny wpis wystarczy sprawdzić, kiedy transakcja została zakończona i na tej podstawie można by stwierdzić, czy wpis powinien zostać wyświetlony, czy nie. Jednak taka struktura, wraz z pracą systemu, stale by rosła, a to nie jest akceptowalne rozwiązanie. W jakiś sposób trzeba tę strukturę zmniejszać. Przeanalizujmy dokładnie, czego potrzebujemy. Dla bieżącej transakcji t_k, musimy wiedzieć, czy transakcja t_l została zatwierdzona przed czasem rozpoczęcia t_k. Warto zauważyć, że wraz z pracą systemu w końcu niektóre zatwierdzone transakcje stają się tak "stare", że ich wpisy stają się widoczne dla wszystkich aktywnych transakcji. Wygodnie będzie je nazwać.

Definicja 5.1.1 *Transakcję, której wpisy są widoczne dla wszystkich aktywnych transakcji nazwijmy* ustaloną.

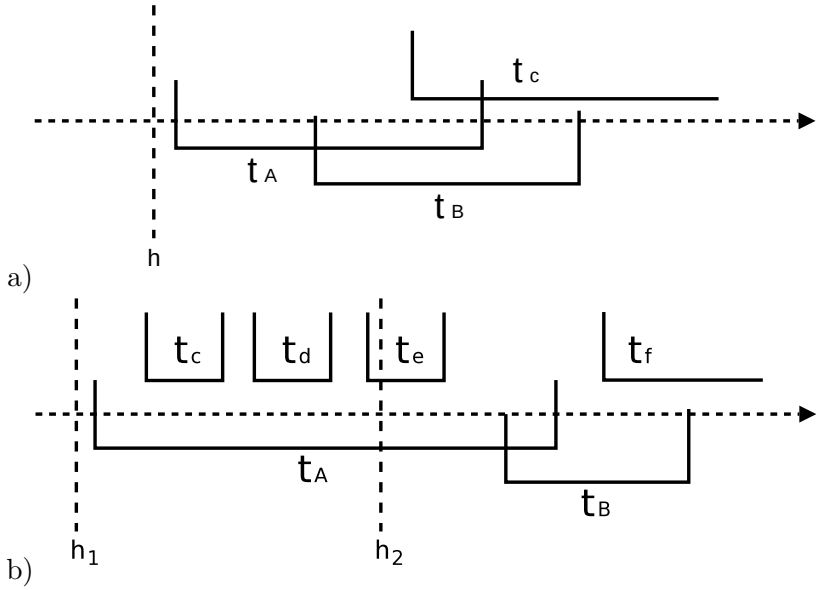
Jeśli transakcja jest ustalona, oznacza to, że została ona zatwierdzona przed rozpoczęciem jakiejkolwiek z aktywnych transakcji. Spośród transakcji ustalonych wybierzmy najpóźniej rozpoczętą, taką że wszystkie wcześniejsze również są ustalone. Oznaczmy ją jako t_l, a chwilę $h = b(t_l) + \epsilon$ nazwijmy *horyzontem* [rys. 5.1]. Wszystkie transakcje rozpoczęte przed horyzontem nie muszą być pamiętane. Nie są nam potrzebne czasy ich zatwierdzenia, bo z góry wiadomo, że ich wpisy będą widoczne dla bieżącej transakcji. Identyfikatorami kolejnych transakcji są kolejne liczby, więc mając tylko identyfikator transakcji (przechowywany razem z wpisem) łatwo stwierdzić, czy transakcja rozpoczęła się przed horyzontem. Z odwzorowania można wyrzucić wszystkie transakcje rozpoczęte przed horyzontem. W ten sposób można efektywnie zmniejszać kolekcję transakcji zatwierdzonych.

Pozostaje jeszcze problem wyliczania horyzontu, a właściwie jego przesuwania. Po starcie systemu horyzont znajduje się tuż przed momentem rozpoczęcia pierwszej transakcji. Kiedy



Rysunek 5.1: W chwili $e(t_2)$ horyzont znajduje się przy początku transakcji t_1. Co prawda t_3 również jest ustalona, ale przed nią jest t_2, która nie jest ustalona.

i jak będzie się przesuwał? Przesunąć się może tylko do przodu i tylko w momencie zakończenia (zatwierdzenia bądź wycofania) pierwszej (najwcześniej rozpoczętej) z aktywnych transakcji, oznaczmy ją t_n. Tylko wtedy mogą (choć nie muszą) pojawić się nowe transakcje zakończone przed rozpoczęciem jakiejkolwiek aktywnej [rys. 5.2]. Zatem jak daleko należy przesunąć horyzont? Po zakończeniu t_n horyzont należy przesunąć tuż za początek ostatniej transakcji



Rysunek 5.2: W chwili $e(t_A)$ horyzont znajduje się przed początkiem transakcji t_A. Po zatwierdzeniu pierwszej z aktywnych w tej chwili transakcji (t_B) horyzont: a) nie zmienia się, b) przesuwa się tuż za $b(t_e)$.

zakończonej po $b(t_n)$, ale przed $b(t_{n+1})$. Dla dalszych rozważań wygodnie będzie uogólnić powyższy wniosek i rozpatrywać pary kolejnych aktywnych transakcji.

Definicja 5.1.2 Horyzontem aktywnej transakcji t_n (oznaczenie: $h(t_n)$) *nazwijmy moment rozpoczęcia ostatniej transakcji zakończonej przed* $b(t_n)$*, ale po* $b(t_{n-1})$

Po zakończeniu pierwszej z aktywnych transakcji (t_n) nowym horyzontem dla całego systemu staje się $h(t_{n+1})$. Horyzont transakcji łatwo jest obliczać. Pomiędzy początkami kolejnych transakcji aktywnych (t_n i t_{n+1}) trzeba monitorować wszystkie kończące się wtedy transakcje. Zbiór tych transakcji nazwijmy $T(n, n+1)$. Ostatnią transakcję z tego zbioru oznaczmy t_i. Teraz można już wyliczyć $h(t_{n+1})$. Będzie to moment rozpoczęcia t_i.

5.2. Wycofanie

W rozdziale 5.1 rozważane były tylko transakcje aktywne i zatwierdzone. Należy jednak poprawnie obsłużyć także wycofanie transakcji. Aby postąpić podobnie jak w pozostałej części LoXiMa, należałoby wycofać transakcję, czyli natychmiast usunąć wszystkie dokonane przez nią zmiany. Ze względu na brak dzienników, z każdą aktywną transakcją należałoby związać wszystkie dokonane przez nią modyfikacje, w chwili wycofywania dotrzeć do tych wpisów w drzewie i po prostu je usunąć. Jednak dotarcie do tych zmian byłoby kosztowne gdyż:

- wpisy te nie mogłyby być adresowane w prosty sposób, np. identyfikator węzła i pozycja w węźle. Od momentu wprowadzenia zmian, do chwili wycofania transakcji, wpisy mogłyby znaleźć się w zupełnie innym węźle i na innej pozycji w wyniku standardowego zachowania B^+drzewa (rozbijanie i scalanie węzłów [rozdz. 7]).
- adresować można trzymając klucze. Ale pamiętanie potencjalnie wielu długich kluczy (napisy) z każdą z wielu aktywnych transakcji mogłoby być bardzo dużym narzutem.
- można też trzymać prefiksy długich kluczy. To jednak wymaga przeszukiwania całego poddrzewa (danego prefiksu) podczas wycofywania każdego z wpisów.

Jednym z założeń tej pracy była optymalizacja bazy pod stosunkowo niewielką liczbę transakcji wycofywanych. Jednak przedstawione powyżej narzuty pamięciowe dotyczą wszystkich transakcji aktywnych, a nie tylko wycofanych.

Ponieważ natychmiastowe usunięcie zmian wycofanych transakcji jest tak kosztowne, zdecydowano się robić to w sposób leniwy. Oznacza to, że wpisy z danej transakcji pozostaną w drzewie nawet po jej wycofaniu. Opisywane w poprzednim rozdziale wersjonowanie potrafi poradzić sobie z tym problemem. Gdy podczas wyszukiwania w drzewie, natkniemy się na wpis z wycofanej transakcji, wystarczy go nie wyświetlać (odsiać). Znowu jednak pojawiają się dwa problemy. Po pierwsze pozostawianie tych wpisów w drzewie to strata przestrzeni dyskowej. W jakiś sposób trzeba się więc tych wpisów pozbywać. Usuwanie zbędnych wpisów zostało dokładnie opisane w rozdziale 7.1. Po drugie, trzeba umieć rozpoznać, czy wpis został dodany przez transakcję wycofaną. Należy przechowywać jakąś kolekcję transakcji wycofanych. I znowu pojawia się znany już kłopot: jak sprawić, aby ta kolekcja stale nie rosła? Rozwiązanie tego problemu ściśle łączy się z problemem usuwania niepotrzebnych wpisów z drzewa. Gdy usuniemy wszystkie wpisy danej transakcji, już nie będzie potrzeby pamiętać, że została wycofana. W ten sposób można zmniejszać kolekcję transakcji wycofanych. Na potrzeby tego mechanizmu, z każdą transakcją związany jest licznik wpisów wstawionych do indeksu przez tę transakcję. Gdy z drzewa usuwany jest wpis należący do takiej transakcji, licznik jest zmniejszany. Gdy dojdzie do zera, w drzewie nie ma już żadnych wpisów danej transakcji i przechowywanie informacji o niej nie jest już potrzebne. Podkreślenia wymaga fakt, że taki licznik nie może mówić ile transakcja dokonała wpisów we wszystkich drzewach, ale musi przechowywać taką informację dla każdego drzewa oddzielnie. Powodem jest to, że użytkownik może w każdej chwili skasować indeks. Jeśli nie będzie wiadomo ile wpisów było w tym indeksie, to nie wiadomo o ile zmniejszyć licznik tej transakcji, a co za tym idzie nie będzie można jej wyrzucić z kolekcji.

Znając obsługę transakcji wycofanych, mamy już całościowy obraz modelu transakcyjności. Wpis może należeć do transakcji będącej w jednym z trzech możliwych stanów. Transakcja t_i może być:

- wycofana - wpis nigdy nie powinien być pokazywany

- aktywna - wpis powinien być pokazany tylko transakcji t_i
- zatwierdzona - wpis powinien być pokazany tylko transakcjom rozpoczętym po zatwierdzeniu t_i

Te trzy stany muszą być rozróżnialne, bo od tego zależy, czy dany wpis powinien być widoczny dla transakcji odpytującej, czy nie. Gdy wykonujemy wyszukiwanie transakcją t_i i natrafiamy na wpis wstawiony przez t_j, algorytm badania widoczności sprowadza się do sprawdzenia:

1. czy $t_j = t_i$. Jeśli tak, to wpis należy do bieżącej transakcji i powinien zostać pokazany.
2. czy czas dodania wpisu jest większy od $b(t_i)$. Jeśli tak, to wpis nie powinien zostać pokazany. To jest operacja nadmiarowa, ale wykonuje się w czasie stałym i może oszczędzić późniejszego przeszukiwania kolekcji.
3. czy t_j jest wycofana. Jeśli tak, to oczywiście wpis nie może zostać pokazany.
4. czy t_j jest w kolekcji transakcji zatwierdzonych.
   (a) jeśli tak, to sprawdzamy, kiedy zatwierdzono t_j. Wpis powinien zostać pokazany wtedy i tylko wtedy, gdy $e(t_j) < b(t_i)$.
   (b) jeśli nie, to t_j albo jest za horyzontem albo jest aktywna. Sprawdzamy, czy jest za horyzontem, bo można to zrealizować w czasie stałym. Jeśli t_j jest za horyzontem to wpis będzie widoczny. W przeciwnym przypadku t_j jest aktywna, a ponieważ $t_j \neq t_i$ to wpis nie będzie widoczny.

Jako optymalizację zastosowano specjalne oznaczanie wpisów należących do transakcji zatwierdzonych, które wyszły poza horyzont. W miejsce identyfikatora transakcji wstawiany jest znacznik "transakcja zawsze widoczna". Dzięki temu, gdy natrafimy na wpis wstawiony przez taką transakcje, przeszukanie kolekcji nastąpi tylko za pierwszym razem. Przy następnym napotkaniu takiego wpisu będzie wiadomo (w czasie stałym), że wpis jest widoczny dla transakcji odpytującej. Wydaje się, że możliwe do zastosowania są kolejne dwie optymalizacje (pomysł pojawił się dopiero w trakcie pisania pracy, więc nie znalazły się one w implementacji):

- oznaczanie znacznikiem "zawsze widoczna" nie tylko transakcji, które wyszły poza horyzont, ale wszystkich transakcji ustalonych. Wystarczy sprawdzić czy transakcja jest zatwierdzona i czy rozpoczęła się przed najwcześniejszą z aktywnych transakcji. Wydaje się, że do tego celu nie są potrzebne żadne dodatkowe struktury, LoXiM przechowuje listę kolejnych aktywnych transakcji, więc ten dodatkowy warunek można zbadać w czasie stałym.
- rozszerzenie wpisów w indeksach o możliwość przechowywania czasu zatwierdzenia transakcji. Przy pierwszym badaniu czasu zatwierdzenia transakcji, która wstawiła wpis, czas ten dołączamy do wpisu. W ten sposób przy następnym napotkaniu tego wpisu, nie musimy już przeszukiwać kolekcji. Jeśli wykorzystamy opisany wcześniej licznik wpisów, to po zapisaniu czasu we wszystkich wpisach wstawionych przez daną transakcję, można ją usunąć z kolekcji jeszcze zanim wyjdzie za horyzont.

Rozdział 6

Różne typy danych

Klucze, po których następuje indeksowanie mogą być różnych typów. Dla każdego typu należy w inny sposób interpretować zawartość węzłów i zarządzać tą zawartością. Zmieniają się typy zwracane przez metody wyszukiwania, typy parametrów metod wyszukujących i wstawiających. Niedopuszczalne jest pisanie nowego B^+drzewa dla każdego kolejnego typu. O ile dla typów o stałej wielkości duża część pracy zostanie wykonana za pomocą typów generycznych (*template* w c++), to jednak będzie to tylko część syntaktyczna. Nadal pozostają kwestie interpretacji danych. Problemy pojawiające się podczas implementacji to:

- Automatyczna konwersja
  Pewne indeksy mogą dopuszczać pojawianie się kluczy różnych typów. Niektóre typy powinny być konwertowane do innych w procesie indeksowania. Jednym z wymagań pracy było, aby indeks typu string pozwalał na dodawanie także kluczy całkowitoliczbowych i zmiennoprzecinkowych w taki sposób, aby były one traktowane jako napisy. Możliwe są oczywiście także inne konwersje pomiędzy typami. Realizacja powyższego założenia miała na celu zaprezentowanie jednej z nich. Wszystkie pozostałe konwersje można zaimplementować analogicznie, bez większego nakładu pracy (jak to pokazano na stronie 29).

- Porównywanie wartości typów użytkownika
  Przechowywane klucze musimy umieć porównywać, żeby efektywnie wyszukiwać je w drzewie. Typy wbudowane przeważnie mają już zaimplementowane metody pozwalające na określenie, która z dwóch wartości jest większa (operatory porównania). Jednak typy zdefiniowane przez użytkownika nie zawsze udostępniają taką funkcjonalność. Często do takiego typu można dopisać odpowiednie operatory (o ile jest dostęp do kodu), choć czasem może to być sztuczne [GGT]. Czasem nie istnieje żaden sensowny porządek liniowy (liczby zespolone), a czasem chcemy porównywać obiekty względem różnych kryteriów i nie powinno być żadnego wyróżnionego operatora porównania.

- Klucze zmiennej długości
  Przy typach o stałej długości wiadomo ile bajtów należy interpretować. Gdy długość wpisu może być zmienna, wówczas należy dodatkowo przechowywać (lub wyliczać) informację o tej długości. Wszystko to sprawia, że bardzo trudno ponownie wykorzystać kod, który zarządza wpisami o stałym rozmiarze.

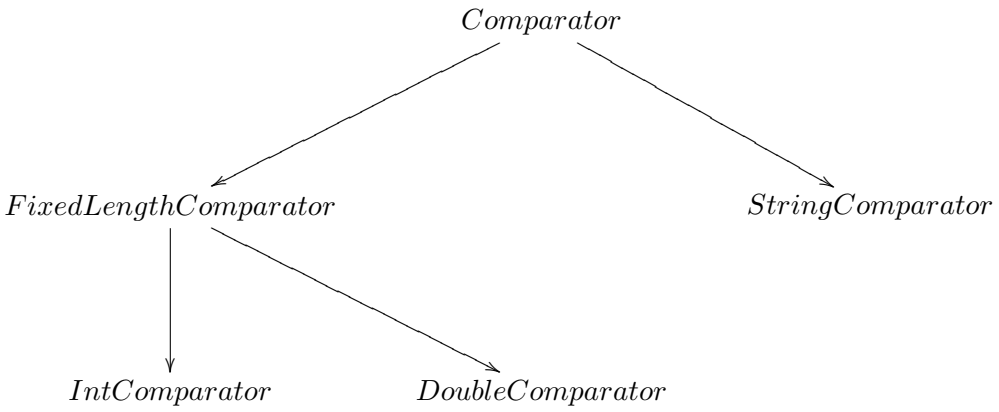
Aby uniknąć pisania różnych wersji B^+drzewa dla różnych typów danych zostało ono zaimplementowane w sposób możliwie niezależny od struktury wpisów. Dodana została warstwa abstrakcji. Drzewo widzi węzeł, który nie jest ciągiem bajtów, ale kolekcją wpisów. Może iterować po tej kolekcji, dodawać i usuwać wpisy. A co z porównywaniem kluczy? Powszechnie

stosowanym wzorcem projektowym jest *comparator* [GGT]. `Comparator` to klasa, do której przeniesiono kod odpowiedzialny za porównywanie innych obiektów. W niniejszej implementacji oprócz porównywania potrzebne jest szeroko pojęte zarządzanie wpisami. W szczególności pobieranie kolejnych wpisów, dodawanie ich i usuwanie. Komparatory używane przez B^+drzewa w tej implementacji mają więc nieco więcej metod niż typowe komparatory, ale ich funkcja jest podobna - przejąć logikę zarządzania obiektami od samych obiektów. Do głównych zadań tego komparatora należy:

- odnalezienie pierwszego wpisu w węźle
- zwrócenie następnika danego wpisu
- wczytanie klucza
- wstawienie wczytanego klucza
- porównanie klucza z węzła z kluczem wczytanym
- usunięcie klucza z węzła

Jak wygląda praca z komparatorami? Z drzewem przez cały czas związany jest komparator odpowiedniego typu. Jest to ustalane już na etapie tworzenia nowego indeksu. Składnia tego polecenia [dodatek B] wymaga podania typu indeksu i na tej podstawie wybierany jest właściwy rodzaj komparatora. Wszystkie operacje wykonywane przez B^+drzewa [rozdz. 7] używają komparatorów. Pobierają je, klonując komparator związany z drzewem. Drzewo pobiera kolejne wpisy z węzła nie bezpośrednio, ale właśnie poprzez komparator. Drzewo nie wie, na jakiego typu danych pracuje. Aby znaleźć właściwe miejsce w drzewie, komparator inicjujemy poszukiwanym kluczem, a następnie jest on odpytywany, czy dany klucz w węźle jest większy od tego, którego poszukujemy, czy nie. Klucz zawarty w komparatorze możemy dodatkowo wstawić do drzewa jako wpis w liściu. To komparator definiuje, jakiego typu dane mogą być indeksowane i jakie typy mogą znajdować się w tym samym indeksie.

Jednym z wymagań pracy było umożliwienie indeksowania po obiektach typu int, double oraz string. Zaimplementowane zatem zostały trzy klasy komparatorów, odpowiadające tym typom wraz z abstrakcyjnymi nadklasami [rys. 6.1]. W ten sposób znacznie łatwiej dodawać



Rysunek 6.1: Zaimplementowana hierarchia komparatorów.

jest kolejne typy danych, które mogą być indeksowane. Operacja taka wymaga dodania kolejnego komparatora. Jednak ze względu na już istniejącą hierarchię klas, jest to znacznie

ułatwione. Kod klasy `DoubleComparator` [rys. 6.2] pokazuje, jak niewiele pracy wymaga dodanie nowego typu. Najistotniejsza jest pierwsza z metod. Trzy pozostałe mają za zadanie

```
class DoubleComparator : public FixedLengthComparator<double> {
  public:

    int setValue(double value) {
      this->value = value;
      return 0;
    }

    DoubleComparator* emptyClone() {
      return new DoubleComparator();
    }

    Comparator::comparatorType getType() const {
      return Comparator::DOUBLE_C;
    }

    string typeToString() const {
      return "double";
    }

};
```

Rysunek 6.2: Kod źródłowy klasy DoubleComparator

zwrócić, odpowiednio: niezainicjowany komparator tego samego typu, znacznik typu komparatora zapisywany do bazy oraz znacznik typu "przyjazny do użytkownika". Pierwsza z metod odpowiada natomiast za zainicjowanie komparatora nową wartością i kontrolę poprawności tego procesu. Indeks typu double powinien zwracać błąd przy próbie dodania obiektu typu string. Jest to zrealizowane w nadklasie `Comparator`. Zawiera ona metody `setValue` z parametrami wszystkich typów, po których dozwolone jest indeksowanie. Wszystkie te metody zwracają błąd informujący o niekompatybilności typów. `DoubleComparator` przeciąża jedną z nich, zezwalając na inicjowanie komparatora liczbami typu double, a co za tym idzie na indeksowanie obiektów tego typu. Co zrobić, aby umożliwić indeksowanie tym samym indeksem nie tylko wartości typu double, ale także typu int? Wystarczy umożliwić zainicjowanie tego samego komparatora wartościami typu int, czyli przeciążyć metodę `int setValue(int)`. Ta metoda będzie wyglądać tak:

```
int DoubleComparator::setValue(int value) {
  this->value = value;
  return 0;
}
```

Wykorzystując ten sam mechanizm, zrealizowano indeksowanie liczb przy użyciu indeksów typu string. Właśnie w metodach `setValue` zrealizowano konwersję liczb do napisów.

To co rzuca się w oczy, to brak jakiejkolwiek metody realizującej porównywanie zawartości komparatora i wpisu w węźle. Zarówno typ int jak i double są porównywalne standardowymi operatorami, więc samo porównanie zostało wyłączone do nadklasy `FixedLengthComparator`.

W przypadku dodania typu użytkownika należałoby przeciążyć jeszcze metodą `compare`, realizującą porównanie.

Oczywiście klasa `StringComparator` jest znacznie bardziej skomplikowana. Obsługuje ona klucze zmiennej długości (nieprzekraczającej jednak pewnej długości maksymalnej (rozdz. 3.3)). Klucz jest złożony. Składa się z informacji o swojej długości oraz właściwej zawartości. Jednak znowu, dopisanie kolejnego typu, który trzyma się tego schematu, nie wymaga wiele pracy. Należy oczywiście zaimplementować metody `setValue` oraz `compare`, aby umożliwić przyjęcie nowego typu i porównywanie. Dodatkowo należy jeszcze przeciążyć metodę `getValue`, odpowiedzialną za wczytanie klucza z węzła (deserializacja ciągu bajtów do obiektu) oraz `putKey`, wykonującą operację odwrotną (serializację klucza).

Oczywiście pozostaje jeszcze dodać "kosmetykę", taką jak składnia, która pozwala na tworzenie indeksów danego typu, pozostałe metody z rys. 6.2, czy ogólnie rzecz biorąc, kod informujący system o istnieniu w bazie kolejnego typu, z którym trzeba współpracować. Jednak ogólna koncepcja i główny nakład pracy sprowadza się do odpowiedniej implementacji komparatorów.

Rozdział 7

Operacje na B$^+$drzewach

7.1. Wstawianie i usuwanie

Wstawianie do B$^+$drzewa bazuje na standardowym, jednoprzebiegowym algorytmie szczegółowo opisanym w [CLR]. Algorytm sprawdza czy w węźle jest miejsce na dodatkowy wpis maksymalnej długości i jeśli nie, to profilaktycznie rozbija go na dwa nowe.

Ze względu na zastosowanie wersjonowania nie ma jawnej operacji usuwania wpisów z drzewa. Usunięcie obiektu z bazy skutkuje dodaniem do drzewa nowego wpisu, który reprezentuje to usunięcie. Drzewo zatem fizycznie rozrasta się, a nie maleje. Po pewnym czasie jednak cześć z tych wpisów staje się zbędna i można się ich pozbyć. Algorytm dodawania został więc zmodyfikowany tak, aby w trakcie, przy okazji przeprowadzane było usuwanie niepotrzebnych już wpisów i węzłów. Nie można tego robić również w trakcie wyszukiwania, ponieważ wtedy każdy węzeł pobierany jest w trybie tylko do odczytu.

Jak wygląda zmniejszanie rozmiaru drzewa? Aby fizycznie usunąć jakiekolwiek dane, muszą pojawić się wpisy "znoszące się nawzajem". Wszelkie informacje o obiekcie l_k mogą być fizycznie usunięte z drzewa w kilku przypadkach:

- transakcja wstawia obiekt l_k, a następnie jest wycofywana
- transakcja wstawiła obiekt l_k, a następnie go usuwa
- transakcja t_i wstawia obiekt l_k, następnie transakcja t_j usuwa obiekt l_k, po czym zmiany obu transakcji stają się widoczne dla wszystkich bieżących i przyszłych transakcji

Pierwszy przypadek jest łatwy do zrealizowania. Usuwamy z drzewa każdy napotkany wpis należący do transakcji wycofanej. Dwa pozostałe przypadki są nieco bardziej skomplikowane. Trzeba znaleźć oba "przeciwstawne wpisy" dotyczące tego samego obiektu. Załóżmy, że wpis w_+ informuje o dodaniu danego obiektu (o kluczu k), a wpis w_- o jego usunięciu. Jeżeli w bazie znajduje się dużo obiektów o kluczu k, to w_+ i w_- wcale nie muszą znajdować się obok siebie. Jeśli w_+ i w_- znajdą się w różnych węzłach, to mogą nigdy nie zostać usunięte.

Żeby takie sytuacje wykryć i usunąć zbędne wpisy, należałoby przeszukiwać wiele kolejnych liści i spamiętywać wszystkie odwiedzone wpisy, a to byłoby zbyt kosztowne. Co gorsza wpisów dotyczących tego samego obiektu może być więcej, gdyż w LoXiMie modyfikacja obiektu została zaimplementowana jako usunięcie i dodanie obiektu o tym samym identyfikatorze. Trzeba wtedy zdecydować, które z tych wpisów można usunąć.

Z tych powodów wszystkie wpisy dotyczące tego samego obiektu są grupowane razem. Dzięki komparatorom (rozdz. 6) nie wymagało to jakiejkolwiek zmiany działania B$^+$drzewa.

Komparatory jako część klucza wykorzystują logiczny identyfikator obiektu związanego z danym wpisem. Gdy oba klucze okazują się równe, następuje porównanie tych identyfikatorów. W ten sposób prawie wszystkie takie wpisy znajdą się w obrębie pojedynczego węzła. Te, które akurat są w różnych węzłach w końcu też znajdą się w tym samym (na skutek dodawania i usuwania innych wpisów). A zatem jest możliwość efektywnego usuwania zbędnych danych. Gdy transakcja usuwa obiekt, który sama wstawiła, algorytm trafia do miejsca, w którym znajduje się wpis oznaczający wstawienie. Nie trzeba więc dodawać kolejnego wpisu i polegać na późniejszym czyszczeniu, od razu można fizycznie usunąć wpis oznaczający wstawienie. W ten sposób rozwiązany zostaje też problem usuwania i dodawania tego samego obiektu w trakcie operacji modyfikacji.

W wyniku usuwania wpisów z węzła (czyszczenia węzła) może się okazać, że ma on mniej niż minimalną dopuszczalną ilość danych. W literaturze można znaleźć opinie, iż usuwanie węzłów z drzewa nie jest konieczne [GUW], jednak jednym z założeń pracy było zaimplementowanie także tej operacji. Aby nie ograniczać współbieżności (rozdz. 9) zostało to zrealizowane na wzór sortowania bąbelkowego. Węzły o zbyt małym wypełnieniu są stopniowo (nie w jednym przebiegu) wypychane w górę drzewa, aż do etapu, na którym usuwany jest korzeń. Gdy po zakończeniu procedury czyszczenia węzła v okazuje się, że ma on zbyt mało danych, nie ma już dostępu ani do ojca (aby zwiększyć współbieżność jest on jak najszybciej zwalniany, szczegóły w rozdziale 9), ani do braci v, więc scalenie lub zrównoważenie v, z którymś z braci (zgodnie ze standardowym algorytmem usuwania [CLR]) nie jest możliwe. Taki węzeł jest więc pozostawiany. Jednak przy ponownym dodawaniu do drzewa, gdy węzeł v znowu zostanie odwiedzony (w trybie do zapisu) będzie to wykryte w chwili, gdy nadal dostępny będzie ojciec v w trybie do zapisu. Wówczas można dokonać wszelkich standardowych operacji usuwania. Można pobrać obu braci (o ile istnieją), odpowiednio rozłożyć pomiędzy nich dane lub scalić dwa węzły, a na koniec dokonać odpowiednich poprawek w ojcu. W wyniku scalania dwóch węzłów w ojcu ubywa jeden wpis. Może się więc zdarzyć, że teraz ojciec v będzie miał zbyt mało danych. Tutaj sytuacja jest podobna, w tym momencie jest już za późno na modyfikację dziadka v i trzeba czekać do następnego napotkania tego węzła.

7.2. Wyszukiwanie

Istnieje wiele sposobów wykorzystania indeksów do przyspieszania wykonywania zapytań (m.in. *unique scan*, *range scan*, *skip scan*, *index join*) [Ora]. Ze względu na obszerność tematu, praca skupia się jedynie na implementacji *range scan*. Oznacza to, że należy zapewnić obsługę wielu różnych zapytań. Główne przypadki to:

1. wyszukiwanie wartości a
2. wyszukiwanie wartości od $-\infty$ do a
3. wyszukiwanie wartości od a do ∞
4. wyszukiwanie wartości od a do b

Liczba przypadków jednak wzrasta, gdyż wyszukiwania nr 2 i 3 mogą mieć dwa różne warianty (w zależności od tego czy chcemy, aby wartość brzegowa zawierała się w zbiorze wynikowym czy nie), a wyszukiwanie nr 4 może ich mieć aż cztery. Co więcej wartość brzegowa może znajdować się w indeksie bądź nie. Czy implementacja musi rozpatrywać wszystkie te przypadki?

Wyszukiwanie danych w B^+drzewie składa się z dwóch etapów. Najpierw trzeba dotrzeć od korzenia aż do odpowiedniego wpisu w liściu, a potem przejść po kolejnych liściach do

momentu zebrania wszystkich potrzebnych danych. Pierwszy etap realizowany jest standardowym algorytmem wyszukiwania dla B^+drzew [CLR]. Znajduje on pierwszy wpis z poszukiwanym kluczem (lub większym, gdy szukanego nie ma). Okazuje się, że drugi etap można opisać w sposób bardzo podobny dla każdego z możliwych przypadków. Dla najbardziej skomplikowanego przypadku, czyli wyszukiwania wartości od a do b, algorytm będzie wyglądał tak:

1. znajdź pierwszą wartość a (lub większą)
2. jeśli znaleziono dokładnie a i nie chcemy, żeby a zawierała się w zbiorze wynikowym, to przejdź do klucza większego od a
3. dodawaj kolejne wartości do wyniku i przesuwaj się póki wartości są mniejsze od b
4. jeśli wartość b ma się znaleźć w zbiorze wynikowym oraz aktualna wartość to b, to przesuwaj się aż do klucza większego od b i dodawaj kolejne wartości do wyniku

Podobnie można zaimplementować pozostałe trzy główne przypadki. Zadanie to zrealizowane zostało poprzez obiekty klas `Constraints` i `LeafTraveller`. LeafTraveller jest obiektem, którego jedynym zadaniem jest poruszanie się po liściach w kierunku rosnących kluczy (do tego celu używa komparatorów [rozdz. 6]) i dodawanie kolejnych wpisów do wyniku lub ich pomijanie. Jest to obiekt, którego główna część interfejsu składa się z metod ustalających parametry przesunięcia (miejsce początkowe, docelowe, czy wpisy dodawać do wyniku czy nie) i metod wykonujących przesunięcie. Najważniejsze publiczne metody klasy LeafTraveller:

1. `travelToEnd()` - przejście aż do końca B^+drzewa
2. `travelToDestination()` - przejście aż do ustawionej wcześniej wartości klucza
3. `travelToGreater()` - przejście do pozycji o kluczu większym niż aktualny
4. `setGrab(bool grab)` - ustalenie czy odwiedzane wpisy będą dodawane do wyniku

Dzięki wykorzystaniu komparatorów trzy pierwsze z wymienionych metod składają się z kilku linijek kodu. Wykorzystują one metodę niepubliczną `travelTo`, przekazując jako parametr odpowiedni warunek zatrzymania się. Dwie ostatnie są najczęściej używanymi metodami klasy LeafTraveller. Największą ich zaletą jest fakt, iż rozwiązują problem ostrości ograniczenia i redukują liczbę rozpatrywanych przypadków.

Obiekty klasy Constraints odpowiedzialne są za obsługę czterech głównych przypadków wyszukiwania. Za każdy z nich odpowiedzialna jest inna podklasa. Realizują one proste, kilkulinijkowe algorytmy, jak ten przedstawiony powyżej. Używają obiektów klasy LeafTraveller. Nadają im znalezioną wartość początkową i kilkakrotnie je przesuwają, dodając wpisy do wyniku lub nie (w zależności od tego czy ograniczenie wyszukiwania jest ostre i czy w drzewie znajduje się wartość brzegowa). Warto też zauważyć, że wyszukiwanie dokładnej wartości a może być potraktowane jako szczególny przypadek wyszukiwania zakresowego i zapisać je jako $\langle a;\ a\rangle$. Jednak ze względu na przejrzystość kodu zostało ono zaimplementowane jako oddzielna klasa.

Specjalnego potraktowania wymaga przypadek, gdy szukamy wszystkich wpisów o kluczu mniejszym od wartości a. Wówczas należy dostać się do skrajnie lewego wpisu w drzewie. W tym celu można by dla każdego drzewa pamiętać najmniejszy zawarty w nim klucz lub adres skrajnie lewego liścia. Wymagałoby to jednak odpowiedniego zarządzania tym wskaźnikiem podczas scalania i rozbijania liści. Dużo prostszym rozwiązaniem jest użycie specjalnego komparatora, który zawsze będzie kierował przeszukiwanie do najmniejszego klucza.

Bardziej szczegółowego omówienia wymaga dodawanie do wyniku kolejnych wpisów, po których przesuwa się obiekt klasy LeafTraveller. Oczywiście dodaje on tylko wpisy, które są dla niego widoczne (zgodnie z rozdziałem 5). To jednak nie wystarcza. Ze względu na wersjonowanie w drzewie znajdują się również wpisy reprezentujące obiekty usunięte. Oznaczają one, że nie można pokazać wcześniejszych wpisów dotyczących tego samego obiektu. Dzięki zgrupowaniu wpisów dotyczących tego samego obiektu można łatwo zdecydować czy i który wpis wyświetlić (przy stałym koszcie pamięciowym). Wystarczy pamiętać wpis z największym znacznikiem czasowym i jeśli oznacza on dodanie obiektu, to obiekt (a dokładniej jego logiczny identyfikator) należy dodać do wyniku.

Rozdział 8

Menadżer buforów

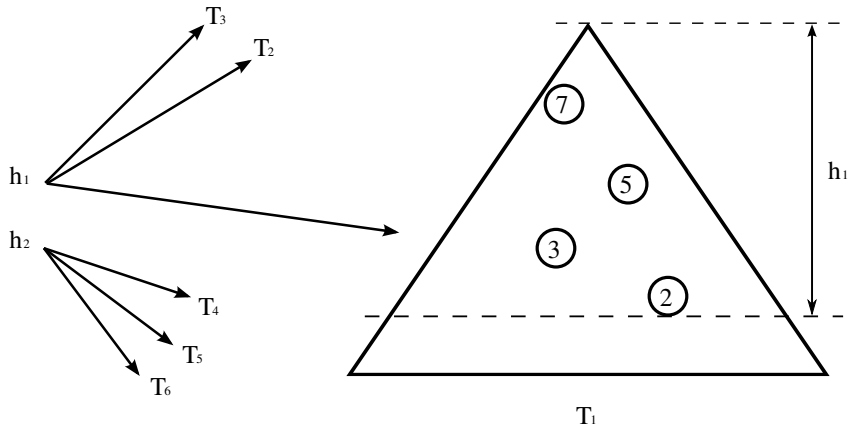
Aby zwiększyć efektywność baz danych stosuje się buforowanie stron dyskowych. Im częściej dane pobierane są z pamięci, tym szybciej działa baza. Z drugiej strony nie można wszystkich stron trzymać w pamięci. Są różne strategie wyboru, które strony należy zrzucić na dysk, aby zrobić miejsce dla innych, a które dłużej przechowywać w pamięci [G, BMS]. Podobnie, w przypadku B^+drzew buforowane są węzły, z których każdy jest stroną (bądź kilkoma kolejnymi stronami [K]) pamięci dyskowej. Jednak strategie odpowiednie dla składu, zastosowane do B^+drzew, nie wypadają najlepiej. W pamięci zawsze warto trzymać strony odwiedzane najczęściej, a najwięcej transakcji korzysta z najwyżej położonych węzłów [CLQS, BMS]. Wobec tego im węzeł dalej od korzenia, tym wcześniej powinien być zwolniony.

Menadżer buforów zajmuje się przezroczystym dla drzew sprowadzaniem węzłów z dysku, przechowywaniem węzłów wczytanych i zrzucaniem ich z powrotem na dysk. W celu wygodnego ukrycia buforowania przed pozostałą częścią systemu zastosowano wzorzec projektowy *proxy* [GHJV]. Drzewo prosząc o węzeł, otrzymuje obiekt klasy `CachedNode`, z którego to dopiero pobiera obiekt typu `Node`, będący stroną pamięci dyskowej (z nałożoną strukturą dla ułatwienia pracy). Jeśli węzeł nie znajduje się w buforze, to podczas pobierania obiektu Node zostanie on sprowadzony z dysku. Każdej transakcji żądającej węzła o tym samym adresie udostępniany jest ten sam obiekt CachedNode. Jest on odpowiedzialny za zarządzanie buforowaniem i współbieżnym dostępem (rozdz. 9) do tej strony dyskowej. Każda prośba o dostarczenie węzła, którego nie ma w buforze, powoduje utworzenie nowego obiektu CachedNode. Pierwsze odwołanie do danych wskazywanych przez ten obiekt powoduje wczytanie strony z dysku. Gdy węzeł przestaje być potrzebny, drzewo jawnie powiadamia o tym menadżera buforów.

Jednak w trakcie pracy bazy bufor przeważnie zawiera już maksymalną liczbę wczytanych węzłów. W takim wypadku, jeśli wszystkie te węzły są aktualnie używane, operacja pobrania kolejnego węzła staje się operacją blokującą i czeka do chwili zwolnienia któregoś z wczytanych węzłów. Gdy są jakieś węzły aktualnie nie używane, wówczas wybierany jest jeden z nich, który zostanie wyrzucony z bufora. Jeśli dane w węźle uległy zmianie, to przed usunięciem z bufora zmiany muszą zostać zrzucone na dysk. Informacja o zmodyfikowaniu zawartości jest jawnie podawana przez drzewo. Nie można domyślnie przyjąć, że skoro węzeł został pobrany w trybie do zapisu, to na pewno został zmodyfikowany. Algorytm dodawania [CLR] z założenia może modyfikować wszystkie węzły na drodze od korzenia do liścia, więc każdy węzeł pobierany jest w trybie do zapisu (rozdz. 9). Jednak w wielu węzłach (zwłaszcza blisko korzenia) żadne modyfikacje nie są wykonywane, nie ma więc potrzeby zapisu.

Węzeł, który jest nieużywany przez żadną transakcję ale znajduje się w buforze, nazwijmy *węzłem wolnym*. Jak już wcześniej wspomniano, do zrzucenia na dysk i zwolnienia pamięci

powinny być wybierane wolne węzły leżące jak najgłębiej w drzewie. Sprawa się nieco bardziej komplikuje, gdy w systemie jest kilka drzew różnej wysokości. W tej implementacji przyjęto wybierać węzeł leżący najdalej od korzenia spośród wszystkich wolnych węzłów we wszystkich drzewach. Ponieważ B^+drzewa rosną w górę, węzłom nie można po prostu przypisać ich głębokości i na tej podstawie szybko znaleźć najgłębszy z nich. Gdy drzewo otrzymuje nowy korzeń, głębokość wszystkich węzłów drzewa wzrasta i należałoby uaktualnić wszystkie zbuforowane węzły. Jedyna stała informacja to odległość węzła od liści. Nazwijmy ją *priorytetem*. Dla każdego indeksu trzeba też znać priorytet korzenia. W ten sposób możemy obliczyć głębokość danego węzła. *Priorytetem drzewa* nazwijmy głębokość najgłębszego z jego wolnych węzłów. Pozostaje jeszcze trzymać globalną strukturę przyporządkowującą istniejącym w systemie priorytetom drzew zbiory tych drzew. Teraz, gdy trzeba wybrać jakiś węzeł do zrzucenia na dysk, wybierany jest zbiór indeksów o największym priorytecie (h_1 w przypadku rysunku 8.1), z tego zbioru dowolny indeks (T_1), a z indeksu węzeł o najmniejszym priorytecie (2).



Rysunek 8.1: Wybór węzła do usunięcia z bufora. W drzewie zaznaczono wszystkie wolne węzły.

Gdy zmienia się wysokość drzewa, wystarczy odnaleźć to drzewo w odwzorowaniu (pod dotychczasowym priorytetem), usunąć z niego i wstawić z nowym priorytetem, mniejszym lub większym o jeden. Nie ma potrzeby uaktualniania informacji o jakichkolwiek węzłach. Utrzymanie tej struktury też nie jest kosztowne. Gdy węzeł staje się wolny, sprawdzamy czy jest on głębszy niż jakikolwiek inny wolny węzeł. Jeśli tak, to wykonujemy analogiczną operację jak przy zmianie wysokości drzewa. Podobnie gdy węzeł przestaje być wolny. Wtedy sprawdzamy, czy był on najgłębszym z wolnych węzłów. Jeśli tak, to zmieniamy priorytet drzewa.

Oprócz bufora węzłów używanych jest również bufor węzłów usuniętych. Usuwany węzeł jest wkładany na listę węzłów usuniętych (rozdz. 3.2), musi więc otrzymać wskaźnik na kolejny element listy. Jest zatem modyfikowany i powinien zostać zrzucony na dysk. Możliwe jednak, że za chwilę drzewo rozrośnie się i będzie potrzebowało nowego węzła. Gdy pobierany jest nowy, zawsze następuje próba odzyskania węzła usuniętego, o ile taki jest. Zatem pozostawienie usuniętego węzła w buforze może zaoszczędzić zrzucenia na dysk. Gdy w buforze trzeba zwolnić miejsce, w pierwszej kolejności na dysk zrzucane są węzły usunięte.

Fizyczna lokalizacja węzłów jest dla drzewa przezroczysta. Sposób adresowania węzłów nie może mieć wpływu na działanie drzewa. Podczas uruchamiania LoXiMa drzewo otrzymuje adres korzenia, który w razie potrzeby przekazywany jest do menadżera buforów. Wczytany

węzeł zawiera adresy dzieci (lub prawego sąsiada, w przypadku liści) i to jest wystarczająca informacja dla prawidłowej pracy drzewa.

Rozdział 9

Współbieżność

Aby wiele transakcji na raz mogło korzystać z tego samego drzewa, należało wprowadzić możliwość blokowania węzłów. Drzewo prosi o węzeł z założoną blokadą do zapisu lub tylko do odczytu. Wszystkie operacje (rozdz. 7) są jednoprzebiegowe, co pozwala zwalniać wyżej położone węzły zanim przeglądanie drzewa dojdzie do liścia. Nigdy na raz nie są blokowane węzły z więcej niż dwóch sąsiadujących poziomów.

W celu wyeliminowania potencjalnych zakleszczeń, jednoznacznie ustalono kolejność blokowania węzłów. Blokowanie będzie się odbywało z góry na dół i z lewej na prawo. Jest to sposób najbardziej naturalny dla operacji wstawiania i wyszukiwania. W ten sposób zawsze bez oczekiwania może wykonać się transakcja, która posiada węzeł znajdujący się najgłębiej i wysunięty najbardziej na prawo. Nieco bardziej skomplikowana jest sytuacja podczas usuwania wpisu. Gdy okazuje się, że węzeł w ma zbyt mało wpisów, wówczas następuje próba połączenia lub zrównoważenia z bratem z prawej strony (zgodnie ze schematem w dół i w prawo). Czasami jednak w jest skrajnie prawym synem i nie ma brata z prawej strony. Nie można wówczas czekać na odblokowanie brata z lewej strony, gdyż może to być liść zablokowany przez inną transakcję, oczekującą na odblokowanie kolejnego liścia, czyli właśnie w. Następuje więc próba nieblokującego zajęcia lewego brata w. Jeśli jest on zablokowany przez jakąś transakcję, zajęcie nie powiedzie się. W tym wypadku w jest zwalniany, po czym następuje pobranie lewego brata, a następnie w (zgodnie z zasadą w dól i w prawo). Nie ma przy tym obawy, że w międzyczasie węzły zostaną usunięte lub zmieni się ich zawartość, gdyż cały czas transakcja posiada ojca w w trybie tylko do zapisu. A każda transakcja, która chciałaby pisać w węzłach, musiałaby przejść do nich od korzenia poprzez ojca. Zatem jeśli w nie jest liściem, to bez obawy o zakleszczenie można zająć także lewego brata w.

Podczas wstawiania, jeśli nie jest przeprowadzane scalanie lub równoważenie węzłów, to w każdej chwili dla danej transakcji zarezerwowane są co najwyżej dwa węzły w trybie do zapisu. Jeśli trzeba węzły scalić lub zrównoważyć, to zarezerwowane będą trzy węzły. Podczas wyszukiwania zawsze zajęte są co najwyżej dwa węzły tylko do odczytu. Jeden krok w dół drzewa dla algorytmu dodawania wygląda tak:

1. pobierz węzeł v (na początku korzeń) do zapisu
2. znajdź adres syna v, do którego należy zejść
3. pobierz właściwego syna v (nazwijmy go w). Jeśli w ma zbyt mało danych to:
   (a) jeśli w ma prawego brata, pobierz tego brata
   (b) jeśli w nie ma prawego brata i nie jest liściem, to pobierz lewego brata
   (c) jeśli w jest liściem, to zwolnij w, pobierz lewego brata w, a potem pobierz w

(d) wykonaj równoważenie lub scalanie w z pobranym bratem

4. zwolnij v

Wyszukiwanie zrealizowane jest analogicznie. Pomijany jest jedynie pkt 3. Po dojściu do liścia zmienia się tylko to, że zamiast pobierania syna, pobierany jest prawy sąsiad.

Powyższe rozważania byłyby kompletne, gdyby nie fakt, że transakcje oprócz oczekiwania na zwolnienie węzłów przez inne transakcje, muszą jeszcze oczekiwać na dostarczenie węzła z menadżera buforów (rozdz. 8). To nieco komplikuje sytuację. Jeśli bufor ma określoną pojemność, to w tym momencie również mogą się pojawić zakleszczenia. Może się zdarzyć, że jedyna transakcja, która mogłaby się wykonać bez oczekiwania (zgodnie z dotychczasowymi rozważaniami), poprosi o kolejny węzeł, ale bufor będzie pełny. Zatem jedyna transakcja, która miała szansę się wykonać, będzie czekać, aż inna transakcja zwolni jakiś węzeł. Wyobraźmy sobie następującą sytuację. Załóżmy, że rozmiar bufora wynosi n. Rozpoczyna się n transakcji i każda z nich korzysta z innego drzewa. Wszystkie pobierają korzeń i potem już żadna nie może pobrać kolejnego węzła. Jak więc poradzić sobie z ograniczonym buforem? Możliwych jest kilka rozwiązań.

- Zamiast nakładać ograniczenia na rozmiar bufora, można określić maksymalną dopuszczalną liczbę transakcji, które mogą w tym samym czasie korzystać z indeksów. Jednak w ten sposób nakładane jest pewne sztuczne ograniczenie. Przecież nic nie stoi na przeszkodzie, żeby na raz wykonywało się wiele transakcji odczytujących te same węzły.

- Można też wyliczać, która transakcja jest w drzewie na uprzywilejowanej pozycji (najniżej i najbardziej po prawej) i pozwolić jej powiększyć bufor, po czym nie puszczać kolejnych transakcji, aż nie powróci do podstawowego rozmiaru. Ten sposób opisuje "najczystsze" rozwiązanie - definiujemy rozmiar bufora i nigdy nie jest on przekraczany. Jednak jest to dosyć skomplikowane. Podczas każdego pobierania i zwalniania węzła należałoby sprawdzać, która transakcja to robi i jak się ma pozycja danego węzła do węzłów wczytanych przez inne transakcje. Co więcej należałoby rozpatrywać wszystkie drzewa znajdujące się w systemie, a nie tylko jedno. Znalezienie transakcji uprzywilejowanej jest trudne.

- Można też przyjąć stałe ograniczenie na rozmiar bufora, a po jego przekroczeniu pozwalać pracować wszystkim transakcjom, które już zaczęły korzystać z drzewa, bo ich zatrzymanie mogłoby spowodować zakleszczenie. Żadna nowa transakcja nie pobierze nowego węzła, póki rozmiar bufora nie spadnie do poziomu podstawowego. W prezentowanej implementacji wybrano właśnie to rozwiązanie jako kompromis pomiędzy ograniczaniem współbieżności, rozrostem bufora i skomplikowaniem implementacji.

Rozdział 10

Optymalizacja zapytań

Indeksów można używać w sposób przezroczysty lub nieprzezroczysty. Drugi sposób polega na jawnym wywoływaniu funkcji wyszukiwania w indeksie. Przezroczyste wykorzystanie indeksu obejmuje takie przepisywanie zwykłych zapytań, aby wtedy, gdy to jest opłacalne, odwoływały sie one do funkcji wyszukiwania w indeksie.

Ze względu na obszerność tematu [Ora, P, PS, RG, S], optymalizacje nie były główną częścią prezentowanej implementacji. Założeniem pracy było zrealizowanie prostych optymalizacji niewymagających użycia metamodelu ani statystyk. Zatem jakie (pod)zapytania chcemy zoptymalizować? Przede wszystkim te, które bezpośrednio implementuje indeks (rozdz. 7.2), czyli np.

- `emp where age = 20`
- `emp where age > 25`
- `emp where age > 25 and age <= 50`

Bardziej ogólnie, cytując za [P], zoptymalizować można podzapytanie postaci

$$\textit{obiekt}\ \texttt{where}\ (atr_1\ op_1\ val_1\ \texttt{and}\ \ldots\ \texttt{and}\ atr_n\ op_n\ val_n\ \texttt{and}\ p) \tag{10.1}$$

gdzie dla $i \in \{1, 2, \ldots, n\}$, atr_i jest atrybutem obiektu *obiekt*, na którym założony jest indeks, op_i jest dwuargumentowym operatorem relacji (np. $=$, $<$, $\geqslant$), val_i jest zapytaniem zwracającym pojedynczy obiekt atomowy, p jest predykatem, a nazwa "obiekt" jest związana z bazową sekcją stosu.

Warto zmienić tę postać na semantycznie równoważną jednak wygodniejszą do rozpatrywania pod kątem implementacji

$$\textit{obiekt}\ \texttt{where}\ (w_1\ \texttt{and}\ \ldots\ \texttt{and}\ w_n) \tag{10.2}$$

gdzie warunek w_i ma jedną z trzech postaci:

$$atr_i\ op_i\ val_i \text{ lub } val_i\ op_i\ atr_i \tag{10.3}$$

$$\text{lub } p_i \tag{10.4}$$

Taka postać pozwala rozpoznać zapytanie nadające się do optymalizacji bez względu na kolejność poszczególnych podzapytań. Przykładem takiego zapytania jest

```
emp where 20 < age and (sal < 1000 or sal > 2000) and age < 30
```

Wybierając warunki, których użyjemy do wyszukiwania po indeksie, podzapytanie (10.2) możemy przekształcić do postaci semantycznie równoważnej

$$\begin{array}{r} indexCall(obiekt, w_k)\ \texttt{where} \\ (w_1\ \texttt{and}\ \ldots\ \texttt{and}\ w_{k-1}\ \texttt{and}\ w_{k+1}\ \texttt{and}\ldots\ \texttt{and}\ w_n) \end{array} \tag{10.5}$$

albo

$$\begin{array}{r} indexCall(obiekt, w_k, w_l)\ \texttt{where} \\ (w_1\ \texttt{and}\ \ldots\ \texttt{and}\ w_{k-1}\ \texttt{and}\ w_{k+1}\ \texttt{and}\ldots\ w_{l-1}\ \texttt{and}\ w_{l+1}\ \texttt{and}\ldots\ \texttt{and}\ w_n) \end{array} \tag{10.6}$$

W drugim przypadku op_k jest operatorem ograniczającym z jednej strony (np. $\leqslant$), a op_j z drugiej (np. $>$). Poprzez *indexCall* oznaczamy wywołanie indeksu na danych korzeniach z odpowiednio jednym lub dwoma ograniczeniami. Aby taka optymalizacja była możliwa konieczne jest, żeby warunki w_k, w_l były postaci (10.3), a podzapytania val_k i val_l były niezależne od operatora `where`, czyli nie wiązały się z sekcją stosu utworzoną przez *obiekt* [S].

Ze względu na niekorzystanie ze statycznej analizy, w prezentowanej implementacji musiały zostać przyjęte pewne uproszczenia. Załóżmy, że mamy indeks na korzeniach r po polu s. Rozważmy zapytanie

$$q.\underbrace{(r\ \texttt{where}\ s\ =\ 5)}_{\sigma} \tag{10.7}$$

Czy można je zoptymalizować? Bez znajomości sekcji wiązań, nie. Jeśli r lub s zostanie związane z wnętrzem q, wówczas po zastąpieniu podzapytania σ wywołaniem indeksu, otrzymane zapytanie nie będzie semantycznie równoważne (10.7). Z tego powodu zostało przyjęte następujące ograniczenie. Zapytanie (10.2) nie może być podzapytaniem. W ten sposób będziemy mieć pewność, że zapytanie dotyczy korzeni i ich podobiektów.

Podobny problem jest z podzapytaniami val_k i val_l występującymi w zapytaniach (10.5) i (10.6). Bez metamodelu nie ma możliwości sprawdzenia czy któreś z nich nie wiąże się z sekcją utworzoną przez włożenie na stos obiektu *obiekt*. Mając przykładowe zapytanie

$$emp\ \texttt{where}\ salary > age * 100 \tag{10.8}$$

nie wiemy, z którą sekcją zwiąże się nazwa *age*, nie można więc użyć indeksu. Z tego powodu przyjęte zostało kolejne ograniczenie. Podzapytania val_k i val_l muszą być literałami.

Podsumowując, w prezentowanej implementacji zapytanie zostanie zoptymalizowane, jeśli będzie postaci (10.2), podzapytanie val_i będzie literałem, a $op_i \in \{=, <, >, \leqslant, \geqslant\}$.

Może się zdarzyć, że wiele warunków w_i będzie spełniać kryteria wyboru do wyszukiwania po indeksie. Jak dokonać optymalnego wyboru? Należy użyć tych najbardziej selektywnych. Nie mając jednak żadnych informacji statystycznych o selektywności poszczególnych pól, nie można stwierdzić, które z warunków najlepiej wybrać. W prezentowanej implementacji zastosowano bardzo prostą strategię wyboru (jednak zostało to zrealizowane tak, aby w przyszłości łatwo można było wykorzystać dodatkowe informacje o selektywności). Przyjęto, że operator równości jest bardziej selektywny niż obustronne ograniczenie zakresowe, a ono z kolei jest bardziej selektywne niż jednostronne ograniczenie zakresowe. Jeżeli kilka różnych indeksów oferuje taką samą selektywność, wówczas wybierany jest dowolny z nich.

Jak zostało to zaimplementowane? Optymalizacja przeprowadzana jest na drzewie składniowym, już po ewentualnych innych przekształceniach wykonywanych przez parser. Najpierw sprawdzane jest czy zapytanie jest postaci (10.2). Następnie dla każdego pola atr_i, dla

którego val_i jest literałem, tworzony jest obiekt klasy `OptimizedField`. Obiekt ten otrzymuje informacje o ograniczeniach jakie są związane z danym polem. Gdy napotkany zostanie kolejny warunek dotyczący tego samego pola, odnajdywany jest właściwy obiekt `OptimizedField` i dodawane jest do niego kolejne ograniczenie. Jeśli dodawanych jest wiele ograniczeń, obiekt ten wybiera te najlepsze (wg. przedstawionej wcześniej strategii. Właśnie tutaj jest miejsce na przyszłe dodanie informacji o selektywności pola). Na podstawie nałożonych ograniczeń taki obiekt potrafi określić swoją selektywność. Po przeanalizowaniu całego zapytania wybierane jest najbardziej selektywne `OptimizedField`. Taki obiekt pamięta, które warunki (nazwijmy je W) zostaną zastąpione wywołaniem indeksu. Jeśli oprócz W w drzewie są jeszcze jakieś inne warunki, to `OptimizedField` potrafi usunąć W z tego drzewa. Obiekt ten potrafi też zwrócić węzeł reprezentujący wywołanie indeksu. Potem, jeśli pozostały jeszcze jakieś warunki, wystarczy już tylko skleić je z węzłem reprezentującym wywołanie indeksu, za pomocą operatora `where`.

Rozdział 11

Podsumowanie

Praca była pierwszą próbą włączenia indeksów do systemu LoXiM. Udało się zrealizować wszystkie założenia, choć przyjętych zostało kilka uproszczeń związanych z półstrukturalnym modelem danych. Przetwarzanie stosowe nie jest na razie stosowane w jakimkolwiek komercyjnym produkcie, bazy półstrukturalne też nie są jeszcze popularne. Efektywne optymalizacje (w tym indeksy) są konieczne, aby baza danych mogła stać się czymś więcej niż tylko projektem badawczym. Mam więc nadzieję, że ta praca dobrze wskazuje pewne problemy, które muszą być wzięte pod uwagę podczas projektowania takich systemów w przyszłości.

Najważniejsze z dalszych kierunków rozwoju to:

- rozszerzenie możliwości indeksowania także na obiekty nie będące korzeniami (indeksy ścieżkowe).
- rozluźnienie wymogów co do struktury indeksowanych obiektów (rozdz. 2).
- wykorzystanie metamodelu do zniesienia ograniczeń nałożonych na optymalizowane zapytanie (rozdz. 10).
- implementacja kolejnych rodzajów optymalizacji wykorzystujących indeksy (złączenia, wyszukiwanie *like*, *min* itp.).

Dodatek A

Testy jednostkowe

Testy jednostkowe przeprowadzane były przy pomocy programu *check* w wersji 0.93 dostępnego na licencji GNU LGPL ze strony projektu: `http://check.sourceforge.net`.

Środowisko testowe budowane jest analogicznie jak podczas uruchamiania systemu. Jednak część odpowiedzialna za nasłuchiwanie połączeń od klientów, została zmieniona tak, aby możliwa była symulacja tych połączeń bez użycia gniazd. Dodatkowo, w środowisku testowym, wszystkie te symulowane połączenia wraz z przydzielonymi im aparatami wykonawczymi pracują w jednym wątku. Ułatwia to debugowanie jak również testowanie przeplotów oraz poprawności wewnętrznego stanu aplikacji (w dowolnym momencie pracy systemu), a nie tylko wyników zwracanych na konsolę użytkownika. To z kolei sprawia, że testy mogą być mniejsze i bardziej precyzyjne, a co za tym idzie znacznie prostsze do napisania. Na potrzeby testów ustawiany był bardzo mały rozmiar węzłów (tak, aby jak najczęściej następowało ich rozbijanie, scalanie i wymiana w buforze), dzięki czemu testy pokrywały większą część kodu. Aby przeprowadzić część testów integracyjnych, wiele modułów LoXiMa musiało zostać nieznacznie zmienionych, gdyż nie zostały one napisane z myślą o wielokrotnym uruchamianiu w obrębie jednego wykonania programu (inicjacja zmiennych statycznych).

Źródła odpowiedzialne za testowanie znajdują się w katalogu `Indexes/Tests`

- `Tester.cpp, .h` - pomocnicza klasa umożliwiająca dostęp do prywatnych danych, które będą testowane.
- `ConnectionThread.cpp, .h` - klasa symulująca jedno połączenie z serwerem. Umożliwia wykonywanie poleceń analogicznie jak z konsoli.
- `testRun.cpp, Suits.h` - nagłówki testów oraz plik uruchamiający wszystkie testy. Uruchomiony bez parametrów przeprowadza testy w osobnych procesach. Jeśli zostanie podany jakikolwiek parametr, testy uruchomią się w jednym wątku, ułatwiając debugowanie.
- `BTreeTest.cpp` - testy operacji na B^+drzewie. Tworzenie, wstawianie, wyszukiwanie.
- `CleanTest.cpp` - testy czyszczenia węzłów. Poprawne wykrywanie i usuwanie niepotrzebnych już wpisów.
- `ComparatorTest.cpp` - testy komparatorów (rozdz. 6). Ustawianie wartości odpowiedniego typu, wstawianie wartości do węzła.
- `DropNodes.cpp` - testy zarządzania węzłami usuniętymi. Utrzymywanie węzłów indeksu w listach, utrzymywanie listy węzłów usuniętych, usuwanie indeksu.

- `ImplicitCalling.cpp` - testy optymalizacji zapytań poprzez niejawne wywołania indeksów. Poprawna identyfikacja zapytań, które mogą zostać zoptymalizowane. Wybór właściwego wywołania indeksu.
- `IntegrationTest.cpp` - testy integracyjne. Uruchamianie i zamykanie systemu, symulowana praca z konsoli: tworzenie indeksu, wyszukiwanie przy użyciu indeksu, modyfikowanie obiektów indeksowanych. Transakcyjność: widoczność zmian i izolacja.
- `NodeBuffer.cpp` - testy menadżera buforów. Zrzucanie na dysk węzłów o odpowiednim priorytecie. Zwracanie węzłów z bufora i sprowadzanie z dysku.
- `ParentName.cpp` - testy usuwania obiektów z bazy. Wykrywanie czy obiekt jest indeksowany, czy poprawnie przechowuje odnośnik do swojego obiektu nadrzędnego i czy jego usunięcie jest legalne.
- `ScopeSelection.cpp` - testy klas `Constraints` (rozdz. 7.2). Wyszukiwania zakresowe dużej ilości wpisów.
- `StateSaving.cpp` - testy przechowywania danych pomiędzy zamknięciem i ponownym uruchomieniem serwera.
- `Syntax.cpp` - testy poprawności parsowania poleceń podawanych z konsoli, jawnie wywołujących wyszukiwanie za pomocą indeksu.
- `Bugs.cpp` - testy poprawek dla błędów znalezionych po scaleniu indeksów do głównej gałęzi projektu.
- `Makefile` - plik budujący programu *make*. Zawiera trzy główne reguły: all - buduje wszystkie testy, wymaga skompilowanego LoXiMa; run - uruchamia testy w osobnych procesach; clean - usuwa skompilowane pliki, pozostawia tylko źródła.

Dodatek B

Składnia

Założenie indeksu na zadanych korzeniach po zadanym polu:

`create` [`int`|`double`|`string`] `index` *nazwa_indeksu* `on` *nazwa_korzenia* `(` *nazwa_pola* `)`

przykłady:

```
create int index emp_age on emp(age)
create double index cash on client(price)
create string index names on emp(name)
```

Usunięcie indeksu:

`delete index` *nazwa_indeksu*

przykład:

```
delete index emp_age
```

Wypisanie wszystkich istniejących indeksów:

```
index
```

Poniżej przedstawiona jest składnia dla jawnego wywołania indeksów. Polecenia te zawierają w sobie inne zapytania, które są wyliczane zanim zostanie użyty indeks. Rezultat każdego z nich musi być pojedynczą wartością lub jedynym elementem dowolnej kolekcji. Ponadto wartość ta musi mieć typ obsługiwany przez wywoływany indeks. W poleceniach wyszukiwania zakresowego występują też cztery rodzaje ograniczeń ustalających obsługę wartości brzegowych. Ograniczenia `<|` oraz `|>` oznaczają, że wartość ma znaleźć się w zbiorze wynikowym, a `(|` oraz `|)`, że nie. W przypadku każdego z wywołań indeksu, zwrócony zostanie wielozbiór zawierający logiczne identyfikatory korzeni, których zaindeksowane pole ma zadaną wartość (w przypadku zapytań równościowych) lub należy do zadanego przedziału (w przypadku zapytań zakresowych). Zatem wywołań indeksu można używać jako elementów składowych większych zapytań.

Wyszukiwanie równościowe:

`index` *nazwa_indeksu* `=` *zapytanie*

przykłady:

```
index emp_age = 25                          wyszukanie wartości 25
index emp_age = max(dept.emp.age)           wyszukanie wartości max(dept.emp.age)
```

Wyszukiwanie zakresowe "większy od":

`index` *nazwa_indeksu* [`(|`|`<|`] *zapytanie* `|`

przykłady:

`index emp_age (| 20 |` przedział $(20; \infty)$

`index emp_age (| avg(dept.emp.age) |` przedział $(avg(dept.emp.age); \infty)$

`index emp_age <| 20 |` przedział $\langle 20; \infty)$

Wyszukiwanie zakresowe "mniejszy od":

`index` *nazwa_indeksu* `|` *zapytanie* [`|)||>`]

przykłady:

`index emp_age | 20 |)` przedział $(-\infty; 20)$

`index emp_age | avg(dept.emp.age) |)` przedział $(-\infty; avg(dept.emp.age))$

`index emp_age | 20 |>` przedział $(-\infty; 20\rangle$

Wyszukiwanie zakresowe "od do":

`index` *nazwa_indeksu* [`(||<|`] *zapytanie_a* `to` *zapytanie_b* [`|)||>`]

przykłady:

`index emp_age <| 20 to 40 |)` przedział $\langle 20; 40)$

`index emp_age (|max(sal)+2 to 30|>` przedział $(max(sal) + 2; 30\rangle$

Dodatek C

Scenariusz testowy

Aby skorzystać z niejawnych wywołań indeksów, w pliku konfiguracyjnym musi zostać włączona opcja *implicit_index_call*. Po każdym poleceniu podanym z konsoli klienta, w dzienniku serwera pojawia się informacja mówiąca czy zapytanie zostało niejawnie przepisane tak, aby wywoływało indeksy. Jeśli nie, pojawia się komunikat `IndexManager: no implicit index optimisation`. Jeśli jednak możliwa była optymalizacja, w dzienniku znajdzie się `IndexManager: index implicitly used:` oraz postać nowego zapytania. Poniżej przestawiono scenariusz pozwalający ręcznie przetestować te optymalizacje.

zapytanie: `emp where age > 20 and age <= 40`
odpowiedź: `IndexManager: no implicit index optimisation`

Nie ma jeszcze stworzonego żadnego indeksu, więc żadna optymalizacja nie zostanie wykonana. Stwórzmy zatem indeks.

```
create int index emp_age on emp(age)
```

Zakładanie indeksu nie podlega transakcjom, więc indeks zostanie założony i pozostanie w systemie aż do jego usunięcia, niezależnie czy jesteśmy wewnątrz transakcji, czy nie. Ponownie wprowadźmy pierwsze zapytanie.

zapytanie: `emp where age > 20 and age <= 40`
odpowiedź: `index emp_age ( 20  to  40 >`

Oznacza to, że zapytanie zostało przekształcone. Wykorzystany będzie indeks *emp_age*, użyte zostanie wyszukiwanie zakresowe, a konkretnie przedział $(20; 40\rangle$.

zapytanie: `emp where age = 30`
odpowiedź: `index emp_age = 30`

Oznacza to, że wykorzystane będzie wyszukiwanie równościowe.

zapytanie: `emp where age > 30`
odpowiedź: `index emp_age ( 30  to +inf)`

Oznacza wyszukiwanie zakresowe, przedział $(30; \infty\rangle$.

zapytanie: `emp where name = "kowalski" and age > 30`
odpowiedź: `index emp_age ( 30  to +inf) where ( name = "kowalski" )`

Oznacza, że wykonane zostanie wyszukiwanie zakresowe, a następnie z wyniku zostaną wybrane obiekty spełniające predykat `name = "kowalski"`.

Bibliografia

[BMS] Lech Banachowski, Elżbieta Mrówka-Matejewska, Krzysztof Stencel, *Systemy baz danych. Wykłady i ćwiczenia*, Wydawnictwo PJWSTK, 2004

[CLQS] Raul F. Chong, Clara Liu, Sylvia F. Qi, Dwaine R. Snow, *Zrozumieć DB2. Nauka na przykładach*, Wydawnictwo Naukowe PWN, 2006

[CLR] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo-Techniczne, 1997

[G] Joe Greene, et al., *Oracle 8 Server. Księga eksperta*, Helion, 2000

[GGT] Natasha Gelfand, Michael T. Goodrich, Roberto Tamassia, *Teaching Data Structure Design Patterns*, ACM SIGCSE Bulletin, v.30 n.1, p.331-335, Mar. 1998

[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, *Design Patterns. Elements of Reusable Object Oriented Software*, Addison-Wesley Professional, 1995

[GUW] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, *Systemy baz danych. Pełny wykład*, Wydawnictwa Naukowo-Techniczne, 2006

[GUW2] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, *Implementacja systemów baz danych*, Wydawnictwa Naukowo-Techniczne, 2003

[K] Donald E. Knuth, *Sztuka programowania. Tom III, Sortowanie i wyszukiwanie*, Wydawnictwa Naukowo-Techniczne, 2002

[Ora] Oracle, *Oracle Database Performance Tuning Guide, 10g Release 1 (10.1)*, http://download.oracle.com/docs/cd/B14117_01/server.101/b10752/toc.htm

[P] Jacek Płodzień, *Optimization Methods in Object Query Languages* PhD Thesis, IPI PAN, 2000

[PS] Jacek Płodzień, Kazimierz Subieta *Kompilacja i optymalizacja obiektowych zapytań*, IPI PAN, 1998

[RG] Raghu Ramakrishnan, Johannes Gehrke *Databse Management Systems. Third Edition*, McGraw-Hill, 2002

[S] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo PJWSTK, 2004

[SQLite] SQLite, *SQLite 2.X Database File Format*, http://www.sqlite.org/fileformat.html