KANON INFORMATYKI

Systemy baz danych

Kompletny podręcznik

PIERWSZE TAK SZCZEGÓŁOWE, KOMPLEKSOWE WPROWADZENIE! TRZEJ ZNANI NAUKOWCY Z DZIEDZINY IT NAUCZĄ CIĘ:

- profesjonalnego projektowania i korzystania z baz danych
- tworzenia i wdrażania złożonych aplikacji bazodanowych
- sprawnej implementacji systemów zarządzania bazami danych

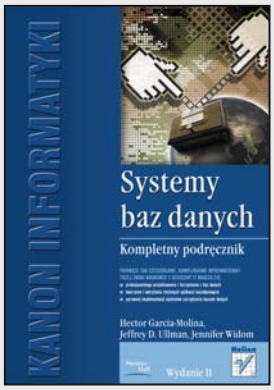
Hector Garcia-Molina,
Jeffrey D. Ullman, Jennifer Widom

Prentice Hall

Wydanie II

Helion

Systemy baz danych. Kompletny podręcznik. Wydanie II



Autorzy: Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom
Tłumaczenie: Tomasz Walczak
ISBN: 978-83-246-3303-6
Tytuł oryginału: Database Systems: The Complete Book (2nd Edition)
Format: 172×245, stron: 1048

Pierwsze tak szczegółowe, kompleksowe wprowadzenie!

Trzej znani naukowcy z dziedziny IT nauczą Cię:

- profesjonalnego projektowania i korzystania z baz danych
- tworzenia i wdrażania złożonych aplikacji bazodanowych
- sprawnej implementacji systemów zarządzania bazami danych

Z kluczowej roli, jaką bazy danych odgrywają w codziennym życiu milionów ludzi, zdajemy sobie sprawę za każdym razem, gdy wpisujemy hasło w wyszukiwarce Google, robimy zakupy w internetowej księgarni czy logujemy się do swojego konta w banku. Szybkie, bezpieczne i niezawodne przetwarzanie oraz przechowywanie ogromnych ilości informacji stało się dziś strategicznym czynnikiem funkcjonowania większości firm, organizacji i instytucji państwowych. Ten ogromny potencjał współczesnych baz danych jest dziś sumą wiedzy i technologii rozwijanych przez kilka ostatnich dziesięcioleci. Owocem tych prac jest przede wszystkim wyspecjalizowane oprogramowanie – systemy zarządzania bazami danych DBMS, czyli rozbudowane narzędzia do wydajnego tworzenia dużych zbiorów informacji i zarządzania nimi. Niestety, mają one jedną zasadniczą wadę – należą do najbardziej złożonych rodzajów oprogramowania!

W związku z tym trzech znanych naukowców w dziedzinie IT z Uniwersytetu Stanforda. Hector Garcia-Molina, Jeffrey D. Ullman i Jennifer Widom – postanowiło stworzyć pierwszy tak kompletny podręcznik, wprowadzający do systemów baz danych. Zawiera on opis najnowszych standardów bazy danych SQL 1999, SQL/PSM, SQL/CLI, JDBC, ODL oraz XML – i to w znacznie szerszym zakresie niż w większości publikacji. Podręcznik został przygotowany w taki sposób, aby po jego przeczytaniu użytkowanie czy projektowanie baz danych, pisanie programów w różnych językach związanych z systemami DBMS oraz ich sprawna implementacja nie stanowiły dla Czytelnika najmniejszego problemu!

Kompleksowe podejście do systemów baz danych z punktu widzenia projektanta, użytkownika i programisty

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

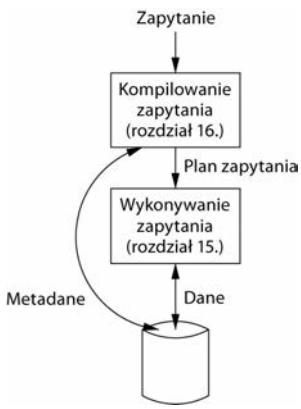
Spis treści

15

Wykonywanie zapytań

Obszerne zagadnienie przetwarzania zapytań opisano w tym i następnym (16.) rozdziale. *Procesor zapytań* to grupa komponentów systemu DBMS, które przekształcają zapytania użytkownika i polecenia modyfikacji danych na ciąg operacji na bazie danych oraz wykonują je. Ponieważ SQL umożliwia wyrażanie zapytań na bardzo wysokim poziomie, procesor zapytań musi obsługiwać wiele szczegółów dotyczących wykonywania zapytań. Ponadto naiwna strategia wykonywania zapytań może spowodować nieoptymalne wykorzystanie czasu.

Na rysunku 15.1 przedstawiono podział zagadnień z rozdziałów 15. i 16. Ten rozdział dotyczy głównie wykonywania zapytań — algorytmów manipulujących danymi z bazy. Skoncentrowano się na operacjach rozszerzonej algebry relacji opisanych w podrozdziale 5.2. Ponieważ SQL działa zgodnie z modelem opartym na wielozbiorach, założono, że relacje to wielozbiory, dlatego użyto przeznaczonych dla nich operatorów z podrozdziału 5.1.



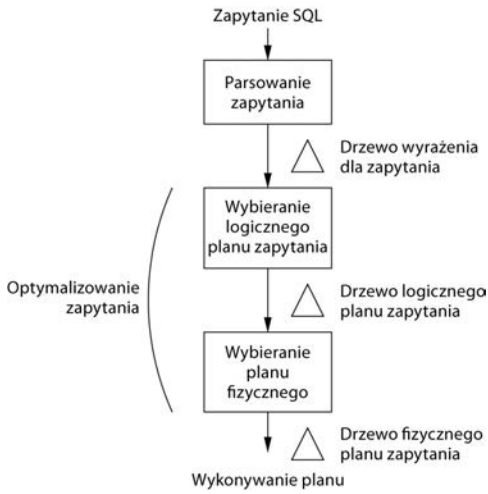
15.1. *Główne elementy procesora zapytań*

Omówiono podstawowe metody wykonywania operacji algebry relacji. Metody te różnią się podstawową strategią. Główne podejścia to skanowanie, haszowanie, sortowanie i indeksowanie. Techniki różnią się też założeniami dotyczącymi ilości dostępnej pamięci głównej. Niektóre algorytmy wymagają, aby pamięć główna umożliwiała zapisanie przynajmniej jednej z relacji uwzględnianych w operacji. W innych przyjęto, że argumenty operacji są za duże, aby mieściły się w pamięci. Poszczególne algorytmy różnią się kosztem i używanymi strukturami.

Wstęp do kompilowania zapytań

Aby wykonywaniu zapytania nadać kontekst, przedstawiono bardzo krótki zarys następnego rozdziału. Kompilacja zapytania dzieli się na trzy główne kroki pokazane na rysunku 15.2.

a) *Parsowanie*. Tworzone jest *drzewo wyprowadzenia* dla zapytania.
b) *Przepisywanie zapytania*. Drzewo wyprowadzenia jest przekształcane na wyjściowy plan zapytania, który zwykle stanowi algebraiczną reprezentację zapytania. Wyjściowy plan jest następnie przekształcany na równoważny, którego wykonanie powinno zająć mniej czasu.
c) *Generowanie planu fizycznego*. Abstrakcyjny plan zapytania z punktu b), często nazywany *logicznym planem zapytania*, jest przekształcany na *fizyczny plan zapytania* przez wybór algorytmów będących implementacją operatorów z planu logicznego, a także przez wybór kolejności stosowania tych operatorów. Plan fizyczny, podobnie jak efekt parsowania i plan logiczny, jest reprezentowany jako drzewo wyrażenia. Plan fizyczny obejmuje też takie szczegóły jak sposób dostępu do relacji z zapytania, a także informacje o tym kiedy (i czy w ogóle) relację należy sortować.



15.2. *Zarys procesu kompilowania zapytań*

Kroki b) i c) związane są z *optymalizatorem zapytań* i stanowią trudne aspekty kompilowania zapytania. Aby wybrać najlepszy plan zapytania, trzeba ustalić pewne rzeczy.

(1) Która z algebraicznie równoważnych postaci zapytania prowadzi do powstania najwydajniejszego algorytmu ustalania odpowiedzi?
(2) Których algorytmów należy użyć jako implementacji każdej operacji z wybranej postaci zapytania?
(3) Jak operacje powinny przekazywać dane między sobą — potokowo, przez bufory pamięci głównej czy przez dysk?

Każdy wybór oparty jest na metadanych dotyczących bazy. Typowe metadane dostępne dla optymalizatora zapytań obejmują rozmiar każdej relacji, statystyki w rodzaju przybliżonej liczby i częstotliwości występowania różnych wartości atrybutu, informacje o istnieniu określonych indeksów i informacje o układzie danych na dysku.

15.1. Wprowadzenie do operatorów z fizycznego planu zapytania

Fizyczne plany zapytania bazują na operatorach, z których każdy stanowi implementację jednego kroku planu. Operatory fizyczne zwykle są konkretnymi implementacjami jednej z operacji algebry relacji. Potrzebne są też operatory fizyczne do wykonywania zadań, które nie obejmują operacji algebry relacji. Przykładowo często trzeba „przeskanować" tabelę, czyli przenieść do pamięci głównej każdą krotkę relacji. Relacja jest zwykle operandem pewnej innej operacji.

W tym podrozdziale wprowadzono podstawowe cegiełki fizycznych planów zapytań. W dalszych podrozdziałach omówiono bardziej skomplikowane algorytmy, które są wydajną implementacją operatorów algebry relacji. Algorytmy te stanowią kluczową część fizycznych planów zapytań. Wprowadzono też iteratory — jest to ważne narzędzie, które operatorom z fizycznego planu zapytania umożliwia przekazywanie między sobą żądań krotek i odpowiedzi.

15.1.1. Skanowanie tabel

Prawdopodobnie najbardziej podstawowym zadaniem w fizycznym planie zapytania jest wczytanie całej zawartości relacji R. Odmiana tej operacji obejmuje prosty predykat, co umożliwia wczytanie tylko tych krotek relacji R, które spełniają predykat. Są dwa podstawowe podejścia do znajdowania krotek relacji R.

(1) Często relacja R jest przechowywana w pamięci drugiego stopnia, a krotki są uporządkowane w blokach. System „wie", w których blokach znajdują się krotki relacji R, i może pobierać bloki jeden po drugim. Operacja ta to *skanowanie tabeli*.

(2) Jeśli istnieje indeks na dowolnym atrybucie relacji R, można go użyć, aby uzyskać dostęp do wszystkich krotek z R. Przykładowo indeks rzadki relacji R można, co opisano w punkcie 14.1.3, wykorzystać do przejścia do wszystkich bloków przechowujących R, nawet jeśli nie wiadomo, które to bloki. Ta operacja to *skanowanie indeksu*.

Skanowanie indeksu opisano ponownie w punkcie 15.6.2, w ramach omówienia implementowania selekcji. Na razie warto zaobserwować, że indeks można wykorzystać nie tylko do pobrania *wszystkich* krotek indeksowanej relacji, ale też krotek o określonej wartości (lub przedziału wartości) atrybutu lub atrybutów stanowiących klucz wyszukiwania dla indeksu.

15.1.2. Sortowanie w trakcie skanowania tabel

Jest wiele powodów, dla których przydatne jest sortowanie relacji w czasie wczytywania krotek. Zapytanie może obejmować klauzulę `ORDER BY`, wymagającą posortowania relacji. Ponadto niektóre techniki implementowania operacji algebry relacji wymagają, aby jeden lub oba argumenty były posortowanymi relacjami. Algorytmy te występują w podrozdziale 15.4 i w innych miejscach.

Operator *sortowania w czasie skanowania* w fizycznym planie zapytania przyjmuje relację R oraz wykaz atrybutów, według których należy posortować dane, i zwraca posortowaną relację R. Sortowanie w czasie skanowania można zaimplementować na kilka sposobów. Jeśli relację R trzeba posortować według atrybutu a i istnieje indeks w postaci drzewa zbalansowanego na a, skanowanie indeksu umożliwia uporządkowanie R w pożądanej kolejności. Jeżeli relacja R jest na tyle mała, że mieści się w pamięci głównej, można pobrać krotki przez skanowanie tabeli lub indeksu, a następnie wykorzystać algorytm sortowania w pamięci głównej. Jeśli R jest zbyt duża, aby zapisać ją w pamięci głównej, można wykorzystać wielościeżkowe sortowanie przez scalanie, co omówiono w punkcie 15.4.1.

15.1.3. Model obliczeń dla operatorów fizycznych

Zwykle zapytanie obejmuje kilka operacji algebry relacji, a powiązany fizyczny plan zapytania zawiera kilka operatorów fizycznych. Ponieważ rozsądny wybór operatorów do planu fizycznego jest w procesorze zapytań kluczowy, możliwe musi być oszacowanie kosztu każdego używanego operatora. Jako miary kosztu używamy tu liczby dyskowych operacji wejścia-wyjścia. Miara ta jest spójna z założeniem (punkt 13.3.1), że dłużej trwa pobieranie danych z dysku niż wykonywanie przydatnych operacji na nich po przeniesieniu ich do pamięci głównej.

Przy porównywaniu algorytmów dla tych samych operacji obowiązuje założenie, które początkowo może wydawać się zaskakujące.

- Przyjmujemy, że argumenty operatora znajdują się na dysku, natomiast wynik zostaje zapisany pamięci głównej.

Jeśli operator tworzy ostateczną odpowiedź na zapytanie, a wynik jest zapisywany na dysku, koszt wykonania tej operacji zależy tylko od wielkości odpowiedzi, a nie od sposobu jej obliczenia. Wystarczy dodać koszt ostatecznego zapisu do łącznego kosztu zapytania. Jednak w wielu zastosowaniach odpowiedź w ogóle nie jest zapisywana na dysku, tylko wyświetlana lub przekazywana do programu formatującego. Koszt dyskowych operacji wejścia-wyjścia wynosi wtedy zero lub zależy od tego, co pewna nieznana aplikacja robi z danymi. W obu sytuacjach koszt zapisania odpowiedzi nie wpływa na wybór algorytmu wykonywania operacji.

Podobnie wynik operacji stanowiącej część zapytania (w odróżnieniu od całego zapytania) często nie jest zapisywany na dysku. W punkcie 15.1.6 omówiono iteratory. Działają one tak — wynik uruchomienia jednego operatora O_1 jest tworzony w pamięci głównej, często po małym fragmencie, i przekazywany jako argument do innego operatora O_2. Wtedy nigdy nie trzeba zapisywać wyniku działania operatora O_1 na dysku. Ponadto można uniknąć ponoszenia kosztu wczytywania z dysku argumentu operatora O_2.

15.1.4. Parametry do pomiaru kosztów

Wprowadźmy teraz parametry (nazywane czasem statystykami) służące do wyrażania kosztów operatorów. Oszacowanie kosztów jest niezbędne, jeśli optymalizator ma określać, który z wielu planów zapytania prawdopodobnie będzie działał najszybciej. W podrozdziale 16.5 pokazano, jak wykorzystać szacunki kosztów.

Potrzebny jest parametr reprezentujący część pamięci głównej wykorzystywaną przez operator, a także inne parametry określające rozmiar jego argumentów. Załóżmy, że pamięć główna jest podzielona na bufory o rozmiarze równym wielkości bloków dysku. M oznacza liczbę buforów w pamięci głównej dostępnych przy działaniu konkretnego operatora.

Czasem można traktować M jak całą pamięć główną lub większość tej pamięci. Zdarzają się jednak sytuacje, w których kilka operacji współużytkuje pamięć główną, dlatego M może być znacznie mniejsza niż łączna wielkość pamięci głównej. W praktyce, jak opisano to w podrozdziale 15.7, liczba buforów dostępnych operacji może nie być przewidywalną stałą. Nieraz jest ustalana w trakcie wykonywania programu na podstawie tego, jakie inne procesy działają w tym samym czasie. Wtedy M to szacunkowa liczba buforów dostępnych operacji.

Rozważmy teraz parametry do pomiaru kosztu dostępu do argumentów (czyli do relacji). Parametry te, mierzące rozmiar i rozkład danych w relacji, często są obliczane okresowo, aby optymalizatorowi zapytań ułatwić wybór operatorów fizycznych.

Przyjęto tu upraszczające założenie, zgodnie z którym dane są pobierane z dysku po jednym bloku. W praktyce czasem można zastosować jedną z technik omówionych w podrozdziale 13.3

do przyspieszenia pracy algorytmu, jeśli możliwe jest jednoczesne wczytanie wielu bloków z daną relacją, przylegających do siebie na ścieżce. Istnieją trzy rodziny parametrów — B, T i V.

- Przy opisywaniu rozmiaru relacji R najczęściej ważna jest liczba bloków potrzebnych do przechowywania wszystkich krotek relacji. Liczbę bloków oznacza się jako $B(R)$ lub — jeśli wiadomo, że chodzi o relację R — samo B. Zwykle zakłada się, że R jest *sklastrowana* (ang. *clustered*; jest zapisana w B lub około B bloków).
- Czasem trzeba ustalić liczbę krotek w R. Wartość tę oznacza się jako $T(R)$ lub — gdy wiadomo, że chodzi o R — samo T. Jeśli potrzebna jest liczba krotek relacji R mieszczących się w jednym bloku, można wykorzystać stosunek T/B.
- Czasem warto też określić liczbę różnych wartości pojawiających się w kolumnie relacji. Jeśli R to relacja, a jednym z jej atrybutów jest a, wyrażenie $V(R, a)$ oznacza liczbę różnych wartości kolumny a w R. Bardziej ogólnie — jeśli $[a_1, a_2, \ldots, a_n]$ to lista atrybutów, $V(R, [a_1, a_2, \ldots, a_n])$ to liczba różnych n krotek w kolumnach relacji R odpowiadających atrybutom $a_1, a_2, \ldots, a_n$. Formalnie można stwierdzić, że jest to liczba krotek w $\delta(\pi_{a1, a2, \ldots, an}(R))$.

15.1.5. Koszty operacji wejścia-wyjścia dla operatorów skanowania

W ramach prostego zastosowania przedstawionych parametrów można zapisać liczbę dyskowych operacji wejścia-wyjścia potrzebnych dla każdego z omówionych operatorów skanowania tabeli. Jeśli relacja R jest sklastrowana, liczba dyskowych operacji wejścia-wyjścia dla takich operatorów to w przybliżeniu B. Jeżeli R mieści się w pamięci głównej, można zaimplementować sortowanie w czasie skanowania, wczytując R do pamięci i wykonując sortowanie w pamięci, co także wymaga tylko B dyskowych operacji wejścia-wyjścia.

Jeśli jednak R nie jest sklastrowana, liczba potrzebnych dyskowych operacji wejścia-wyjścia jest zwykle dużo większa. Jeżeli R jest rozproszona między krotkami innych relacji, przeglądanie tabeli dla relacji R może wymagać wczytania tylu bloków, ile jest krotek w R (koszt operacji wejścia-wyjścia wynosi T). Podobnie przy sortowaniu relacji R mieszczącej się w pamięci potrzeba T dyskowych operacji wejścia-wyjścia do przeniesienia całej R do pamięci.

Na koniec rozważmy koszt skanowania indeksu. Ogólnie indeks na relacji R zajmuje znacznie mniej niż $B(R)$ bloków. Dlatego skanowanie całej R, na co potrzeba przynajmniej B dyskowych operacji wejścia-wyjścia, wymaga znacznie więcej operacji niż sprawdzenie całego indeksu. Dlatego, choć skanowanie indeksu wymaga sprawdzenia zarówno relacji, jak i indeksu, to:

- nadal można używać B (lub T) do szacowania kosztu dostępu do całych sklastrowanych (lub niesklastrowanych) relacji z wykorzystaniem indeksu.

Jeśli jednak potrzebna jest tylko część R, często można uniknąć sprawdzania całego indeksu i całej R. Analizę takich zastosowań indeksu odkładamy do punktu 15.6.2.

15.1.6. Iteratory do implementowania operatorów fizycznych

Wiele operatorów fizycznych można zaimplementować jako *iterator*. Iterator łączy trzy metody umożliwiające odbiorcy wyniku działania fizycznego operatora pobieranie danych krotka po krotce. Oto te metody.

(1) `Open()`. Metoda rozpoczyna proces pobierania krotek, jednak ich nie wczytuje. Inicjuje struktury danych potrzebne do wykonania operacji i wywołuje metodę `Open()` dla jej argumentów.

(2) `GetNext()`. Zwraca następną krotkę z wyniku i — jeśli trzeba — dostosowuje struktury danych, aby umożliwić pobranie kolejnych krotek. Przy pobieraniu następnej krotki z wyniku zwykle wywołuje raz lub kilka razy metodę `GetNext()` dla argumentów. Jeżeli nie można zwrócić dalszych krotek, `GetNext()` zwraca specjalną wartość `NotFound` (zakładamy, że nie można jej pomylić z krotką).

(3) `Close()`. Ta metoda kończy iterowanie po pobraniu wszystkich krotek (lub krotek żądanych przez odbiorcę). Zwykle wywołuje metodę `Close()` dla argumentów operatora.

W opisie iteratorów i ich metod zakładamy, że dla każdego typu iteratora (na przykład dla każdego typu operatora fizycznego zaimplementowanego jako iterator) istnieje klasa, w której zdefiniowano metody `Open()`, `GetNext()` i `Close()` działające na jej egzemplarzach.

Przykład 15.1. Prawdopodobnie najprostszy jest iterator będący implementacją operatora skanowania tabeli. Iterator jest zaimplementowany jako klasa `TableScan`, a operator skanowania tabeli w planie zapytania to egzemplarz klasy mający parametr w postaci skanowanej relacji R. Załóżmy, że R to relacja sklastrowana w liście bloków, do której można w wygodny sposób uzyskać dostęp (przyjmujemy, iż operacja „pobierz następny blok R" jest zaimplementowana w systemie pamięci i nie trzeba jej szczegółowo opisywać). Ponadto zakładamy, że w bloku znajduje się katalog rekordów (krotek), dlatego można łatwo pobrać następną krotkę z bloku lub stwierdzić, iż osiągnięto ostatnią krotkę.

Na listingu 15.3 znajduje się zarys trzech metod iteratora. Zakładamy, że istnieje wskaźnik b do bloku i wskaźnik t do krotki w bloku wskazywanym przez b. Oba wskaźniki mogą prowadzić „poza" ostatni blok lub ostatnią krotkę w bloku. Można wykryć wystąpienie tych warunków. Zauważmy, że metoda `Close()` w przykładzie nie wykonuje żadnych działań. W praktyce może ona na różne sposoby porządkować wewnętrzną strukturę systemu DBMS. Może informować menedżer bufora, że niektóre bufory nie są już potrzebne, lub menedżer współbieżności o zakończeniu wczytywania relacji. ■

```
Open() {
   b := pierwszy blok R;
   t := pierwsza krotka bloku b;
}

GetNext() {
   IF (t prowadzi poza ostatnią krotkę bloku b) {
      zwiększ b, aby prowadził do następnego bloku;
      IF (nie ma następnego bloku)
         RETURN NotFound;
      ELSE /* b to nowy blok. */
         t := pierwsza krotka bloku b;
   } /* Teraz można zwrócić i zwiększyć t. */
   oldt := t;
   zwiększanie t, aby prowadził do następnej krotki b;
   RETURN oldt;
}

Close() {
}
```

15.3. *Metody iteratora dla operatora skanowania tabeli dla relacji R*

Przykład 15.2. Teraz rozważmy przykład, w którym iterator wykonuje większość zadań w metodzie `Open()`. Używamy operatora sortowania przy skanowaniu. Wczytuje on krotki relacji R i zwraca je w posortowanej kolejności. Do czasu sprawdzenia wszystkich krotek w R nie można zwrócić nawet pierwszej krotki. Dla uproszczenia załóżmy, iż relacja R jest na tyle mała, że mieści się w pamięci głównej.

Metoda `Open()` musi wczytywać całą relację R do pamięci głównej. Może też sortować krotki z R. Wtedy `GetNext()` musi tylko zwrócić po kolei każdą krotkę w posortowanej kolejności. Metoda `Open()` może też pozostawić R nieposortowaną. Wtedy w `GetNext()` należy pobierać pierwszą z pozostałych krotek, co w efekcie oznacza wykonanie jednego przebiegu sortowania przez wybieranie. ■

Przykład 15.3. Na koniec rozważmy prosty przykład łączenia iteratorów przez wywoływanie innych iteratorów. Operacja to suma wielozbiorów $R \cup S$, w której najpierw należy zwrócić wszystkie krotki R, a następnie wszystkie krotki S bez eliminowania powtórzeń. Niech $\boldsymbol{R}$ i $\boldsymbol{S}$ oznaczają iteratory zwracające relacje R i S (tym samym są dziećmi operatora sumy w planie zapytania dla operacji $R \cup S$). Iteratory $\boldsymbol{R}$ i $\boldsymbol{S}$ mogą odpowiadać skanowaniu tabeli relacji składowanych R i S. Ponadto mogą być iteratorami, które wywołują zestaw innych iteratorów w celu obliczenia R i S. Niezależnie od tego ważne jest, aby dostępne były metody `R.Open()`, `R.GetNext()` i `R.Close()` oraz odpowiadające im metody iteratora $\boldsymbol{S}$.

Zarys metod iteratora dla sumy przedstawiono na listingu 15.4. Ciekawe jest to, że metody korzystają ze zmiennej współużytkowanej `ObecRel` przechowującej R lub S (w zależności od tego, która relacja jest obecnie wczytywana). ■

```
Open() {
   R.Open();
   ObecRel := R;
}

GetNext() {
   IF (ObecRel = R) {
      t := R.GetNext();
      IF (t <> NotFound) /* R nie została pobrana w całości. */
         RETURN t;
      ELSE /* R została pobrana w całości. */ {
         S.Open();
         ObecRel := S;
      }
   }
   /* Tu trzeba wczytać dane z S. */
   RETURN S.GetNext();
   /* Zauważmy, że jeśli pobrano całą S, S.GetNext()
      zwróci NotFound, co jest prawidłowym działaniem
      metody GetNext. */
}

Close() {
   R.Close();
   S.Close();
}
```

15.4. *Tworzenie iteratorów dla sumy na podstawie iteratorów R i S*

Dlaczego iteratory?

W podrozdziale 16.7 pokazano, że połączenie iteratorów w planie zapytania pozwala uzyskać wysoką wydajność. Inaczej działa *materializacja*, kiedy to wynik działania każdego operatora jest zwracany w całości — zostaje zapisany albo na dysku, albo w pamięci głównej. Przy stosowaniu iteratorów jednocześnie aktywnych jest wiele operacji. Krotki są przekazywane między operatorami na żądanie, co zmniejsza zapotrzebowanie na pamięć. Oczywiście, co opisano dalej, nie wszystkie operatory fizyczne umożliwiają sensowną obsługę iteracji (potokowania). Czasem prawie wszystkie zadania trzeba wykonać w metodzie `Open()`, co przypomina materializację.

15.2. Algorytmy jednoprzebiegowe

W tym miejscu rozpoczyna się omówienie bardzo ważnego zagadnienia w obszarze optymalizowania zapytań: „Jak należy wykonywać każdy z poszczególnych kroków — na przykład złączenie lub selekcję — logicznego planu zapytania?". Wybór algorytmu dla każdego operatora to kluczowa część procesu przekształcania logicznego planu zapytania na plan fizyczny. Choć zaproponowano wiele algorytmów dla operatorów, należą one głównie do trzech klas.

(1) Metody oparte na sortowaniu (podrozdział 15.4).
(2) Metody oparte na haszowaniu (podrozdziały 15.5 i 20.1).
(3) Metody oparte na indeksach (podrozdział 15.6).

Ponadto algorytmy działania operatorów można podzielić na trzy „stopnie" na podstawie trudności i kosztów.

a) Niektóre metody wymagają tylko jednokrotnego wczytania danych z dysku. Są to algorytmy *jednoprzebiegowe*. Omówiono je w tym podrozdziale. Zwykle przynajmniej jeden z argumentów musi mieścić się w pamięci głównej, choć zdarzają się wyjątki, zwłaszcza w zakresie selekcji i projekcji, co opisano w podrozdziale 15.2.1.
b) Niektóre metody działają dla danych, które nie mieszczą się w dostępnej pamięci głównej, ale nie działają dla największych wyobrażalnych zbiorów danych. Są to algorytmy *dwuprzebiegowe*, cechujące się początkowym odczytem danych z dysku, przetwarzaniem ich w pewien sposób, zapisem wszystkich (lub prawie wszystkich) na dysku i późniejszym wczytywaniem ich po raz wtóry w celu dalszego przetwarzania w ramach drugiego przebiegu. Algorytmy tego typu pojawiają się w podrozdziałach 15.4 i 15.5.
c) Niektóre metody działają dla dowolnie dużych danych. Metody te wymagają trzech lub więcej przebiegów i są naturalnym, rekurencyjnym uogólnieniem algorytmów dwuprzebiegowych. Metody wieloprzebiegowe omówiono w podrozdziale 15.8.

W tym podrozdziale skoncentrowano się na metodach jednoprzebiegowych. Tu i w dalszych punktach operatory dzielone są na trzy ogólne grupy.

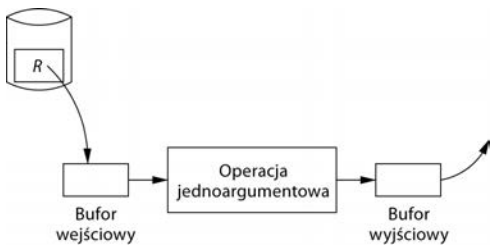
(1) *Krotkowe operacje jednoargumentowe (działające na pojedynczych krotkach)*. Te operacje — selekcja i projekcja — nie wymagają jednoczesnego zapisania całej relacji (a nawet dużej jej części) w pamięci. Dlatego można wczytywać dane blok po bloku, używać jednego bufora w pamięci głównej i generować dane wyjściowe.

(2) *Operacje jednoargumentowe na całych relacjach*. Jednoargumentowe operacje tego typu wymagają jednoczesnego umieszczenia w pamięci wszystkich lub większości krotek, dlatego algorytmy jednoprzebiegowe działają tylko dla relacji o rozmiarze M (liczba dostępnych buforów pamięci głównej) lub mniejszym. Operacje tej klasy to γ (grupowanie) i δ (usuwanie powtórzeń).

(3) *Operacje dwuargumentowe na całych relacjach*. Do tej klasy należą wszystkie pozostałe operacje — suma, część wspólna, różnica, złączenia i iloczyny w wersjach dla zbiorów oraz wielozbiorów. Aby zastosować algorytm jednoprzebiegowy, w każdej operacji, z wyjątkiem sumy wielozbiorów, przynajmniej jeden argument musi mieć rozmiar nie większy niż M.

15.2.1. Algorytmy jednoprzebiegowe dla operacji krotkowych

Operacjom krotkowym, $\sigma(R)$ i $\pi(R)$, odpowiadają oczywiste algorytmy niezależnie od tego, czy relacja mieści się w pamięci głównej. Należy wczytywać bloki relacji R jeden po drugim do bufora wejściowego, wykonywać operacje na każdej krotce i przenosić pobrane lub zrzutowane krotki do bufora wyjściowego, co pokazano na rysunku 15.5. Ponieważ bufor wyjściowy może być buforem wejściowym innego operatora lub udostępniać dane użytkownikowi albo aplikacji, nie uwzględniono bufora wyjściowego przy obliczaniu potrzebnej pamięci. Dlatego konieczne jest tylko, aby — niezależnie od $B - M \geq 1$ dla bufora wejściowego.



15.5. *Przeprowadzanie selekcji lub projekcji na relacji R*

Liczba dyskowych operacji wejścia-wyjścia zależy tylko od sposobu udostępniania relacji R będącej argumentem. Jeśli R początkowo znajduje się na dysku, koszt odpowiada skanowaniu tabeli lub indeksu dla R. Koszt tego typu omówiono w punkcie 15.1.5 — zwykle wynosi B dla sklastrowanej R i T dla niesklastrowanej R. Warto jednak pamiętać o ważnym wyjątku, kiedy w warunku selekcji stała porównywana jest z atrybutem, na którym oparty jest indeks. Wtedy można użyć indeksu do pobrania podzbioru bloków przechowujących R i poprawienia w ten sposób wydajności — często w znacznym stopniu.

Dodatkowe bufory pozwalają przyspieszyć operacje

Choć operacje działające krotka po krotce można wykonać z jednym buforem wejściowym i jednym buforem wyjściowym, co pokazano na rysunku 15.5, często można przyspieszyć przetwarzanie przez przydział dodatkowych buforów wejściowych. Pomysł ten po raz pierwszy pojawił się w punkcie 13.3.2. Jeśli R jest zapisana w cylindrach w przyległych blokach, można do buforów wczytać cały cylinder. Koszt wyszukiwania i opóźnienia obrotowego ponoszony jest tylko dla jednego bloku na cylinder. Podobnie jeśli dane wyjściowe operacji można zapisać w pełnych cylindrach, zapis nie wiąże się z prawie żadną stratą czasu.

15.2.2. Algorytmy jednoprzebiegowe dla jednoargumentowych operacji na całych relacjach

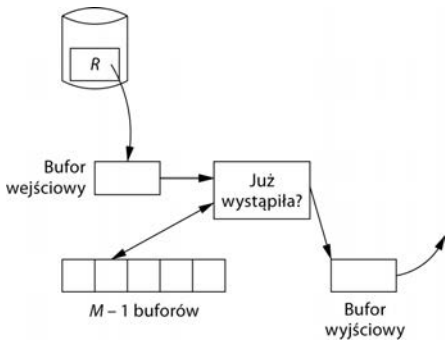
Rozważmy teraz operacje jednoargumentowe stosowane do całych relacji, a nie do pojedynczych krotek, czyli eliminowanie powtórzeń (δ) i grupowanie (γ).

Eliminowanie powtórzeń

Aby wyeliminować powtórzenia, można wczytać jeden po drugim każdy blok z R, przy czym dla każdej krotki trzeba ustalić, czy:

(1) jest to pierwsze wystąpienie danej krotki (wtedy należy skopiować ją do danych wyjściowych);
(2) krotka pojawiła się już wcześniej (wtedy nie należy jej zwracać).

Aby to ustalić, trzeba przechowywać w pamięci jedną kopię każdej napotkanej krotki, co pokazano na rysunku 15.6. Jeden bufor przechowuje w pamięci jeden blok krotek z R, a pozostałe $M - 1$ buforów można wykorzystać do przechowywania pojedynczych egzemplarzy każdej pobranej krotki.



15.6. *Zarządzanie pamięcią przy jednoprzebiegowym eliminowaniu powtórzeń*

Do zapisywania napotkanych krotek trzeba odpowiednio dobrać strukturę danych w pamięci głównej. Naiwne podejście to użycie listy krotek. Przy sprawdzaniu nowej krotki z R należy wtedy porównać ją ze wszystkimi napotkanymi krotkami. Jeśli nie jest równa żadnej z nich, trzeba skopiować ją do danych wyjściowych i dodać do przechowywanej w pamięci listy pobranych krotek.

Jeśli jednak w pamięci głównej znajduje się n krotek, każda nowa krotka zajmuje czas procesora proporcjonalnie do n, dlatego cała operacja potrzebuje czasu proporcjonalnego do n^2. Ponieważ n może być bardzo duże, ilość potrzebnego czasu podaje w wątpliwość założenie, że ważny jest tylko czas wykonywania dyskowych operacji wejścia-wyjścia. Dlatego w pamięci głównej potrzebna jest struktura umożliwiająca dodanie nowej krotki i określenie w czasie rosnącym powoli względem n, czy dana krotka już się pojawiła.

Można na przykład użyć tablicy z haszowaniem z dużą liczbą kubełków lub jednej z wersji zbalansowanego binarnego drzewa wyszukiwań[1]. Każda z tych struktur wymaga dodatkowego

[1] Omówienie odpowiednich struktur pamięci głównej można znaleźć w: A. V. Aho, J. E. Hopcroft i J. D. Ullman, *Data Structures and Algorithms*, 1983 (wydanie polskie: *Algorytmy i struktury danych*, Helion, 2003). Przetwarzanie n elementów za pomocą haszowania zajmuje średnio $O(n)$, a z wykorzystaniem drzew zbalansowanych — $O(n \log n)$. W obu przypadkach wzrost długości czasu przetwarzania jest wystarczająco bliski liniowemu, jak na nasze potrzeby.

miejsca obok pamięci potrzebnej na przechowywanie krotek. Przykładowo tablica z haszowaniem w pamięci głównej wymaga tablicy kubełków i miejsca na wskaźniki prowadzące do krotek w kubełku. Jednak dodatkowe koszty są niskie w porównaniu z pamięcią potrzebną na zapisanie krotek, dlatego w tym rozdziale je pominięto.

Zgodnie z tym założeniem w $M - 1$ dostępnych buforach pamięci głównej można zapisać tyle krotek, ile mieści się w $M - 1$ blokach relacji R. Jeśli w pamięci głównej ma zmieścić się jedna kopia każdej niepowtarzalnej krotki z R, $B(\delta(R))$ musi być nie większe niż $M - 1$. Ponieważ zakładamy, że M jest znacznie większe niż 1, zwykle stosowane jest uproszczone przybliżenie tej reguły:

- $B(\delta(R)) \leq M$

Zauważmy, że zwykle nie można określić rozmiaru $\delta(R)$, nie obliczając samego $\delta(R)$. Jeśli rzeczywisty rozmiar będzie większy od oszacowanego ($B(\delta(R))$ jest większe od M), trzeba ponieść istotne koszty związane z przeładowywaniem, ponieważ bloki przechowujące różne krotki z R trzeba często przenosić do pamięci i z niej usuwać.

Grupowanie

Operacja grupowania, γ_L, zwraca zero lub więcej atrybutów grupujących i zwykle jeden lub więcej atrybutów zagregowanych. Po utworzeniu w pamięci głównej jednego wpisu dla każdej grupy — dla każdej wartości atrybutów grupujących — można przejrzeć krotki z R po jednym bloku na raz. *Wpis* dla grupy składa się z wartości atrybutów grupujących i zakumulowanej wartości każdej agregacji.

- Dla agregacji `MIN(a)` lub `MAX(a)` należy zapisać minimalną lub maksymalną wartość atrybutu a dla wszystkich sprawdzonych już krotek z grupy. Jeśli to konieczne, należy zmienić minimum lub maksimum przy pobraniu każdej krotki z grupy.
- Dla agregacji `COUNT` należy dodać 1 dla każdej pobranej krotki z danej grupy.
- Dla agregacji `SUM(a)` należy dodać wartość atrybutu a do zakumulowanej sumy dla danej grupy (pod warunkiem, że a jest różne od `NULL`).
- `AVG(a)` to trudny przypadek. Trzeba przechowywać dwie zakumulowane wartości — liczbę krotek w grupie i sumę wartości atrybutu a w tych krotkach. Obie wartości należy obliczać tak, jak dla agregacji `COUNT` i `SUM`. Po sprawdzeniu wszystkich krotek z R trzeba podzielić sumę przez liczbę krotek, aby uzyskać średnią.

Po wczytaniu wszystkich krotek z R do bufora wejściowego i użyciu ich do obliczenia agregacji dla odpowiednich grup można utworzyć dane wyjściowe, zapisując krotkę dla każdej grupy. Warto zauważyć, że do czasu sprawdzenia ostatniej krotki nie można rozpocząć tworzenia danych wyjściowych dla operacji γ. Dlatego algorytm ten nie pasuje do modelu działania iteratorów. W metodzie `Open` trzeba zakończyć grupowanie, zanim za pomocą metody `GetNext` będzie można pobrać pierwszą krotkę.

Aby przetwarzanie każdej krotki w pamięci było wydajne, trzeba w pamięci głównej użyć struktury danych umożliwiającej znalezienie wpisu dla każdej grupy na podstawie wartości atrybutów grupujących. Jak opisano to w kontekście operacji δ, dobrze nadają się do tego typowe struktury danych stosowane w pamięci głównej, takie jak tablice z haszowaniem i drzewa zbalansowane. Należy jednak pamiętać, że kluczem wyszukiwania dla tych struktur są tylko atrybuty grupujące.

Operacje na niesklastrowanych danych

Wszystkie obliczenia dotyczące liczby dyskowych operacji wejścia-wyjścia wymaganych dla omawianych operacji są oparte na założeniu, że relacje podane jako operandy są sklastrowane. W — zwykle rzadkich — sytuacjach, kiedy operand R nie jest sklastrowany, wczytanie wszystkich krotek z R może wymagać $T(R)$, a nie $B(R)$ dyskowych operacji wejścia-wyjścia. Zauważmy jednak, że można przyjąć, iż każda relacja zwracana przez operator jest sklastrowana — nie ma powodu, aby zapisywać tymczasową relację w niesklastrowany sposób.

Liczba dyskowych operacji wejścia-wyjścia potrzebnych dla tego jednoprzebiegowego algorytmu wynosi B (jest tak w każdym jednoprzebiegowym algorytmie dla operatora jednoargumentowego). Liczba niezbędnych buforów pamięci M nie jest powiązana z B w żaden prosty sposób, choć zwykle M ma wartość mniejszą niż B. Problem polega na tym, że wpisy dla grup mogą być dłuższe lub krótsze niż krotki z R, a liczba grup może być równa liczbie krotek z R lub mniejsza. Jednak zwykle wpisy dla grup nie są dłuższe od krotek z R, a grup jest dużo mniej niż krotek.

15.2.3. Algorytmy jednoprzebiegowe dla operacji dwuargumentowych

Dalej opisano operacje dwuargumentowe — sumę, część wspólną, różnicę, iloczyn i złączenie. Ponieważ w niektórych przypadkach trzeba odróżnić wersje operatorów oparte na zbiorach i wielozbiorach, występują przy nich indeksy dolne Z (dla zbiorów) i W (dla wielozbiorów). Przykładowo $\cup_W$ oznacza sumę wielozbiorów, a $-_Z$ — różnicę zbiorów. Aby uprościć omawianie złączeń, uwzględniono tylko złączenia naturalne. Złączenie równościowe można zaimplementować w ten sam sposób (po odpowiednim przemianowaniu atrybutów), a złączenia warunkowe można potraktować jak iloczyn lub złączenie równościowe, po którym następuje selekcja dla warunków niewyrażalnych w złączeniu równościowym.

Sumę wielozbiorów można obliczyć za pomocą bardzo prostego algorytmu jednoprzebiegowego. Aby obliczyć $R \cup_W S$, należy skopiować do danych wyjściowych każdą krotkę z R, a następnie z S, tak jak w przykładzie 15.3. Liczba dyskowych operacji wejścia-wyjścia to $B(R) + B(S)$, jak zawsze w jednoprzebiegowych algorytmach na operandach R i S, natomiast $M = 1$ jest wystarczającą wartością niezależnie od wielkości R i S.

Inne operacje dwuargumentowe wymagają wczytania mniejszego z operandów R i S do pamięci głównej oraz utworzenia odpowiedniej struktury danych, tak aby krotki można było szybko wstawiać i znajdować, co opisano w punkcie 15.2.2. Tak jak wcześniej, wystarczająca jest tablica z haszowaniem lub drzewo zbalansowane. Dlatego w przybliżeniu można określić wymóg dla operacji dwuargumentowych na relacjach R i S, jeśli mają być wykonywane w jednym przebiegu:

- $\min(B(R), B(S)) \leq M$

Ujmijmy to dokładniej — jeden bufor służy do wczytywania bloków większej relacji, natomiast około M buforów trzeba na całą mniejszą relację i jej strukturę danych przechowywaną w pamięci głównej.

Dalej przedstawiono szczegóły różnych operacji. W każdym przypadku założono, że R jest większą z relacji, dlatego S jest przechowywana w pamięci głównej.

Suma zbiorów

Należy wczytać S do $M - 1$ buforów w pamięci głównej i zbudować ułatwiającą wyszukiwanie strukturę, której kluczem wyszukiwania jest cała krotka. Wszystkie krotki są też kopiowane do danych wyjściowych. Następnie trzeba wczytać jeden po drugim każdy blok z R do M-tego bufora. Dla każdej krotki t z R trzeba sprawdzić, czy t znajduje się w S. Jeśli nie, należy skopiować t do danych wyjściowych. Jeżeli t występuje w S, t można pominąć.

Część wspólna zbiorów

Należy wczytać S do $M - 1$ buforów i zbudować ułatwiającą wyszukiwanie strukturę, której kluczem wyszukiwania są całe krotki. Trzeba wczytać każdy blok R i dla każdej krotki z R sprawdzić, czy t znajduje się także w S. Jeśli tak, należy skopiować t do danych wyjściowych; w przeciwnym razie t można pominąć.

Różnica zbiorów

Ponieważ różnica nie jest przemienna, trzeba rozróżnić wyrażenie $R -_Z S$ od $S -_Z R$ (nadal obowiązuje założenie, że R to większa relacja). W obu przypadkach należy wczytać S do $M - 1$ buforów i zbudować ułatwiającą wyszukiwanie strukturę, której kluczem wyszukiwania są całe krotki.

Aby obliczyć $R -_Z S$, trzeba wczytać każdy blok z R i sprawdzić każdą krotkę t z tego bloku. Jeśli t znajduje się w S, należy pominąć t; w przeciwnym razie trzeba skopiować t do danych wyjściowych.

Aby obliczyć $S -_Z R$, także trzeba wczytać wszystkie bloki R i po kolei sprawdzić każdą krotkę t. Jeśli t znajduje się w S, należy usunąć t z przechowywanej w pamięci głównej kopii S. Jeżeli t nie występuje w S, nie trzeba nic robić. Po sprawdzeniu wszystkich krotek z R można skopiować do danych wyjściowych wszystkie pozostałe krotki z S.

Część wspólna wielozbiorów

Należy wczytać S do $M - 1$ buforów i powiązać każdą wartość krotki z *licznikiem*, który początkowo mierzy liczbę wystąpień krotki w S. Poszczególne kopie krotki t nie są przechowywane osobno. Zamiast tego należy zapisać jedną kopię t i powiązać ją z licznikiem określającym liczbę wystąpień t.

Jeśli powtórzeń jest niewiele, struktura ta może zająć nieco więcej niż $B(S)$ bloków, choć często pozwala zmniejszyć S. Dlatego nadal obowiązuje założenie, że $B(S) \leq M$ wystarczy do działania algorytmu jednoprzebiegowego, choć ten warunek to tylko przybliżenie.

Następnie trzeba wczytać każdy blok R i dla każdej krotki t z R sprawdzić, czy t występuje w S. Jeśli nie, można pominąć t — krotka ta nie należy do części wspólnej. Jeżeli jednak t pojawia się w S, a licznik powiązany z t ma wartość dodatnią, trzeba dodać t do danych wyjściowych i zmniejszyć licznik o 1. Jeśli t występuje w S, jednak licznik doszedł do 0, nie należy zwracać t — w danych wyjściowych znajduje się już tyle kopii t, ile istniało ich w S.

Różnica wielozbiorów

Aby obliczyć $S -_W R$, należy wczytać krotki z S do pamięci głównej i ustalić liczbę wystąpień krotek o różnych wartościach (tak jak dla części wspólnej wielozbiorów). Przy wczytywaniu każdej krotki t z R należy sprawdzić, czy występuje ona w S. Jeśli tak, trzeba zmniejszyć wartość powiązanego licznika. Na koniec trzeba skopiować z pamięci głównej do danych wyjściowych każdą krotkę o pozytywnej wartości licznika (wartość ta wyznacza liczbę kopii).

Aby obliczyć $R -_W S$, także trzeba wczytać krotki z S do pamięci głównej i ustalić liczbę wystąpień różnych krotek. Krotkę t z licznikiem o wartości c można traktować jak c powodów,

aby nie kopiować t do danych wyjściowych przy wczytywaniu krotek z R. Przy wczytywaniu krotki t z R należy sprawdzić, czy t występuje w S. Jeśli nie, t kopiowana jest do danych wyjściowych. Jeżeli t pojawia się w S, trzeba sprawdzić wartość c powiązaną z t. Jeśli $c = 0$, należy skopiować t do danych wyjściowych. Przy $c > 0$ t nie jest kopiowana, natomiast trzeba zmniejszyć wartość c o 1.

Iloczyn

Należy wczytać S do $M - 1$ buforów pamięci głównej. Nie jest potrzebna żadna specjalna struktura danych. Następnie wczytywany jest każdy blok z R. Każdą krotkę t z R należy złączyć t z każdą krotką S z pamięci głównej. Złączone krotki trzeba dodać do danych wyjściowych.

Algorytm ten wymaga dużej ilości czasu procesora na krotkę z R, ponieważ każdą z nich trzeba dopasować do $M - 1$ bloków pełnych krotek. Jednak dane wyjściowe także są duże, a czas w przeliczeniu na krotkę z tych danych jest krótki.

Złączenie naturalne

W tym i innych algorytmach złączenia przyjęto, że relacja $R(X, Y)$ jest złączana z $S(Y, Z)$, gdzie Y reprezentuje wszystkie atrybuty wspólne dla R i S, X to wszystkie atrybuty z R, które nie występują w S, a Z to atrybuty z S nieobecne w R. Nadal zakładamy, że S to mniejsza relacja. Aby obliczyć złączenie naturalne, należy wykonać następujące operacje.

(1) Wczytać wszystkie krotki z S i umieścić je w pamięci głównej we wspomagającej wyszukiwanie strukturze (z atrybutami Y jako kluczem wyszukiwania). Można wykorzystać na to $M - 1$ bloków pamięci.

(2) Wczytać każdy blok z R do jednego pozostałego bufora pamięci głównej. Używając struktury ułatwiającej wyszukiwanie, należy dla każdej krotki t z R znaleźć krotki z S o zgodnych atrybutach Y. Dla każdej pasującej krotki z S trzeba utworzyć krotkę przez złączenie jej z t. Wynikowa krotka jest umieszczana w danych wyjściowych.

Podobnie jak wszystkie jednoprzebiegowe algorytmy dwuargumentowe, ten wymaga $B(R) + B(S)$ dyskowych operacji wejścia-wyjścia do wczytania operandów. Działa dla $B(S) \leq M - 1$ (w przybliżeniu dla $B(S) \leq M$).

Nie omawiamy tu złączeń innych niż naturalne. Warto pamiętać, że złączenie równościowe jest obliczane w zasadzie tak samo, jednak trzeba uwzględnić fakt, iż równe atrybuty z obu relacji mogą mieć różne nazwy. Złączenie warunkowe, które nie jest złączeniem równościowym, można zastąpić złączeniem równościowym lub iloczynem, po którym następuje selekcja.

15.2.4. Ćwiczenia do podrozdziału 15.2

Ćwiczenie 15.2.1. Dla każdej z podanych operacji napisz iterator z wykorzystaniem algorytmu opisanego w tej sekcji: a) projekcja, b) usuwanie powtórzeń (δ), c) grupowanie (γ_L), d) suma zbiorów, e) część wspólna zbiorów, f) różnica zbiorów, g) część wspólna wielozbiorów, h) różnica wielozbiorów, i) iloczyn, j) złączenie naturalne.

Ćwiczenie 15.2.2. Dla każdego z operatorów z ćwiczenia 15.2.1 określ, czy operator jest *blokujący* (czyli czy pierwsze dane wyjściowe można utworzyć dopiero po wczytaniu wszystkich danych wejściowych). Ujmijmy to inaczej — operator blokujący to taki, dla którego jedyne możliwe iteratory wykonują wszystkie ważne zadania w funkcji `Open`.

Ćwiczenie 15.2.3. W tabeli 15.9 pokazano wymogi dotyczące pamięci i dyskowych operacji wejścia-wyjścia dla algorytmów opisanych w tym i następnym podrozdziale. Założono jednak, że wszystkie argumenty są sklastrowane. Jak zmienią się wartości, jeśli jeden lub oba argumenty nie są sklastrowane?

! **Ćwiczenie 15.2.4.** Przedstaw algorytmy jednoprzebiegowe dla każdego z poniższych operatorów z rodziny złączeń:

a) $R \ltimes S$ przy założeniu, że R mieści się w pamięci (definicję złączenia częściowego przedstawiono w ćwiczeniu 2.4.8).
b) $R \ltimes S$ przy założeniu, że S mieści się w pamięci.
c) $R \overline{\ltimes} S$ przy założeniu, że R mieści się w pamięci (definicję nierównozłączenia częściowego przedstawiono w ćwiczeniu 2.4.9).
d) $R \overline{\ltimes} S$ przy założeniu, że S mieści się w pamięci.
e) $R \overset{\circ}{\ltimes}_L S$ przy założeniu, że R mieści się w pamięci (definicje złączeń zewnętrznych zawiera punkt 5.2.7).
f) $R \overset{\circ}{\ltimes}_L S$ przy założeniu, że S mieści się w pamięci.
g) $R \overset{\circ}{\ltimes}_R S$ przy założeniu, że R mieści się w pamięci.
h) $R \overset{\circ}{\ltimes}_R S$ przy założeniu, że S mieści się w pamięci.
i) $R \overset{\circ}{\ltimes} S$ przy założeniu, że R mieści się w pamięci.

15.3. Złączenia zagnieżdżone

Przed przejściem do bardziej złożonych algorytmów przedstawionych w następnych podrozdziałach warto przyjrzeć się rodzinie algorytmów dla operatora złączenia zagnieżdżonego (z pętlą). Algorytmy te wykonują w pewnym sensie półtora przebiegu, ponieważ w każdej wersji dla jednego z dwóch argumentów krotki wczytywane są tylko raz, natomiast dla drugiego — wielokrotnie. Złączeń zagnieżdżonych można używać dla relacji dowolnej wielkości. Nie jest konieczne, aby jedna z relacji mieściła się w pamięci głównej.

15.3.1. Krotkowe złączenia zagnieżdżone

Najprostsza odmiana złączeń zagnieżdżonych obejmuje pętle przechodzące po poszczególnych krotkach używanych relacji. W tym algorytmie, *krotkowym złączeniu zagnieżdżonym*, złączenie $R(X, Y) \bowtie S(Y, Z)$ obliczane jest tak:

```
FOR każda krotka s z S DO
   FOR każda krotka r z R DO
      IF r i s można złączyć w krotkę t THEN
         zwróć t;
```

Przy nieostrożnym buforowaniu bloków relacji R i S algorytm może wymagać aż $T(R)T(S)$ dyskowych operacji wejścia-wyjścia. Jednak w wielu sytuacjach algorytm można zmodyfikować, tak aby znacznie obniżyć koszty. Jedną z możliwości jest użycie indeksu na atrybucie lub atrybutach z R uwzględnianych w złączeniu, aby znaleźć krotki z R pasujące do danej krotki z S bez konieczności wczytywania całej relacji R. Złączenia oparte na indeksach omówiono w punkcie 15.6.3. Drugie usprawnienie wymaga starannego określenia podziału krotek z R i S na bloki. Należy wykorzystać maksymalnie dużo pamięci, aby zmniejszyć liczbę dyskowych operacji wejścia-wyjścia przy przechodzeniu przez pętlę wewnętrzną. W punkcie 15.3.3 omówiono wersję złączeń zagnieżdżonych opartą na blokach.

15.3.2. Iterator dla krotkowych złączeń zagnieżdżonych

Zaletą złączeń zagnieżdżonych jest to, że dobrze wpasowują się w model iteratorów, co — jak opisano w punkcie 16.7.3 —w niektórych sytuacjach pozwala uniknąć zapisywania relacji pośrednich na dysku. Iterator dla operacji $R \bowtie S$ można łatwo utworzyć na podstawie iteratorów dla R i S, które obsługują metody `R.Open()` i inne opisane w punkcie 15.1.6. Kod trzech metod iteratora dla złączeń zagnieżdżonych pokazano na listingu 15.7 (przyjęto założenie, że R i S nie są puste).

```
Open() {
   R.Open();
   S.Open();
   s := S.GetNext();
}

GetNext() {
   REPEAT {
      r := R.GetNext();
      IF (r = NotFound) { /* Pobrano całą relację R dla
            obecnego s. */
         R.Close();
         s := S.GetNext();
         IF (s = NotFound) RETURN NotFound;
            /* Pobrano całe relacje R i S. */
         R.Open();
         r := R.GetNext();
      }
   }
   UNTIL (r i s zostaną złączone);
   RETURN złączenie r i s;
}

Close() {
   R.Close();
   S.Close();
}
```

15.7. *Metody iteratora dla krotkowych złączeń zagnieżdżonych dla relacji R i S*

15.3.3. Algorytm złączenia zagnieżdżonego opartego na blokach

Można usprawnić przedstawione w punkcie 15.3.1 krotkowe złączenie zagnieżdżone. Przy obliczaniu $R \bowtie S$ warto zadbać o:

(1) zapewnienie dostępu do obu argumentów (relacji) według bloków,
(2) wykorzystanie maksymalnej ilości pamięci głównej do zapisania krotek należących do relacji S (relacji z pętli zewnętrznej).

Punkt 1. gwarantuje, że przy przechodzeniu w wewnętrznej pętli po krotkach R do wczytania R potrzebna będzie jak najmniejsza liczba dyskowych operacji wejścia-wyjścia. Punkt 2. pozwala złączyć każdą wczytaną krotkę z R z tyloma krotkami z S, ile mieści się w pamięci.

Tak jak w punkcie 15.2.3, załóżmy, że $B(S) \leq B(R)$. Przyjmijmy jednak, że $B(S) > M$ (żadna relacja nie mieści się w całości w pamięci głównej). Trzeba wielokrotnie wczytać $M - 1$ bloków z S do buforów pamięci głównej. Dla krotek S w pamięci głównej należy utworzyć wspomagającą wyszukiwanie strukturę, której klucze wyszukiwania to wspólne atrybuty R i S. Następnie trzeba przejść przez wszystkie bloki relacji R, wczytując po kolei każdy z nich do ostatniego bloku pamięci. Wtedy można porównać wszystkie krotki bloku R ze wszystkimi krotkami z bloków S wczytanych do pamięci głównej. Pasujące krotki należy zwrócić jako złączoną krotkę. Struktura pętli zagnieżdżonej z tego algorytmu jest widoczna w bardziej formalnym jego ujęciu na listingu 15.8. Algorytm z listingu jest czasem nazywany blokowym złączeniem zagnieżdżonym. Dalej nazywamy go po prostu *złączeniem zagnieżdżonym*, ponieważ jest to wersja rozwiązania z pętlą zagnieżdżoną najczęściej stosowana w praktyce.

```
FOR każdej porcji M-1 bloków S DO BEGIN
   wczytaj bloki do buforów pamięci głównej;
   uporządkuj krotki we wspomagającej wyszukiwanie strukturze,
      której kluczem wyszukiwania są wspólne atrybuty R i S;
   FOR każdego bloku b z R DO BEGIN
      wczytaj b do pamięci głównej;
      FOR każdej krotki t z b DO BEGIN
         znajdź w pamięci głównej krotki z S, które
            można złączyć z t;
         zwróć złączenie t z każdą z takich krotek;
      END;
   END;
END;
```

15.8. *Algorytm złączenia zagnieżdżonego*

Program z listingu 15.8 wygląda tak, jakby miał trzy pętle zagnieżdżone. Jeśli jednak spojrzeć na kod na odpowiednim poziomie abstrakcji, w rzeczywistości istnieją tylko dwie pętle. Pierwsza, zewnętrzna, przechodzi po krotkach z S. Dwie pozostałe przechodzą po krotkach z R — użyto tu dwóch pętli, aby podkreślić fakt, że kolejność przechodzenia po krotkach z R nie jest dowolna. Trzeba sprawdzać krotki blok po bloku (służy do tego druga pętla), a w bloku należy sprawdzić wszystkie krotki przed przejściem do następnego bloku (odpowiada za to trzecia pętla).

Przykład 15.4. Niech $B(R) = 1000$, $B(S) = 500$, a $M = 101$. 100 bloków pamięci posłuży do buforowania S w 100-blokowych częściach, dlatego zewnętrzna pętla z listingu 15.8 zostanie wykonana pięć razy. W każdej iteracji trzeba wykonać 100 dyskowych operacji wejścia-wyjścia do wczytania części relacji S, a ponieważ w drugiej pętli trzeba wczytać całą relację R, zajmuje to 1000 dyskowych operacji wejścia-wyjścia. Tak więc łączna liczba takich operacji to 5500.

Zauważmy, że po zamianie rolami relacji R i S algorytm będzie wymagał nieco więcej dyskowych operacji wejścia-wyjścia. Trzeba 10 razy wykonać zewnętrzną pętlę, a w każdym powtórzeniu wykonać 600 operacji, co w sumie daje ich 6000. Przetwarzanie mniejszej relacji w zewnętrznej pętli zwykle daje pewne korzyści. ■

15.3.4. Analiza złączeń zagnieżdżonych

Analizy z przykładu 15.4 można powtórzyć dla dowolnego $B(R)$, $B(S)$ i M. Przy założeniu, że S to mniejsza relacja, liczba jej części (iteracji zewnętrznej pętli) wynosi $B(S)/(M - 1)$. W każdej iteracji należy wczytać $M - 1$ bloków z S i $B(R)$ bloków z R. Liczba dyskowych operacji wejścia-wyjścia wynosi więc $B(S)(M - 1 + B(R))/(M - 1)$ lub $B(S) + (B(S)B(R))/(M - 1)$.

Przy założeniu, że M, $B(S)$ i $B(R)$ to duże wartości, jednak M jest najmniejszą z nich, przybliżeniem powyższego wzoru jest $B(S)B(R)/M$. Oznacza to, że koszt jest proporcjonalny do iloczynu rozmiarów obu relacji podzielonego przez ilość dostępnej pamięci głównej. Jeśli obie relacje są duże, można uzyskać znacznie wyższą wydajność niż dla złączeń zagnieżdżonych. Jednak dla małych relacji, takich jak w przykładzie 15.4, koszt dla takich złączeń nie jest dużo wyższy niż dla złączenia jednoprzebiegowego, które tu wymagałoby 1500 operacji. W rzeczywistości dla $B(S) \leq M - 1$ złączenie zagnieżdżone działa tak samo jak jednoprzebiegowy algorytm złączenia opisany w punkcie 15.2.3.

Choć złączenie zagnieżdżone zwykle nie jest najwydajniejszym algorytmem złączenia, warto zauważyć, że w niektórych wczesnych relacyjnych systemach DBMS była to jedyna dostępna metoda. Nawet obecnie jest ona czasem potrzebna jako podprocedura w wydajniejszych algorytmach złączenia, na przykład wtedy, kiedy duża liczba krotek z każdej relacji ma tę samą wartość atrybutów używanych przy złączeniu. Przykład, w którym złączenie zagnieżdżone jest niezbędne, przedstawiono w punkcie 15.4.6.

15.3.5. Przegląd opisanych wcześniej algorytmów

W tabeli 15.9 pokazano wymogi związane z pamięcią główną i dyskowymi operacjami wejścia-wyjścia dla algorytmów omówionych w podrozdziałach 15.2 i 15.3. Wymogi dotyczące pamięci dla operacji γ i δ są bardziej złożone — $M = B$ to tylko luźne przybliżenie. Dla γ M zależy od liczby grup, a dla δ — od liczby różnych krotek.

Operatory	Potrzebne M (w przybliżeniu)	Operacje dyskowe	Punkt
σ, π	1	B	15.2.1
γ, δ	B	B	15.2.2
$\cup, \cap, -, \times, \bowtie$	$min(B(R), B(S))$	$B(R) + B(S)$	15.2.3
$\bowtie$	Dowolne $M \geq 2$	$B(R)B(S)/M$	15.3.3

15.9. *Wymogi związane z pamięcią główną i dyskowymi operacjami wejścia-wyjścia dla algorytmów jednoprzebiegowych oraz zagnieżdżonych*

15.3.6. Ćwiczenia do podrozdziału 15.3

Ćwiczenie 15.3.1. Przedstaw trzy metody iteratora dla wersji złączeń zagnieżdżonych opartej na blokach.

Ćwiczenie 15.3.2. Załóżmy, że $B(R) = B(S) = 10\,000$, a $M = 1000$. Oblicz koszt dyskowych operacji wejścia-wyjścia dla złączenia zagnieżdżonego.

Ćwiczenie 15.3.3. Używane są relacje z ćwiczenia 15.3.2. Jaka wartość M jest potrzebna do obliczenia $R \bowtie S$ za pomocą algorytmu zagnieżdżonego w nie więcej niż: a) 100 000, !b) 25 000, !c) 15 000 dyskowych operacji wejścia-wyjścia?

! **Ćwiczenie 15.3.4.** Może się wydawać, że jeśli R i S są niesklastrowane, złączenie zagnieżdżone wymaga około $T(R)T(S)/M$ dyskowych operacji wejścia-wyjścia.

a) Jak można w istotny sposób zmniejszyć ten koszt?

b) Jak można przeprowadzić złączenie zagnieżdżone, jeśli tylko jedna relacja (R lub S) jest niesklastrowana? Rozważ zarówno przypadek, kiedy niesklastrowana jest większa relacja, jak i sytuację, kiedy niesklastrowana jest mniejsza relacja.

! **Ćwiczenie 15.3.5.** Iterator z listingu 15.7 nie działa poprawnie, jeśli R lub S jest pusta. Zmodyfikuj metody, tak aby działały nawet wtedy, kiedy jedna lub obie relacje są puste.

15.4. Algorytmy dwuprzebiegowe oparte na sortowaniu

Od tego miejsca rozpoczyna się omówienie algorytmów wieloprzebiegowych. Służą one do wykonywania operacji algebry relacji na relacjach zbyt dużych dla jednoprzebiegowych algorytmów z podrozdziału 15.2. Skoncentrowano się tu na *algorytmach dwuprzebiegowych*, w których dane z operandów (relacji) są wczytywane do pamięci głównej, przetwarzane w pewien sposób, ponownie zapisywane na dysku, a następnie powtórnie wczytywane z dysku w celu dokończenia operacji. Naturalnie można rozwinąć tę technikę do dowolnej liczby przebiegów, w których dane są wielokrotnie wczytywane do pamięci głównej. Skoncentrowano się jednak na algorytmach dwuprzebiegowych, ponieważ:

a) dwa przebiegi to zwykle wystarczająca liczba nawet dla bardzo dużych relacji,

b) uogólnienie na więcej niż dwa przebiegi nie jest trudne. Omówiono je w punkcie 15.4.1 i — bardziej ogólnie — w podrozdziale 15.8.

Na początku opisano implementację operatora sortowania, τ, aby zilustrować ogólne podejście — podział relacji R, dla której $B(R) > M$, na porcje o wielkości M, sortowanie ich, a następnie przetwarzanie posortowanych podlist, tak że w danym momencie w pamięci głównej potrzebny jest tylko jeden blok z posortowaną podlistą.

15.4.1. Dwuetapowe wielościeżkowe sortowanie przez scalanie

Bardzo duże relacje można sortować w dwóch przebiegach, używając algorytmu *dwuetapowego wielościeżkowego sortowania przez scalanie* (DWSPS; ang. *two-phase, multiway merge sort*). Załóżmy, że przy sortowaniu można użyć M buforów w pamięci głównej. DWSPS sortuje relację R w następujący sposób.

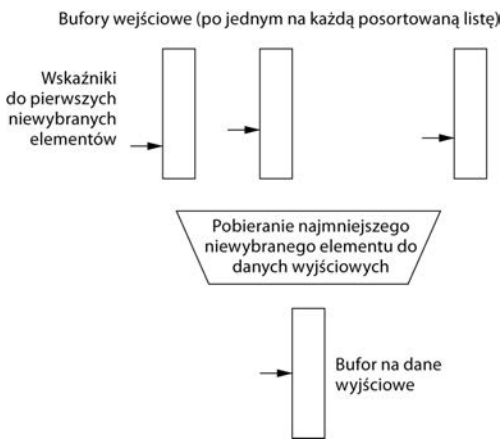
- *Etap 1*. Należy wielokrotnie zapełnić M buforów nowymi krotkami z R i posortować je, używając algorytmu sortowania w pamięci głównej. Każdą *posortowaną podlistę* trzeba zapisać w pamięci drugiego stopnia.
- *Etap 2*. Należy scalić posortowane podlisty. Ten etap wymaga, aby istniało co najwyżej M – 1 posortowanych podlist, co ogranicza rozmiar R. Potrzebny jest jeden blok wejściowy na każdą posortowaną podlistę i jeden blok na dane wyjściowe. Wykorzystanie buforów pokazano na rysunku 15.10. Wskaźnik do każdego bloku wejściowego określa pierwszy z posortowanych elementów, którego nie przeniesiono jeszcze do danych wyjściowych. Posortowane podlisty scalane są w jedną posortowaną listę wszystkich rekordów w następujący sposób.

  (1) Znajdowanie najmniejszego klucza wśród pierwszych niewybranych elementów na wszystkich listach. Ponieważ porównywanie odbywa się w pamięci głównej, odpowiednie jest wyszukiwanie liniowe. Liczba instrukcji maszynowych jest proporcjonalna do liczby podlist. Jeśli jednak trzeba, można zastosować metodę opartą na kolejkach priorytetowych[2]. Czas znajdowania najmniejszego elementu jest wtedy proporcjonalny do logarytmu liczby podlist.

  (2) Przenoszenie najmniejszego elementu na pierwszą dostępną pozycję w bloku danych wyjściowych.

  (3) Jeśli blok danych wyjściowych jest pełny, należy zapisać go na dysku i ponownie zainicjować ten sam bufor w pamięci głównej dla następnego bloku wyjściowego.

  (4) Jeżeli blok, z którego pobrano najmniejszy element, jest już pusty, należy wczytać kolejny blok tej samej posortowanej podlisty do bufora zajmowanego przez opróżniony blok. Jeśli nie ma więcej bloków, bufor pozostaje pusty i przy dalszym wyszukiwaniu najmniejszego z pozostałych elementów nie należy uwzględniać elementów z tej listy.



15.10. *Uporządkowanie pamięci głównej dla wielościeżkowego scalania*

[2] Zobacz: A. V. Aho i J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, 1992.

Aby DWSPS działał, nie może istnieć więcej niż $M - 1$ podlist. Załóżmy, że R mieści się w B blokach. Ponieważ każda podlista składa się z M bloków, liczba podlist wynosi B/M. Dlatego konieczne jest, aby: $B/M \leq M - 1$ lub $B \leq M(M - 1)$ (około $B \leq M^2$).

Algorytm wymaga wczytania w pierwszym przebiegu B bloków i kolejnych B dyskowych operacji wejścia-wyjścia na zapis posortowanych podlist. Każda posortowana podlista jest ponownie wczytywana w drugim przebiegu, co daje w sumie $3B$ operacji. Jeśli — jak zwykle — pomijamy koszty zapisu wyniku na dysku (ponieważ wynik można przekazywać w potoku bez zapisywania na dysku), $3B$ to wszystkie operacje potrzebne dla operatora sortowania τ. Jeżeli jednak trzeba zapisać wynik na dysku, niezbędnych jest $4B$ operacji.

Przykład 15.5. Załóżmy, że bloki mają po 64 000 bajtów, a pamięć główna ma jeden gigabajt pojemności. M ma więc wartość 16 000. Dlatego relację mieszczącą się w B blokach można posortować, o ile B nie wynosi więcej niż $(16\,000)^2 = 2^{28}$. Ponieważ bloki mają rozmiar $64\,000 = 2^{16}$, relację można posortować, o ile ma nie więcej niż 2^{44} bajtów, czyli 16 terabajtów. ■

W przykładzie 15.5 pokazano, że nawet w przeciętnym komputerze DWSPS wystarcza do posortowania w dwóch przebiegach wszystkich relacji, oprócz niezwykle dużych. Jeśli jednak nawet relacja jest większa, ten sam pomysł można zastosować rekurencyjnie. Należy podzielić relację na porcje o wielkości $M(M - 1)$, wykorzystać DWSPS do posortowania każdej z nich, a następnie potraktować uzyskane posortowane listy jako podlisty w trzecim przebiegu. W podobny sposób rozwiązanie można rozwinąć do dowolnej liczby przebiegów.

15.4.2. Eliminowanie powtórzeń za pomocą sortowania

Aby wykonać operację $\delta(R)$ w dwóch przebiegach, należy posortować krotki z R w podlistach, tak jak w DWSPS. W drugim przebiegu dostępną pamięć główną trzeba wykorzystać na przechowywanie jednego bloku z każdej posortowanej podlisty i jednego bloku wyjściowego, tak jak w DWSPS. Jednak zamiast sortować dane w drugim przebiegu, należy wielokrotnie pobierać pierwszą (w kolejności sortowania) niesprawdzoną krotkę t z wszystkich posortowanych podlist. W danych wyjściowych trzeba zapisać jedną kopię t, a z bloków wejściowych usunąć wszystkie wystąpienia t. W ten sposób dane wyjściowe będą zawierać dokładnie jedną kopię każdej krotki z R (krotki te będą posortowane). Kiedy blok wyjściowy jest pełen (lub blok wejściowy jest pusty), zarządzanie buforami odbywa się tak jak w DWSPS.

Liczba dyskowych operacji wejścia-wyjścia w tym algorytmie (jak zawsze, z pominięciem obsługi danych wyjściowych) jest taka sama jak przy sortowaniu — $3B(R)$. Warto porównać tę liczbę z wartością $B(R)$ dla algorytmu jednoprzebiegowego z punktu 15.2.2. Jednak algorytm dwuprzebiegowy działa dla dużo większych plików niż algorytm jednoprzebiegowy. W DWSPS dla algorytmu dwuprzebiegowego obowiązuje wymóg $B \leq M^2$, natomiast dla algorytmu jednoprzebiegowego — $B \leq M$. Ujmijmy to inaczej — wyeliminowanie powtórzeń za pomocą algorytmu dwuprzebiegowego wymaga tylko $\sqrt{B(R)}$ bloków pamięci głównej w porównaniu z $B(R)$ bloków niezbędnych w algorytmie jednoprzebiegowym.

15.4.3. Grupowanie i agregacja z wykorzystaniem sortowania

Algorytm dwuprzebiegowy dla operacji $\gamma_L(R)$ przypomina algorytm dla operacji $\delta(R)$ lub DWSPS. Oto jego krótki opis.

(1) Należy wczytać do pamięci krotki z R po M bloków na raz. Krotki trzeba posortować w każdym zbiorze M bloków, używając jako klucza sortowania atrybutów grupujących z L. Każda posortowana podlista zapisywana jest na dysku.
(2) Trzeba użyć jednego bufora pamięci głównej na każdą podlistę i początkowo wczytać do bufora pierwszy blok każdej podlisty.
(3) Należy wielokrotnie znajdować najmniejszą wartość klucza sortowania (atrybutów grupujących) wśród pierwszych dostępnych krotek z buforów. Wartość ta, v, staje się następną grupą, dla której:

   (a) następują przygotowania do obliczenia wszystkich agregacji z listy L tej grupy; tak jak w punkcie 15.2.2, można użyć liczby elementów i sumy zamiast średniej,
   (b) sprawdzana jest każda krotka o kluczu sortowania v i następuje akumulacja wartości dla potrzebnych agregacji,
   (c) jeśli bufor staje się pusty, należy zastąpić go następnym blokiem z danej podlisty.

   Kiedy nie ma już więcej krotek o kluczu sortowania v, należy zwrócić krotkę składającą się z atrybutów grupujących z L i powiązanych wartości agregacji obliczonych dla grupy.

Dwuprzebiegowy algorytm dla δ, podobnie jak dla γ, wymaga $3B(R)$ dyskowych operacji wejścia-wyjścia i działa dla $B(R) \leq M^2$.

15.4.4. Algorytm obliczania sumy oparty na sortowaniu

Kiedy potrzebna jest suma wielozbiorów, jednoprzebiegowy algorytm z punktu 15.2.3 (w którym po prostu kopiowane są obie relacji) działa niezależnie od wielkości argumentów, dlatego nie trzeba rozważać dwuprzebiegowego algorytmu dla $\cup_W$. Jednak jednoprzebiegowy algorytm dla $\cup_Z$ funkcjonuje tylko wtedy, kiedy przynajmniej jedna relacja mieści się w pamięci głównej, dlatego trzeba poznać algorytm dwuprzebiegowy dla sumy zbiorów. Opisana metoda działa dla części wspólnej i różnicy zbiorów oraz wielozbiorów, co opisano w punkcie 15.4.5. Aby obliczyć $R \cup_Z S$, należy zmodyfikować DWSPS w następujący sposób.

(1) W pierwszym kroku na podstawie R i S trzeba utworzyć posortowane podlisty.
(2) Należy użyć po jednym buforze pamięci głównej na każdą podlistę z R i S oraz zainicjować bufory pierwszym blokiem odpowiedniej podlisty.
(3) Trzeba wielokrotnie znajdować pierwszą krotkę t z wszystkich buforów, kopiować t do danych wyjściowych i usuwać z buforów wszystkie jej kopie (jeśli R i S to zbiory, istnieją najwyżej dwie kopie). Zarządzanie pustymi buforami wejściowymi i pełnym buforem wyjściowym odbywa się tak jak w DWSPS.

Zauważmy, że każda krotka z R i S jest wczytywana do pamięci głównej dwukrotnie — przy tworzeniu podlist i jako element jednej z nich. Krotka jest ponadto raz zapisywana na dysku jako część nowo utworzonej podlisty. Dlatego koszt dyskowych operacji wejścia-wyjścia wynosi $3(B(R) + B(S))$.

Algorytm działa, o ile łączna liczba podlist dla dwóch relacji nie przekracza $M - 1$, ponieważ potrzebny jest jeden bufor na każdą podlistę i jeden na dane wyjściowe. Dlatego suma rozmiarów obu relacji nie może przekraczać $M^2 — B(R) + B(S) \leq M^2$.

15.4.5. Obliczanie za pomocą sortowania części wspólnej i różnicy

Niezależnie od wersji (dla zbiorów lub wielozbiorów) algorytmy działają w zasadzie tak samo jak w punkcie 15.4.4, natomiast inaczej traktowane są kopie krotki t na początku posortowanych podlist. W każdym algorytmie używana jest krotka t najmniejsza (w kolejności sortowania) spośród wszystkich pozostałych w buforach wejściowych. Należy utworzyć dane wyjściowe w opisany sposób, a następnie usunąć wszystkie kopie t z buforów wejściowych.

- Dla części wspólnej zbiorów należy zwrócić t, jeśli występuje zarówno w R, jak i w S.
- Dla części wspólnej wielozbiorów należy zwrócić t tyle razy, ile razy minimalnie występuje w R i S. Zauważmy, że t nie jest zwracana, jeśli liczba ta wynosi 0 (czyli t nie występuje w jednej lub obu relacjach).
- Dla różnicy zbiorów ($R -_Z S$) należy zwrócić t wtedy i tylko wtedy, jeśli występuje w R, ale nie w S.
- Dla różnicy wielozbiorów ($R -_W S$) należy zwrócić t tyle razy, ile występuje w R, minus liczbę jej wystąpień w S. Oczywiście, jeśli t pojawia się w S przynajmniej tyle razy, co w R, w ogóle nie należy jej zwracać.

W operacjach na wielozbiorach trzeba pamiętać o pewnym drobiazgu. Przy liczeniu wystąpień t może się okazać, że wszystkie pozostałe krotki w buforze wejściowym to t. Wtedy w następnym bloku podlisty mogą pojawiać się dalsze t. Dlatego jeśli w buforze pozostają same t, trzeba wczytać kolejny blok podlisty i kontynuować liczenie t. Proces ten może obejmować kilka bloków i podlist.

Analizy dla tej rodziny algorytmów są takie same jak dla algorytmu obliczania sumy zbiorów opisanego w punkcie 15.4.4. Potrzeba:

- $3(B(R) + B(S))$ dyskowych operacji wejścia-wyjścia,
- w przybliżeniu $B(R) + B(S) \leq M2$, aby algorytm mógł działać.

15.4.6. Prosty algorytm złączenia oparty na sortowaniu

Istnieje kilka sposobów na wykorzystanie sortowania do złączenia dużych relacji. Przed analizą algorytmów zwrócimy uwagę na problem, który może wystąpić przy złączeniu, choć nie dotyczył wcześniejszych operacji dwuargumentowych. Przy złączeniu liczba krotek relacji, które mają tę samą wartość porównywanych atrybutów, a tym samym muszą jednocześnie znajdować się w pamięci głównej, może przekroczyć pojemność pamięci. Skrajny przypadek występuje, kiedy porównywane atrybuty mają tylko jedną wartość, dlatego każdą krotkę jednej relacji trzeba złączyć z wszystkimi krotkami drugiej. Wtedy nie ma innego wyboru, jak zastosowanie złączenia zagnieżdżonego dla dwóch zbiorów krotek o identycznych wartościach porównywanych atrybutów.

Aby tego uniknąć, można spróbować zmniejszyć wykorzystanie pamięci głównej przez inne elementy algorytmu i udostępnić dużą liczbę buforów na krotki o danej wartości porównywanych atrybutów. W tym punkcie opisano algorytm, który zapewnia największą liczbę buforów na takie krotki. W punkcie 15.4.8 omówiono inny algorytm oparty na sortowaniu, który wymaga mniej dyskowych operacji wejścia-wyjścia, ale może prowadzić do problemów przy dużej liczbie krotek o tej samej wartości porównywanych atrybutów.

Przy złączeniu relacji $R(X, Y)$ i $S(Y, Z)$ oraz M blokach pamięci głównej na bufory należy wykonać następujące kroki.

(1) Posortować R algorytmem DWSPS z Y jako kluczem sortującym.

(2) W podobny sposób posortować S.

(3) Scalić posortowane R i S. Wystarczą do tego dwa bufory — jeden na aktualny blok z R i drugi na aktualny blok z S. Poniższe kroki są wykonywane wielokrotnie.

   (a) Wyszukiwanie najmniejszej wartości y porównywanych atrybutów Y na początku bloków R i S.

   (b) Jeśli y nie pojawia się na początku drugiej relacji, należy usunąć krotki o kluczu sortowania y.

   (c) W przeciwnym razie w obu relacjach trzeba znaleźć wszystkie krotki o kluczu sortowania y. Jeśli trzeba, należy wczytywać bloki z posortowaną relacją R i (lub) S do momentu, kiedy wiadomo, że w żadnej relacji nie ma już y. Można do tego wykorzystać M buforów.

   (d) Zwracanie wszystkich krotek utworzonych przez złączenie krotek z R i S o identycznej wartości y atrybutów Y.

   (e) Jeśli w pamięci głównej nie ma niesprawdzonych krotek z którejś z relacji, należy dla tej relacji ponownie wczytać dane do bufora.

Przykład 15.6. Rozważmy relacje R i S z przykładu 15.4. Przypominamy, że relacje zajmują 1000 i 500 bloków, a liczba buforów w pamięci głównej to $M = 101$. Przy stosowaniu dla relacji algorytmu DWSPS i zapisywaniu wyniku na dysku potrzeba czterech dyskowych operacji wejścia-wyjścia na blok (po dwie na każdy etap). Dlatego sortowanie R i S wymaga $4(B(R) + B(S))$ (6000) dyskowych operacji wejścia-wyjścia.

Przy scalaniu posortowanych R i S w celu znalezienia złączonych krotek każdy blok z R i S wczytywany jest po raz piąty, co daje kolejnych 1500 operacji. Przy scalaniu zwykle potrzebne są tylko dwa spośród 101 bloków pamięci. Jeśli jednak trzeba, można wykorzystać wszystkie 101 bloków na krotki R i S o wspólnej wartości y atrybutów Y. Dlatego wystarczy, jeśli dla żadnego y krotki z R i S o tej wartości atrybutów Y nie zajmują razem więcej niż 101 bloków.

Zauważmy, że łączna liczba dyskowych operacji wejścia-wyjścia w tym algorytmie to 7500 (złączenie zagnieżdżone z przykładu 15.4 wymagało ich 5500). Jednak złączenie zagnieżdżone to z natury algorytm o złożoności kwadratowej, działający w czasie proporcjonalnym do $B(R)B(S)$, natomiast koszt operacji wejścia-wyjścia w złączeniu przez sortowanie rośnie liniowo, proporcjonalnie do $B(R) + B(S)$. Jedynie stałe czynniki i mały rozmiar przykładowych relacji (każda jest tylko 5 do 10 razy większa od pojemności przydzielonych buforów) sprawiają, że tu złączenie zagnieżdżone jest wydajniejsze. ■

15.4.7. Analiza prostego złączenia przez sortowanie

Jak wspomniano w przykładzie 15.6, algorytm z punktu 15.4.6 wykonuje pięć dyskowych operacji wejścia-wyjścia na każdy blok relacji podanych jako argumenty. Trzeba też ustalić, jak duże musi być M, aby proste złączenie przez sortowanie działało. Podstawowe ograniczenie dotyczy tego, że możliwe musi być przeprowadzenie dla relacji R i S dwuetapowego wielościeżkowego sortowania przez scalanie. W punkcie 15.4.1 stwierdzono, że wymaga to, aby $B(R) \leq M^2$ i $B(S) \leq M^2$. Ponadto konieczne jest, aby wszystkie krotki o tej samej wartości atrybutów Y mieściły się w M buforach. Podsumujmy.

- Proste złączenie przez sortowanie wymaga $5(B(R) + B(S))$ dyskowych operacji wejścia-wyjścia.
- Do działania konieczne jest, aby: $B(R) \leq M^2$ i $B(S) \leq M^2$.
- Krotki o tej samej wartości porównywanych atrybutów muszą mieścić się w M blokach.

15.4.8. Wydajniejsze złączenie przez sortowanie

Jeśli bardzo duża liczba krotek o identycznej wartości złączanych atrybutów nie stanowi zagrożenia, można zaoszczędzić dwie dyskowe operacje wejścia-wyjścia na blok przez połączenie drugiego etapu sortowania z samym złączeniem. Ten algorytm to *złączenie przez sortowanie* (ang. *sort join*; inne nazwy to złączenie przez scalanie lub złączenie przez sortowanie ze scalaniem). Aby obliczyć $R(X, Y) \bowtie S(Y, Z)$ z wykorzystaniem M buforów w pamięci głównej, należy wykonać poniższe kroki.

(1) Utworzyć posortowane podlisty o wielkości M przy użyciu Y jako klucza sortowania dla R i S.

(2) Przenieść pierwszy blok każdej podlisty do bufora. Zakładamy, że liczba podlist nie przekracza M.

(3) Wielokrotnie znaleźć najmniejszą wartość y atrybutów Y wśród pierwszych dostępnych krotek wszystkich podlist. Trzeba znaleźć wszystkie krotki z obu relacji o wartości y, na przykład używając niektórych z M dostępnych buforów na ich zapisanie, jeśli istnieje mniej niż M podlist. Należy zwrócić złączenie wszystkich krotek z R z wszystkimi krotkami z S mającymi tę samą wartość atrybutów Y. Jeśli bufor jednej z podlist zostanie opróżniony, trzeba usunąć go z dysku.

Przykład 15.7. Ponownie wróćmy do problemu z przykładu 15.4 — złączenia relacji R i S o rozmiarach 1000 oraz 500 bloków przy dostępnych 101 buforach. Należy podzielić R na 10, a S — na 5 podlist (każda o długości 100) i je posortować[3]. 15 buforów można wykorzystać na analizowane bloki każdej z podlist. Jeśli zdarzy się sytuacja, w której wiele krotek ma tę samą wartość atrybutów Y, można zapisać je w pozostałych 86 buforach.

Na każdy blok danych wykonywane są trzy dyskowe operacje wejścia-wyjścia. Dwie z nich służą do tworzenia posortowanych podlist. Następnie każdy blok każdej takiej podlisty należy ponownie wczytać do pamięci głównej w procesie wielościeżkowego scalania. Dlatego łączna liczba dyskowych operacji wejścia-wyjścia to 4500. ■

Ten algorytm złączenia przez sortowanie jest wydajniejszy od algorytmu z punktu 15.4.6, choć nie zawsze można z niego skorzystać. W przykładzie 15.7 stwierdzono, że liczba dyskowych operacji wejścia-wyjścia wynosi tu $3(B(R) + B(S))$. Algorytm można zastosować do danych niemal tak dużych, jak w poprzednim algorytmie. Rozmiar posortowanych podlist to M bloków, a na dwóch listach może być najwyżej M podlist. Dlatego wystarczający jest wymóg: $B(R) + B(S) \leq M^2$.

[3] Technicznie można uporządkować podlisty, aby miały długość po 101 bloków. Wtedy ostatnia podlista z R ma 91 bloków, a ostatnia podlista z S — 96 bloków. Jednak wtedy koszty są dokładnie takie same.

15.4.9. Omówienie algorytmów opartych na sortowaniu

W tabeli 15.11 zamieszczono analizy algorytmów omówionych w podrozdziale 15.4. W punktach 15.4.6 i 15.4.8 napisano, że w algorytmach złączenia obowiązują ograniczenia dotyczące liczby krotek o tej samej wartości porównywanych atrybutów. Przy naruszeniu ograniczenia konieczne może być zastosowanie złączenia zagnieżdżonego.

Operatory	Potrzebne M (w przybliżeniu)	Dyskowe operacje wejścia-wyjścia	Punkt
τ, γ, δ	$\sqrt{B}$	$3B$	15.4.1, 15.4.2, 15.4.3
$\cup, \cap, -$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.4, 15.4.5
$\bowtie$	$\sqrt{\max(B(R),B(S)}$	$5(B(R)+B(S))$	15.4.6
$\bowtie$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.8

15.11. *Wymogi związane z pamięcią główną i dyskowymi operacjami wejścia-wyjścia dla algorytmów opartych na sortowaniu*

15.4.10. Ćwiczenia do podrozdziału 15.4

Ćwiczenie 15.4.1. Dla każdej z podanych operacji napisz iterator, który wykorzystuje algorytm opisany w podrozdziale: a) eliminowanie powtórzeń (δ), b) grupowanie (γ_L), c) część wspólna zbiorów, d) różnica wielozbiorów, e) złączenie naturalne.

Ćwiczenie 15.4.2. Jeśli $B(R) = B(S) = 10\,000$ i $M = 1000$, jakie są wymogi dotyczące dyskowych operacji wejścia-wyjścia dla: a) sumy zbiorów, b) prostego złączenia przez sortowanie, c) wydajniejszego złączenia przez sortowanie opisanego w punkcie 15.4.8.

! **Ćwiczenie 15.4.3.** Załóżmy, że drugi przebieg algorytmu opisanego w podrozdziale nie wymaga wszystkich M buforów, ponieważ istnieje mniej niż M podlist. Jak można zmniejszyć liczbę dyskowych operacji wejścia-wyjścia, wykorzystując dodatkowe bufory?

! **Ćwiczenie 15.4.4.** W przykładzie 15.6 omówiono złączenie dwóch relacji R i S o 1000 bloków oraz 500 blokach dla $M = 101$. Jednak potrzebne są dodatkowe dyskowe operacje wejścia-wyjścia, jeśli istnieje tyle krotek o danej wartości, że krotki z żadnej z relacji nie mieszczą się w pamięci głównej. Określ łączną liczbę operacji potrzebnych, jeśli:

a) istnieją tylko dwie wartości atrybutów Y, a każda pojawia się w połowie krotek z R i w połowie krotek z S (Y to atrybut lub atrybuty, na których oparte jest złączenie),
b) istnieje pięć wartości atrybutów Y, a każda występuje w obu relacjach równie często,
c) istnieje 10 wartości atrybutów Y, a każda występuje w obu relacjach równie często.

! **Ćwiczenie 15.4.5.** Wykonaj ćwiczenie 15.4.4 dla wydajniejszego złączenia przez sortowanie, opisanego w punkcie 15.4.8.

Ćwiczenie 15.4.6. Ile pamięci potrzeba, aby można zastosować dwuprzebiegowy oparty na sortowaniu algorytm dla relacji obejmujących po 10 000 bloków, jeśli operacją jest: a) δ, b) γ, c) operacja dwuargumentowa, na przykład złączenie lub suma.

Ćwiczenie 15.4.7. Opisz dwuprzebiegowy oparty na sortowaniu algorytm dla każdego z operatorów z rodziny złączeń wymienionego w ćwiczeniu 15.2.4.

! **Ćwiczenie 15.4.8.** Załóżmy, że rekordy mogą być większe niż bloki (występują rekordy dzielone). Jak zmienia to wymogi dotyczące pamięci dla dwuprzebiegowych algorytmów opartych na sortowaniu?

!! **Ćwiczenie 15.4.9.** Czasem można zaoszczędzić kilka dyskowych operacji wejścia-wyjścia, zostawiając ostatnią podlistę w pamięci. W celu wykorzystania tego podejścia sensowne może być nawet zastosowanie podlist o mniej niż M blokach. Ile dyskowych operacji wejścia-wyjścia można zaoszczędzić w ten sposób?

15.5. Dwuprzebiegowe algorytmy oparte na haszowaniu

Istnieje rodzina opartych na haszowaniu algorytmów, które rozwiązują problemy omówione w podrozdziale 15.4. Wszystkie te algorytmy oparte są na podstawowym pomyśle — jeśli dane są za duże, aby zmieścić je w buforach pamięci głównej, należy utworzyć skrót każdej krotki z argumentu (lub argumentów), używając odpowiedniego klucza haszującego. We wszystkich standardowych operacjach istnieje sposób na taki dobór klucza haszującego, aby wszystkie krotki, które trzeba przetwarzać wspólnie, trafiały do tego samego kubełka.

Następnie operacje wykonywane są kubełek po kubełku (lub na parach kubełków o tej samej wartości skrótu w operacjach dwuargumentowych). W efekcie wielkość operandów zmniejsza się tyle razy, ile jest kubełków (czyli mniej więcej M razy). Zauważmy, że w opisanych w podrozdziale 15.4 algorytmach opartych na sortowaniu można osiągnąć podobne korzyści przez wstępne przetwarzanie, choć w podejściach związanych z sortowaniem i haszowaniem służą do tego różne środki.

15.5.1. Podział relacji przez haszowanie

Zacznijmy od omówienia podziału (z wykorzystaniem M buforów) relacji R na $M - 1$ kubełków o mniej więcej równej wielkości. Funkcja h to funkcja haszująca, przyjmująca jako argument kompletne krotki z R (wszystkie atrybuty z R są częścią klucza haszującego). Każdy kubełek wiązany jest z jednym buforem. Ostatni bufor przechowuje kolejne bloki z R. Należy obliczyć skrót $h(t)$ dla każdej krotki t z tego bloku i skopiować ją do odpowiedniego bufora. Jeśli bufor jest pełny, należy zapisać go na dysku i dla tego samego kubełka zainicjować inny blok. Ostatecznie zapisywany jest ostatni blok każdego kubełka (jeśli nie jest pusty). Bardziej szczegółowy opis algorytmu znajduje się na listingu 15.12.

```
zainicjuj M-1 kubełków za pomocą M-1 pustych buforów;
FOR każdego bloku b z relacji R DO BEGIN
   wczytaj blok b do M-tego bufora;
   FOR każdej krotki t z b DO BEGIN
      IF w buforze dla kubełka h(t) nie ma miejsca na t THEN
         BEGIN
            skopiuj bufor na dysk;
            zainicjuj nowy pusty blok w danym buforze;
         END;
      skopiuj t do bufora dla kubełka h(t);
   END;
END;
FOR każdego kubełka DO
   IF bufor dla kubełka nie jest pusty THEN
      zapisz bufor na dysku;
```

15.12. *Podział relacji R na M – 1 kubełków*

15.5.2. Algorytm usuwania powtórzeń oparty na haszowaniu

Rozważmy teraz szczegóły opartych na haszowaniu algorytmów dla różnych operacji algebry relacji, które mogą wymagać algorytmów dwuprzebiegowych. Po pierwsze, rozważmy usuwanie powtórzeń — $\delta(R)$. Za pomocą haszowania należy podzielić R na $M - 1$ kubełków, tak jak na listingu 15.12. Zauważmy, że dwie kopie tej samej krotki t trafiają do tego samego kubełka. Dlatego można przetwarzać kubełek po kubełku, wykonywać δ na każdym kubełku osobno i obliczyć odpowiedź jako sumę $\delta(R_i)$, gdzie R_i to część relacji R z i-tego kubełka. Jednoprzebiegowy algorytm z punktu 15.2.2 można zastosować do usunięcia powtórzeń kolejno z każdego R_i i zapisania uzyskanych niepowtarzalnych krotek.

Ta metoda działa, o ile poszczególne R_i są na tyle małe, że mieszczą się w pamięci głównej, co pozwala zastosować algorytm jednoprzebiegowy. Ponieważ można przyjąć, że funkcja haszująca h dzieli R na kubełki o równym rozmiarze, każda R_i zajmuje około $B(R)/(M - 1)$ bloków. Jeśli liczba bloków jest nie większa niż M ($B(R) \leq M(M - 1)$), można użyć dwuprzebiegowego algorytmu opartego na haszowaniu. Jak opisano to w punkcie 15.2.2, konieczne jest tylko, aby liczba różnych krotek w jednym kubełku mieściła się w M buforach. Dlatego według ostrożnych szacunków (przy założeniu, że M i $M - 1$ to prawie to samo) wymagane jest, aby $B(R) \leq M^2$, dokładnie tak, jak w opartym na sortowaniu dwuprzebiegowym algorytmie dla δ.

Liczba dyskowych operacji wejścia-wyjścia jest także podobna do tej w algorytmie opartym na sortowaniu. Każdy blok z R wczytywany jest raz przy haszowaniu krotek, a ponadto każdy blok wszystkich kubełków zapisywany jest na dysku. Następnie należy ponownie wczytać każdy blok wszystkich kubełków w jednoprzebiegowym algorytmie działającym dla danego kubełka. Dlatego łączna liczba dyskowych operacji wejścia-wyjścia to $3B(R)$.

15.5.3. Grupowanie i agregacja oparte na haszowaniu

Aby wykonać operację $\gamma_L(R)$, należy ponownie zacząć od haszowania wszystkich krotek z R w celu zapisania ich w $M - 1$ kubełkach. Jednak żeby się upewnić, że wszystkie krotki z tej samej grupy znajdą się w jednym kubełku, trzeba wybrać funkcję haszującą opartą wyłącznie na atrybutach grupujących z listy L.

Po podzieleniu R na kubełki można wykorzystać jednoprzebiegowy algorytm dla γ (opisany w punkcie 15.2.2) do przetworzenia po kolei każdego kubełka. Jak wyjaśniono w kontekście δ w punkcie 15.5.2, każdy kubełek można przetworzyć w pamięci głównej, pod warunkiem, że $B(R) \leq M^2$.

Jednak w drugim przebiegu przy przetwarzaniu każdego kubełka potrzebny jest tylko jeden rekord na grupę. Dlatego jeśli nawet rozmiar kubełka jest większy niż M, można obsłużyć kubełek w jednym przebiegu, pod warunkiem że rekordy dla wszystkich grup z kubełka zajmują nie więcej niż M buforów. Tak więc dla dużych grup można obsłużyć znacznie większe relacje R, niż określa to reguła $B(R) \leq M^2$. Jeśli jednak M jest większe niż liczba grup, nie można zapełnić wszystkich kubełków. Dlatego ograniczenie rozmiaru R za pomocą funkcji od M jest skomplikowane. Według ostrożnych szacunków ograniczenie to $B(R) \leq M^2$. Zauważmy też, że liczba dyskowych operacji wejścia-wyjścia dla γ, podobnie jak dla δ, to $3B(R)$.

15.5.4. Suma, część wspólna i różnica oparte na haszowaniu

W operacjach dwuargumentowych trzeba zastosować tę samą funkcję haszująca dla krotek z obu argumentów. Aby na przykład obliczyć $R \cup_Z S$, należy umieścić każdą z relacji R i S w $M - 1$ kubełkach (na przykład w $R_1, R_2, \ldots, R_{M-1}$ i $S_1, S_2, \ldots, S_{M-1}$). Następnie można obliczyć sumę zbiorów R_i i S_i dla wszystkich i oraz zwrócić wynik. Zauważmy, że jeśli krotka t występuje w R i S, dla pewnego i znajduje się zarówno w R_i, jak i w S_i. Dlatego przy obliczaniu sumy dwóch kubełków należy zwrócić tylko jedną kopię t i nie ma możliwości, że w wyniku pojawią się powtórzenia. Dla operacji $\cup_W$ zalecany jest prosty algorytm obliczania sumy wielozbiorów (punkt 15.2.3).

Aby obliczyć część wspólną lub różnicę R i S, należy — tak jak dla sumy zbiorów — utworzyć $2(M - 1)$ kubełków i zastosować odpowiedni algorytm jednoprzebiegowy do każdej pary powiązanych kubełków. Zauważmy, że wszystkie algorytmy jednoprzebiegowe wymagają $B(R) + B(S)$ dyskowych operacji wejścia-wyjścia. Do tej liczby trzeba dodać dwie dyskowe operacje wejścia-wyjścia na blok niezbędne do podziału przez haszowanie krotek z dwóch relacji i zapisania kubełków na dysku, co w sumie daje $3(B(R) + B(S))$ operacji.

Algorytmy wymagają do działania, aby w jednym przebiegu można było obliczyć sumę, część wspólną lub różnicę R_i i S_i, których rozmiary to w przybliżeniu $B(R)/(M - 1)$ oraz $B(S)/(M - 1)$. Przypominamy, że jednoprzebiegowe algorytmy dla tych operacji wymagają, żeby mniejszy operand mieścił się w $M - 1$ blokach. Dlatego dwuprzebiegowe algorytmy oparte na haszowaniu wymagają, aby — w przybliżeniu — $\min(B(R), B(S)) \leq M^2$.

15.5.5. Algorytm złączenia przez haszowanie

Aby obliczyć $R(X, Y) \bowtie S(Y, Z)$ za pomocą dwuprzebiegowego algorytmu opartego na haszowaniu, można postępować niemal tak samo jak przy innych operacjach dwuargumentowych, opisanych w punkcie 15.5.4. Jedyna różnica polega na tym, że jako klucza haszującego trzeba użyć tylko atrybutów uwzględnianych przy złączeniu (Y). Wiadomo wtedy, że złączane krotki R i S znajdują się w odpowiadających sobie kubełkach R_i i S_i dla pewnego i. Jednoprzebiegowe złączenie wszystkich par z powiązanych kubełków kończy algorytm (nazywamy go *złączeniem przez haszowanie*[4]).

[4] Czasem nazwa złączenie przez haszowanie jest zarezerwowana dla odmiany jednoprzebiegowego algorytmu złączenia z punktu 15.2.3, w którym wspomagającą wyszukiwanie strukturą w pamięci głównej jest tablica z haszowaniem. Wtedy opisany tu algorytm dwuprzebiegowy nazywany jest złączeniem przez haszowanie z podziałem.

Przykład 15.8. Przypominamy dwie relacje R i S z przykładu 15.4, których wielkość to 1000 i 500 bloków. W pamięci głównej dostępnych jest 101 buforów. Za pomocą haszowania można zapisać każdą relację w 100 kubełkach, tak że średnia wielkość kubełków to 10 (dla R) lub 5 bloków (dla S). Ponieważ mniejsza liczba, 5, jest znacznie niższa niż liczba dostępnych buforów, można oczekiwać, że jednoprzebiegowe złączenie każdej pary kubełków nie sprawi problemów.

Liczba dyskowych operacji wejścia-wyjścia wynosi 1500 przy wczytywaniu R i S w czasie haszowania, kolejne 1500 operacji potrzeba na zapis wszystkich kubełków na dysku, a trzecie 1500 operacji zajmuje ponowne wczytanie każdej pary kubełków do pamięci głównej w czasie jednoprzebiegowego złączenia powiązanych kubełków. Tak więc liczba potrzebnych operacji to 4500, tak jak dla wydajnego złączenia przez sortowanie z punktu 15.4.8. ■

Przykład 15.8 można uogólnić, stwierdzając, że:

- złączenie przez haszowanie wymaga $3(B(R) + B(S))$ dyskowych operacji wejścia-wyjścia,
- dwuprzebiegowy algorytm złączenia przez haszowanie działa, jeśli (w przybliżeniu) $\min(B(R), B(S)) \leq M^2$.

Ostatni punkt wynika z tego, co w innych operacjach dwuargumentowych — jeden kubełek z każdej pary musi mieścić się w $M - 1$ buforach.

15.5.6. Zmniejszanie liczby dyskowych operacji wejścia-wyjścia

Jeśli w pierwszym przebiegu mamy dostęp do większej ilości pamięci niż potrzeba na przechowywanie jednego bloku na kubełek, można zmniejszyć liczbę dyskowych operacji wejścia-wyjścia. Jedną z możliwości jest zastosowanie kilku bloków dla każdego kubełka i zapisanie ich jako grupy w przyległych blokach na dysku. Ujmijmy to ściśle — technika ta nie zmniejsza liczby operacji, ale je przyspiesza, ponieważ przy zapisie można pominąć czas wyszukiwania i opóźnienie obrotowe.

Istnieje jednak kilka sztuczek stosowanych w celu uniknięcia zapisu niektórych kubełków na dysku i ponownego ich odczytu. Tu opisano najwydajniejszą technikę, *hybrydowe złączenie przez haszowanie*. Przyjmijmy, że w celu obliczenia złączenia $R \bowtie S$ (S to mniejsza relacja) trzeba utworzyć k kubełków, co wymaga znacznie mniej niż M pamięci (jest to ilość dostępnej pamięci). W czasie haszowania S można zdecydować się na przechowywanie m spośród k kubełków w całości w pamięci głównej i zachować tylko jeden blok na każdy z pozostałych $k - m$ kubełków. Jest to możliwe, pod warunkiem że oczekiwany rozmiar kubełków w pamięci plus jeden blok na każdy z pozostałych kubełków nie przekracza M, czyli:

$$mB(S)/k + k - m \leq M \tag{15.1}$$

Oto wyjaśnienie — oczekiwany rozmiar kubełka to $B(S)/k$, a w pamięci znajduje się m kubełków.

Teraz, przy wczytywaniu krotek z drugiej relacji (R) w celu jej podziału na kubełki w pamięci przechowywane są:

(1) m niezapisanych na dysku kubełków z S

oraz

(2) jeden blok dla każdego z $k - m$ kubełków z R, dla których powiązane kubełki z S zapisano na dysku.

Jeśli krotka t z R trafia w wyniku haszowania do jednego z pierwszych m kubełków, można natychmiast złączyć ją z wszystkimi krotkami z powiązanego kubełka dla S, tak jak przy jednoprzebiegowym złączeniu przez haszowanie. Do usprawnienia złączenia niezbędne jest uporządkowanie każdego z dostępnych w pamięci kubełków z S w wydajną strukturę wspomagającą wyszukiwanie, tak jak przy złączeniu jednoprzebiegowym. Jeśli t trafia do jednego z kubełków, dla których powiązany kubełek z S znajduje się na dysku, t należy zapisać w bloku kubełka w pamięci głównej, a następnie na dysku, tak jak dla dwuprzebiegowego złączenia opartego na haszowaniu.

W drugim przebiegu powiązane kubełki z R i S można złączyć w standardowy sposób. Nie trzeba jednak złączać par kubełków, dla których kubełek z S znajdował się w pamięci. Algorytm już złączył te kubełki i zwrócił wyniki.

Oszczędność dyskowych operacji wejścia-wyjścia to dwa na każdy pozostawiony w pamięci blok kubełków z S i powiązanych kubełków z R. Ponieważ w pamięci znajduje się m/k kubełków, oszczędności wynoszą $2(m/k)(B(R) + B(S))$. Trzeba więc ustalić, jak zmaksymalizować wartość m/k z uwzględnieniem ograniczeń z równania (15.1). Zaskakująca odpowiedź jest taka: należy wybrać $m = 1$, a następnie możliwie najmniejsze k.

Oto intuicyjne uzasadnienie — wszystkie oprócz $k - m$ buforów z pamięci głównej można wykorzystać do przechowywania krotek z S w tej pamięci. Im więcej jest tych krotek, tym mniej trzeba dyskowych operacji wejścia-wyjścia. Dlatego warto zminimalizować k, czyli łączną liczbę kubełków. W tym celu należy utworzyć kubełki tak duże, że ledwo mieszczą się w pamięci głównej. Kubełki mają mieć rozmiar M, a tym samym $k = B(S)/M$. Wtedy w dodatkowej pamięci głównej znajduje się miejsce na tylko jeden kubełek ($m = 1$).

W praktyce kubełki muszą być nieco mniejsze niż M. W przeciwnym razie zabraknie miejsca na jeden kompletny kubełek i jeden blok dla pozostałych $k - 1$ kubełków przechowywanych w danym czasie w pamięci. Jeśli dla uproszczenia założymy, że k to około $B(S)/M$, a $m = 1$, wtedy liczba dyskowych operacji wejścia-wyjścia będzie mniejsza o:

$$2M(B(R) + B(S))/B(S)$$

Łączny koszt wynosi $(3 - 2M/B(S))(B(R) + B(S))$.

Przykład 15.9. Rozważmy problem z przykładu 15.4. Trzeba złączyć relacje R i S o 1000 bloków oraz 500 blokach, używając $M = 101$. Przy stosowaniu hybrydowego złączenia przez haszowanie k (liczba kubełków) powinna wynosić około 500/101. Załóżmy, że $k = 5$. Wtedy kubełek ma średnio 100 bloków z krotkami z S. Przy próbie umieszczenia jednego z takich kubełków i czterech dodatkowych bloków na cztery pozostałe kubełki potrzeba 104 bloków pamięci głównej, a nie można ryzykować, że przechowywany w pamięci kubełek się w niej nie zmieści.

Dlatego lepiej wybrać $k = 6$. Teraz przy haszowaniu S w pierwszym przebiegu dostępnych jest pięć buforów na pięć kubełków i do 96 buforów na kubełek przechowywany w pamięci, którego oczekiwany rozmiar to 500/6, czyli 83. Liczba dyskowych operacji wejścia-wyjścia używanych dla S w pierwszym przebiegu to 500 na wczytanie całej S i 500 – 83 = 417 na zapis pięciu kubełków na dysku. Przy przetwarzaniu R w pierwszym przebiegu trzeba wczytać całą R (1000 dyskowych operacji wejścia-wyjścia) i zapisać 5 z 6 kubełków (833 operacje).

W drugim przebiegu wczytywane są wszystkie kubełki zapisane na dysku, co daje 417 + 833 = 1250 dodatkowych operacji. Łączna liczba dyskowych operacji wejścia-wyjścia wynosi więc 1500 na wczytanie R i S, 1250 na wczytanie 5 z 6 relacji i kolejne 1250 na ponowne wczytanie krotek, co daje 4000 operacji. Można porównać tę liczbę z 4500 dyskowych operacji wejścia-wyjścia potrzebnych przy prostym złączeniu przez haszowanie lub złączeniu przez sortowanie. ■

15.5.7. Przegląd algorytmów opartych na haszowaniu

W tabeli 15.13 przedstawiono wymogi dotyczące pamięci i dyskowych operacji wejścia-wyjścia dla każdego z algorytmów omówionych w podrozdziale. Podobnie jak w innych rodzajach algorytmów, szacunki dla γ i δ są ostrożne, ponieważ operacje te zależą od liczby powtórzeń lub grup, a nie od liczby krotek w relacji podanej jako argument.

Operatory	Potrzebne M (w przybliżeniu)	Dyskowe operacje wejścia-wyjścia	Punkt
γ, δ	$\sqrt{B}$	$3B$	15.5.2, 15.5.3
$\cup, \cap, -$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	15.5.4
$\bowtie$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	15.5.5
$\bowtie$	$\sqrt{B(S)}$	$(3 - 2M/B(S))(B(R) + B(S))$	15.5.6

15.13. *Wymogi dotyczące pamięci głównej i dyskowych operacji wejścia-wyjścia w algorytmach opartych na haszowaniu; dla operacji binarnych zakładamy, że* $B(S) \leq B(R)$

Zauważmy, że wymogi dla algorytmów opartych na sortowaniu i odpowiadających im algorytmów opartych na haszowaniu są niemal identyczne. Oto istotne różnice między dwoma podejściami.

(1) Wymogi dotyczące rozmiaru w opartych na haszowaniu algorytmach dla operacji dwuargumentowych zależą tylko od mniejszego z dwóch argumentów, a nie — tak jak w algorytmach opartych na sortowaniu — od sumy ich rozmiarów.

(2) Algorytmy oparte na sortowaniu czasem umożliwiają utworzenie wyniku w posortowanej postaci i późniejsze wykorzystanie tego faktu. Wyniku można użyć w innym opartym na sortowaniu algorytmie dla dalszego operatora lub podać go jako odpowiedź na zapytanie, jeśli dane mają być posortowane.

(3) Algorytmy oparte na haszowaniu wymagają, aby kubełki miały równy rozmiar. Ponieważ zwykle występują przynajmniej niewielkie różnice w wielkości, nie można wykorzystać kubełków, które średnio zajmują M bloków. Trzeba ograniczyć rozmiar do nieco mniejszej wartości. Jest to szczególnie odczuwalne, jeśli liczba różnych kluczy haszujących jest niewielka (przy małej liczbie grup w czasie grupowania lub niewielkiej liczbie wartości atrybutów uwzględnianych przy złączeniu).

(4) W algorytmach opartych na sortowaniu posortowane podlisty można zapisać w przyległych blokach dysku (jeśli dysk jest odpowiednio uporządkowany). Dlatego jedna na trzy dyskowe operacje wejścia-wyjścia na blok może wiązać się z małym opóźnieniem obrotowym lub czasem wyszukiwania i przebiegać dużo szybciej niż takie operacje w algorytmach opartych na haszowaniu.

(5) Ponadto jeśli wartość M jest znacznie większa niż liczba posortowanych podlist, można jednocześnie wczytać kilka przyległych bloków posortowanej podlisty, co także pozwala skrócić opóźnienie i czas wyszukiwania.

(6) Jeśli jednak w algorytmie opartym na haszowaniu wystarczy utworzyć mniej niż M kubełków, można jednocześnie zapisać kilka bloków z kubełka. Zapewnia to na etapie zapisywania przy haszowaniu te same korzyści, które uzyskujemy przy drugim odczycie w algorytmach opartych na sortowaniu, co opisano w punkcie 5. Także tu można uporządkować dysk w taki sposób, aby zapisać kubełek w kolejnych blokach ścieżki. Wtedy kubełki można wczytywać z niewielkim opóźnieniem i czasem wyszukiwania — tak jak w punkcie 4. możliwy był wydajny zapis posortowanych podlist.

15.5.8. Ćwiczenia do podrozdziału 15.5

Ćwiczenie 15.5.1. Pomysł wykorzystany w hybrydowym złączeniu przez haszowanie, zapisywanie jednego kubełka w pamięci głównej, można zastosować także do innych operacji. Pokaż, jak zmniejszyć koszt o zapisywanie i wczytywanie jednego kubełka z każdej relacji w opartym na haszowaniu dwuprzebiegowym algorytmie dla operacji: a) δ, b) γ, c) $\cap_W$, d) $-_Z$.

Ćwiczenie 15.5.2. Jeśli $B(S) = B(R) = 10\,000$, a $M = 1000$, ile dyskowych operacji wejścia-wyjścia trzeba na hybrydowe złączenie przez haszowanie?

Ćwiczenie 15.5.3. Napisz iteratory będące implementacją opartych na haszowaniu dwuprzebiegowych algorytmów dla operacji: a) δ, b) γ, c) $\cap_W$, d) $-_Z$, e) $\rhd\!\lhd$.

! **Ćwiczenie 15.5.4.** Wykonywana jest oparta na haszowaniu dwuprzebiegowa operacja grupowania na relacji R o odpowiednim rozmiarze — $B(R) \leq M^2$. Istnieje jednak tak mało grup, że niektóre z nich są większe niż M (nie można zapisać ich w całości w pamięci głównej). Jakie modyfikacje, jeśli w ogóle, są potrzebne w przedstawionym algorytmie?

! **Ćwiczenie 15.5.5.** Załóżmy, że zastosowano dysk, w którym czas do przeniesienia głowicy nad blok wynosi 100 milisekund, a odczyt jednego bloku trwa ½ milisekundy. Tak więc odczyt k kolejnych bloków po umiejscowieniu głowicy zajmuje $k/2$ milisekund. Wykonujemy dwuprzebiegowe złączenie przez haszowanie $R \rhd\!\lhd S$, gdzie $B(R) = 1000$, $B(S) = 500$, a $M = 101$. Aby przyspieszyć złączenie, należy użyć jak najmniejszej liczby kubełków (zakładamy, że rozkład krotek w kubełkach jest równomierny) oraz wczytywać i zapisywać możliwie dużo bloków na kolejnych pozycjach dysku. Liczymy 100,5 milisekundy na losową dyskową operację wejścia-wyjścia i $100 + k/2$ milisekundy na odczyt (lub zapis) k kolejnych bloków z dysku (albo na dysk).

a) Ile czasu zajmuje dyskowa operacja wejścia-wyjścia?
b) Ile czasu zajmuje taka operacja w hybrydowym złączeniu przez haszowanie w wersji z przykładu 15.9?
c) Ile czasu w tych samych warunkach zajmuje złączenie oparte na sortowaniu przy założeniu, że posortowane podlisty zapisywane są w kolejnych blokach dysku?

15.6. Algorytmy oparte na indeksach

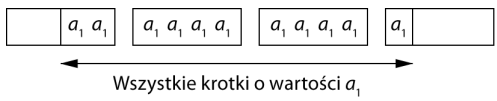
Istnienie indeksu na jednym lub kilku atrybutach relacji pozwala zastosować pewne algorytmy niedostępne bez indeksu. Algorytmy oparte na indeksach są szczególnie przydatne dla operatora selekcji, jednak można je z bardzo dobrymi efektami stosować także przy złączeniu i innych operacjach dwuargumentowych. W tym podrozdziale przedstawiono te algorytmy. Kontynuowane jest też rozpoczęte w punkcie 15.1.1 omówienie operatora skanowania indeksu, służącego do dostępu do tabel składowanych z indeksem. Aby zrozumieć wiele zagadnień, najpierw trzeba zapoznać się z dygresją i indeksami klastrującymi (ang. *clustering index*).

15.6.1. Indeksy klastrujące i nieklastrujące

W punkcie 15.1.3 stwierdzono, że relacja jest sklastrowana, jeśli krotki są upakowane w prawie minimalnej liczbie bloków potrzebnych do ich zapisania. We wszystkich dotychczasowych analizach zakładano, że relacje są sklastrowane.

Istnieją też *indeksy klastrujące*. Są to indeksy na atrybucie lub atrybutach, dzięki którym wszystkie krotki o danej wartości klucza wyszukiwania dla tego indeksu pojawiają się w prawie minimalnej liczbie bloków. Warto pamiętać, że relacja niesklastrowana nie może mieć indeksu klastrującego[5], jednak nawet relacja sklastrowana może mieć indeksy nieklastrujące.

Przykład 15.10. Relacja $R(a, b)$ posortowana według atrybutu a i upakowana w blokach w tej kolejności jest sklastrowana. Indeks na a jest klastrujący, ponieważ dla danej wartości a, a_1, wszystkie krotki o tej wartości znajdują się obok siebie. Dlatego występują upakowane w bloki, czasem z wyjątkiem pierwszego i ostatniego bloku zawierającego wartość a_1 atrybutu a, co pokazano na rysunku 15.14. Jednak indeks na b zwykle nie jest klastrujący, ponieważ krotki o danej wartości b są rozproszone po całym pliku (chyba że wartości a i b są bardzo ściśle skorelowane). ■



15.14. *Indeks klastrujący powoduje zapisanie wszystkich krotek o danej wartości w (prawie) minimalnej możliwej liczbie bloków*

15.6.2. Selekcja oparta na indeksie

W punkcie 15.1.1 omówiono implementację selekcji, $\sigma_C(R)$, przez wczytanie wszystkich krotek relacji R, sprawdzenie warunku C i zwrócenie tych krotek, które go spełniają. Jeśli nie istnieją indeksy dla R, jest to najlepsze rozwiązanie. Liczba dyskowych operacji wejścia-wyjścia wynosi $B(R)$ lub — jeśli R nie jest sklastrowana[6] — nawet $T(R)$ (jest to liczba krotek z R). Załóżmy jednak, że warunek C ma postać $a = v$, gdzie a to atrybut, na którym oparty jest indeks, a v to wartość. Można wtedy poszukać w indeksie wartości v i otrzymać wskaźniki do krotek z R o wartości v atrybutu a. Krotki te stanowią wynik operacji $\sigma_{a=v}(R)$, dlatego wystarczy je pobrać.

Jeśli indeks na $R.a$ jest klastrujący, liczba dyskowych operacji wejścia-wyjścia przy pobieraniu zbioru $\sigma_{a=v}(R)$ wynosi średnio $B(R)/V(R, a)$. Rzeczywista liczba z kilku przyczyn może być nieco wyższa.

(1) Często indeks nie jest w całości przechowywany w pamięci głównej, dlatego potrzebne są dyskowe operacje wejścia-wyjścia na wyszukiwanie indeksu.
(2) Choć wszystkie krotki o $a = v$ można zmieścić w b blokach, krotki mogą być zapisane w $b + 1$ blokach, ponieważ nie zaczynają się od początku bloku.

Jeśli nawet krotki z R są sklastrowane, czasem nie są upakowane w blokach możliwie ściśle. Dostępne może być na przykład dodatkowe miejsce na krotki później wstawiane do R. Relacja R może też znajdować się w sklastrowanym pliku, co omówiono w punkcie 14.1.6.

---

[5] Technicznie jeśli indeks jest oparty na kluczu relacji, istnieje tylko jedna krotka o danej wartości klucza indeksowania, dlatego indeks zawsze jest klastrujący, nawet wtedy, kiedy relacja nie jest sklastrowana. Jeżeli jednak istnieje tylko jedna krotka na wartość klucza indeksowania, klastrowanie nie daje korzyści, a wydajność indeksu jest taka sama jak indeksu nieklastrującego.

[6] Przypominamy notację opisaną w punkcie 15.1.3: $T(R)$ oznacza liczbę krotek z R, $B(R)$ to liczba bloków, w których mieści się R, a $V(R, L)$ określa liczbę różnych krotek w $_L(R)$.

Ponadto wynik trzeba zaokrąglić w górę, jeśli iloraz $B(R)/V(R, a)$ nie jest liczbą całkowitą. Najważniejsze jest to, że jeśli a to klucz R, wtedy $V(R, a) = T(R)$, co jest zwykle dużo większą wartością niż $B(R)$. Potrzebne są wtedy jedna dyskowa operacja wejścia-wyjścia na pobranie krotki o kluczu o wartości v plus inne takie operacje przy dostępie do indeksu.

Rozważmy teraz, co się dzieje, kiedy indeks na $R.a$ jest nieklastrujący. Oto pierwsze przybliżenie. Każda pobierana krotka znajduje się w innym bloku, dlatego trzeba uzyskać dostęp do $T(R)/V(R, a)$ krotek. Dlatego szacunkowo potrzeba $T(R)/V(R, a)$ dyskowych operacji wejścia-wyjścia. Wartość ta może być wyższa, jeśli trzeba wczytać z dysku bloki indeksu. Może też być niższa, jeżeli szczęśliwie niektóre pobrane krotki znajdują się w tym samym bloku, a sam blok pozostaje zbuforowany w pamięci.

Przykład 15.11. Załóżmy, że $B(R) = 1000$, a $T(R) = 20\,000$. Relacja R ma więc 20 000 krotek upakowanych maksymalnie po 20 na blok. Niech a będzie jednym z atrybutów relacji R. Przyjmijmy, że istnieje indeks na a i należy wykonać operację $\sigma_{a=0}(R)$. Oto kilka możliwych sytuacji i liczba dyskowych operacji wejścia-wyjścia potrzebnych w najgorszym przypadku. Koszt dostępu do bloków indeksu jest pomijany.

(1) Jeśli R jest sklastrowana, natomiast nie użyto indeksu, koszt wynosi 1000 dyskowych operacji wejścia-wyjścia. Trzeba pobrać każdy blok R.
(2) Jeżeli R nie jest sklastrowana, koszt to 20 000 dyskowych operacji wejścia-wyjścia.
(3) Jeśli $V(R, a) = 100$, a indeks jest klastrujący, algorytm oparty na indeksie wymaga $1000/100 = 10$ dyskowych operacji wejścia-wyjścia plus operacje potrzebne na dostęp do indeksu.
(4) Jeśli $V(R, a) = 10$, a indeks jest nieklastrujący, algorytm oparty na indeksie wymaga $20\,000/10 = 2000$ dyskowych operacji wejścia-wyjścia. Jest to koszt wyższy niż przy przeglądaniu całej relacji R, jeśli R jest sklastrowana, natomiast indeks nie istnieje.
(5) Jeśli $V(R, a) = 20\,000$, czyli a jest kluczem, algorytm oparty na indeksie wymaga 1 dyskowej operacji wejścia-wyjścia plus operacje potrzebne na dostęp do indeksu. Nie ma znaczenia, czy indeks jest klastrujący.

Przeglądanie indeksu jako metoda dostępu jest pomocne także w kilku innych rodzajach operacji selekcji.

a) Indeks w rodzaju drzewa zbalansowanego umożliwia wydajny dostęp do wartości klucza wyszukiwania z określonego przedziału. Jeśli istnieje indeks na atrybucie a relacji R, można wykorzystać go do pobrania krotek z odpowiedniego przedziału z R w operacjach selekcji podobnych do $\sigma_{a\geq 10}(R)$, a nawet $\sigma_{a\geq 10 \text{ AND } a\leq 20}(R)$.
b) Selekcję ze złożonym warunkiem C można czasem zaimplementować jako skanowanie indeksu, po którym następuje kolejna selekcja dotycząca tylko krotek zwróconych przy przeglądaniu indeksu. Jeśli C ma postać $a = v$ AND $C`$, gdzie $C`$ to dowolny warunek, można rozbić selekcję na dwie takie operacje. Pierwsza powinna sprawdzać tylko równość $a = v$, natomiast druga — warunek $C`$. W pierwszej można zastosować operator skanowania indeksu. Rozbicie operacji selekcji to jedno z wielu usprawnień, jakie optymalizator zapytań może wprowadzić w logicznym planie zapytania. Zagadnienie to opisano przede wszystkim w punkcie 16.7.1.

15.6.3. Złączenie za pomocą indeksu

We wszystkich rozważanych operacjach dwuargumentowych, a także w jednoargumentowych operacjach γ i δ na pełnych relacjach, można z powodzeniem korzystać z pewnych indeksów. Opracowanie większości algorytmów pozostawiamy jako ćwiczenia, natomiast tu koncentrujemy się na złączeniach. Szczególnie analizowane jest złączenie naturalne $R(X, Y) \bowtie S(Y, Z)$. Przypominamy, że X, Y i Z mogą oznaczać zbiory atrybutów, choć można też traktować je jak pojedyncze atrybuty.

W pierwszym algorytmie złączenia opartym na indeksie załóżmy, że w S istnieje indeks na atrybutach Y. Jednym ze sposobów na obliczenie złączenia jest pobranie każdego bloku z R i w ramach wszystkich bloków sprawdzenie każdej krotki t. Niech t_Y oznacza komponent lub komponenty t odpowiadające atrybutom Y. Za pomocą indeksu można znaleźć w S wszystkie krotki, w których t_Y to komponenty atrybutów Y. Te krotki S można złączyć z krotką t z R, dlatego należy zwrócić takie złączenia.

Liczba dyskowych operacji wejścia-wyjścia zależy od wielu czynników. Po pierwsze, jeśli R jest sklastrowana, należy wczytać $B(R)$ bloków w celu pobrania wszystkich krotek z R. Jeżeli R nie jest sklastrowana, potrzebnych może być do $T(R)$ dyskowych operacji wejścia-wyjścia.

Dla każdej krotki t z R trzeba wczytać średnio $T(S)/V(S,Y)$ krotek z S. Jeśli dla S istnieje niesklastrowany indeks na Y, liczba dyskowych operacji wejścia-wyjścia potrzebnych do wczytania S wynosi $T(R)T(S)/V(S,Y)$. Jeżeli jednak indeks jest sklastrowany, wystarczy $T(R)B(S)/V(S,Y)$ operacji[7]. W obu sytuacjach konieczne może być dodanie kilku dyskowych operacji wejścia-wyjścia na wartość Y, aby uwzględnić wczytywanie samego indeksu.

Niezależnie od tego, czy R jest sklastrowana, najistotniejszy jest koszt dostępu do krotek z S. Po pominięciu kosztu wczytywania R należy przyjąć $T(R)T(S)/V(S,Y)$ lub $T(R)(\max(1, B(S)/V(S,Y)))$ jako koszt dla tej metody złączenia dla przypadków z niesklastrowanym i sklastrowanym indeksem dla S.

Przykład 15.12. Wykorzystajmy podstawowy przykład — relacje $R(X, Y)$ i $S(Y, Z)$ obejmujące 1000 i 500 bloków. Załóżmy, że w bloku mieści się po 10 krotek z każdej relacji, zatem $T(R) = 10\,000$, a $T(S) = 5000$. Przyjmijmy też, iż $V(S, Y) = 100$ (w krotkach z S istnieje 100 różnych wartości Y).

Załóżmy, że R jest sklastrowana i dla S istnieje indeks klastrujący na Y. Wtedy przybliżona liczba dyskowych operacji wejścia-wyjścia, z pominięciem dostępu do samego indeksu, wynosi 1000 na odczyt bloków z R plus $10\,000 \times 500 / 100 = 50\,000$. Liczba ta jest wyraźnie wyższa niż dla opisanych wcześniej metod działających na tych samych danych. Jeśli R lub indeks dla S nie są sklastrowane, koszt jest jeszcze wyższy. ■

Choć na podstawie przykładu 15.12 może się wydawać, że złączenie oparte na indeksie to bardzo zły pomysł, istnieją inne sytuacje, w których złączenie $R \bowtie S$ ma większy sens. Najczęściej dzieje się tak, kiedy R jest bardzo mała w porównaniu z S, a wartość $V(S, Y)$ jest duża. W ćwiczeniu 15.6.5 opisano typowe zapytanie, w którym selekcja przed złączeniem sprawia, że R jest mała. Wtedy większa część S nie jest sprawdzana przez algorytm, ponieważ większość wartości Y w ogóle nie występuje w R. Z kolei w złączeniu opartym na sortowaniu i haszowaniu każda krotka z S sprawdzana jest przynajmniej raz.

[7] Trzeba jednak pamiętać, że jeśli $B(S)/V(S,Y)$ ma wartość poniżej 1, trzeba zastąpić ją liczbą 1, co opisano w punkcie 15.6.2.

15.6.4. Złączenia z wykorzystaniem posortowanego indeksu

Jeśli indeks jest drzewem zbalansowanym (lub inną strukturą, z której można łatwo pobrać posortowane krotki relacji), istnieje wiele innych możliwości zastosowania go. Prawdopodobnie najprostszą jest obliczanie $R(X, Y) \bowtie S(Y, Z)$, kiedy istnieje odpowiedni indeks na Y dla R lub S. Można wtedy zastosować zwykłe złączenie przez sortowanie, jednak nie trzeba wykonywać pośredniego kroku, polegającego na sortowaniu jednej z relacji według Y.

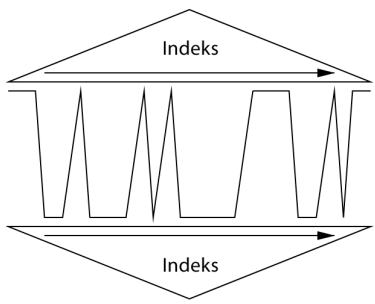
Skrajnym przypadkiem jest sytuacja, kiedy indeksy na Y istnieją zarówno dla R, jak i dla S. Wtedy trzeba wykonać tylko ostatni krok prostego złączenia opartego na sortowaniu (punkt 15.4.6). Metoda ta jest czasem nazywana *złączeniem zig-zag*, ponieważ polega na przechodzeniu między indeksami tam i z powrotem w celu znalezienia wspólnych wartości Y. Zauważmy, że nie trzeba pobierać krotek z R o wartości Y, która nie występuje w S (i na odwrót).

Przykład 15.13. Przyjmijmy, że istnieją relacje $R(X, Y)$ i $S(Y, Z)$ o indeksach na Y w obu relacjach. Niech klucze wyszukiwania (wartości Y) dla krotek z R mają wartości 1, 3, 4, 4, 4, 5, 6, a dla krotek z S — 2, 2, 4, 4, 6, 7. Zaczynamy od pierwszych kluczy z R i S, czyli 1 i 2. Ponieważ $1 < 2$, należy pominąć pierwszy klucz z R i sprawdzić drugi klucz, 3. Teraz obecny klucz z S jest mniejszy niż aktualny klucz z R, dlatego można pominąć dwie 2 z S i dojść do 4.

Na tym etapie klucz 3 z R ma wartość mniejszą niż klucz z S, dlatego można pominąć klucz z R. Teraz oba klucze mają wartość 4. Należy podążyć za wskaźnikami powiązanymi z wszystkimi kluczami o wartości 4 z obu relacji, pobrać odpowiednie krotki i złączyć je. Zauważmy, że do czasu natrafienia na wspólny klucz, 4, nie pobrano z relacji żadnych krotek.

Po wyczerpaniu wartości 4 należy przejść do klucza 5 w R i 6 w S. Ponieważ $5 < 6$, przechodzimy do następnego klucza w R. Teraz oba klucze mają wartość 6, dlatego należy pobrać powiązane krotki i je złączyć. Ponieważ pobrano całą R, wiadomo, że w relacjach nie ma więcej par krotek, które można by złączyć. ■

Jeśli indeksy to drzewa zbalansowane, można sprawdzić liście obu drzew w kolejności od lewej do prawej, używając wbudowanych w strukturę wskaźników do liści, co pokazano na rysunku 15.15. Jeśli R i S są sklastrowane, pobranie wszystkich krotek o danym kluczu wymaga bardzo małej liczby dyskowych operacji wejścia-wyjścia. Zauważmy, że w skrajnych sytuacjach, kiedy istnieje tyle krotek z R i S, iż nie mieszczą się one w pamięci głównej, trzeba zastosować poprawkę, taką jak omówiona w punkcie 15.4.6. Jednak w typowych przypadkach etap złączenia wszystkich krotek o identycznej wartości Y można wykonać za pomocą tylu dyskowych operacji wejścia-wyjścia, ile potrzeba na wczytanie krotek.



15.15. *Złączenie zig-zag z wykorzystaniem dwóch indeksów*

Przykład 15.14. Kontynuujmy przykład 15.12, aby zobaczyć, jaką wydajność dla przykładowych danych ma złączenie z wykorzystaniem sortowania i indeksów. Po pierwsze, załóżmy, że dla S istnieje indeks na Y umożliwiający pobranie krotek z S posortowanych według Y. W przykładzie przyjmujemy też, że obie relacje i indeks są sklastrowane. Na razie załóżmy, że nie ma indeksu dla R.

Jeśli dostępnych jest 101 bloków pamięci głównej, można wykorzystać je do utworzenia 10 posortowanych podlist dla 1000-blokowej relacji R. Liczba dyskowych operacji wejścia-wyjścia przy odczycie i zapisie całej R wynosi 2000. Następnie można użyć 11 bloków pamięci — 10 na podlisty z R i 1 na blok z krotkami z S pobranymi za pomocą indeksu. Pomijamy operacje i bufory pamięci potrzebne do manipulowania indeksem. Jeśli jednak indeks to drzewo zbalansowane, koszty i tak są niskie. W drugim przebiegu należy wczytać wszystkie krotki z R i S, co zajmuje w sumie 1500 dyskowych operacji wejścia-wyjścia plus mała liczba potrzebna na jednokrotne wczytanie bloków indeksu. Dlatego łączna liczba dyskowych operacji wejścia-wyjścia to 3500, czyli mniej niż w omawianych wcześniej metodach.

Przyjmijmy teraz, że w R i S istnieją indeksy na Y. Nie trzeba wtedy sortować żadnej relacji. Wystarczy 1500 dyskowych operacji wejścia-wyjścia na wczytanie bloków z R i S za pomocą indeksów. W praktyce ustalenie na podstawie samych indeksów, że dużej części krotek z R lub S nie można złączyć z krotkami drugiej relacji, pozwala zmniejszyć łączny koszt znacznie poniżej 1500 dyskowych operacji wejścia-wyjścia. Jednak zawsze należy dodać małą liczbę operacji potrzebnych na wczytanie samych indeksów. ■

15.6.5. Ćwiczenia do podrozdziału 15.6

Ćwiczenie 15.6.1. Załóżmy, że istnieje indeks na atrybucie $R.a$. Opisz, jak można wykorzystać ten indeks, aby usprawnić wykonywanie poniższych operacji. W jakich warunkach algorytm oparty na indeksie jest wydajniejszy od algorytmów opartych na sortowaniu lub haszowaniu?

a) $R \cup_Z S$ (przyjmij, że R i S nie zawierają powtórzeń, choć w obu może znajdować się identyczna krotka).
b) $R \cap_Z S$ (także tu R i S to zbiory).
c) $\delta(R)$.

Ćwiczenie 15.6.2. Załóżmy, że $B(R) = 10\,000$, $T(R) = 500\,000$, istnieje indeks na $R.a$, a $V(R, a) = k$ dla pewnego k. Określ koszt operacji $\sigma_{a=0}(R)$ jako funkcję od k w podanych dalej warunkach. Możesz pominąć dyskowe operacje wejścia-wyjścia potrzebne do uzyskania dostępu do samego indeksu.

a) Indeks jest klastrujący.
b) Indeks nie jest klastrujący.
c) R jest sklastrowane, natomiast indeks nie jest używany.

Ćwiczenie 15.6.3. Wykonaj ćwiczenie 15.6.2 dla zapytania zakresowego — $\sigma_{C \leq a \text{ AND } a \leq D}(R)$. Można przyjąć, że C i D to stałe, takie że w przedziale znajduje się $k/10$ wartości.

! **Ćwiczenie 15.6.4.** Jeśli R jest sklastrowana, natomiast indeks na $R.a$ nie jest klastrujący, w zależności od k korzystniejsze może być zaimplementowanie zapytania za pomocą skanowania tabeli R lub indeksu. Dla jakich wartości k lepszy będzie indeks, jeśli relacja i zapytanie są takie jak w: a) ćwiczeniu 15.6.2, b) ćwiczeniu 15.6.3.

Ćwiczenie 15.6.5. Rozważmy zapytanie SQL:

```
SELECT dataUrodzenia FROM GwiazdyWFilmie, GwiazdaFilmowa
WHERE tytulFilmu = 'King Kong' AND nazwiskoGwiazdy = nazwisko;
```

W zapytaniu wykorzystano „filmowe" relacje:

```
GwiazdyWFilmie(tytulFilmu, rokFilmu, nazwiskoGwiazdy)
GwiazdaFilmowa(nazwisko, adres, plec, dataUrodzenia)
```

Po przekształceniu zapytania na algebrę relacji kluczowe będzie złączenie równościowe pomiędzy:

$\sigma_{tytulFilmu='King\ Kong'}(\texttt{GwiazdyWFilmie})$

i relacją `GwiazdaFilmowa`, które można zaimplementować jako złączenie naturalne $R \bowtie S$. Ponieważ istnieją tylko trzy filmy o tytule *King Kong*, wartość $T(R)$ jest bardzo mała. Załóżmy, że dla S (relacja `GwiazdaFilmowa`) dostępny jest indeks na atrybucie `nazwisko`. Porównaj dla operacji $R \bowtie S$ koszt złączenia opartego na indeksie z kosztem złączenia opartego na sortowaniu lub haszowaniu.

! **Ćwiczenie 15.6.6.** W przykładzie 15.14 określono liczbę dyskowych operacji wejścia-wyjścia dla złączenia $R \bowtie S$, w którym dla R i (lub) S istnieją indeksy sortowania na atrybutach uwzględnianych w złączeniu. Jednak metody opisane w owym przykładzie nie zadziałają, jeśli istnieje zbyt wiele krotek o identycznych wartościach tych atrybutów. Poniżej jakiej liczby bloków zajmowanych przez krotki o tej samej wartości w opisanych metodach nie potrzeba dodatkowych dyskowych operacji wejścia-wyjścia?

15.7. Zarządzanie buforem

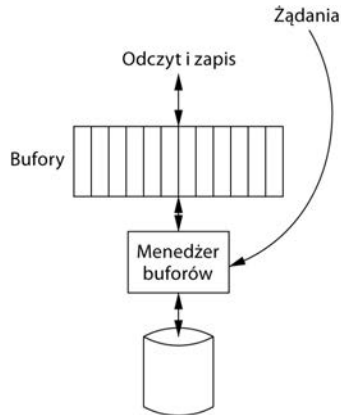
Przyjęto, że dla operatorów działających na relacjach dostępna jest w pamięci głównej pewna liczba M buforów, które można wykorzystać na potrzebne dane. W praktyce bufory rzadko są z góry przydzielane operatorowi, a wartość M może się zmieniać w zależności od warunków w systemie. Za główne zadanie, czyli udostępnianie buforów pamięci głównej procesom (na przykład zapytaniom) działającym na bazie danych, odpowiada *menedżer buforów*. Menedżer odpowiada za udostępnianie procesom potrzebnej pamięci i minimalizowanie przy tym opóźnienia oraz liczby żądań niemożliwych do zrealizowania. Zadania menedżera buforów przedstawiono na rysunku 15.16.

15.7.1. Architektura menedżera buforów

Istnieją dwie ogólne architektury menedżerów buforów.

(1) Menedżer buforów bezpośrednio kontroluje pamięć główną (tak działa wiele relacyjnych systemów DBMS).

(2) Menedżer buforów przydziela bufory w pamięci wirtualnej i umożliwia systemowi operacyjnemu określenie, które bufory znajdują się w danym momencie w pamięci głównej, a które pozostają w zarządzanej przez system przestrzeni wymiany na dysku. W ten sposób działa wiele systemów DBMS opartych na pamięci głównej i obiektowych.



15.16. *Menedżer buforów odpowiada na żądania dostępu do bloków dysku w pamięci głównej*

Niezależnie od podejścia zastosowanego w systemie DBMS powstaje ten sam problem: menedżer buforów powinien ograniczać liczbę używanych buforów, tak aby mieściły się w dostępnej pamięci głównej. Kiedy menedżer buforów bezpośrednio kontroluje pamięć główną, a zażądano zbyt dużej ilości pamięci, menedżer musi określić zwalniany bufor i zwrócić jego zawartość na dysk. Jeżeli blok w buforze się nie zmienił, wystarczy usunąć go z pamięci głównej, jednak po wprowadzeniu zmian blok trzeba z powrotem zapisać na jego miejscu na dysku. Jeśli menedżer buforów przedziela pamięć w pamięci wirtualnej, może przydzielić więcej buforów, niż mieści się w pamięci głównej. Jeżeli jednak rzeczywiście używane są wszystkie bufory, nastąpi przeładowanie — standardowy problem w systemie operacyjnym, kiedy to wiele bloków jest przenoszonych do i z przestrzeni wymiany dysku. W tej sytuacji system poświęca większość czasu na wymianę bloków i wykonuje bardzo niewiele użytecznych operacji.

Standardowo liczba buforów to parametr ustawiany w czasie inicjowania systemu DBMS. Można oczekiwać, że zgodnie z ustawieniami bufory będą zajmować dostępną pamięć główną niezależnie od tego, czy są alokowane w pamięci głównej czy wirtualnej. W dalszych punktach tryb buforowania nie ma znaczenia. Przyjęto, że istnieje *pula buforów* o stałym rozmiarze, czyli zestaw buforów dostępnych przy obsłudze zapytań i innych operacji na bazie danych.

Zarządzanie pamięcią przy przetwarzaniu zapytań

Zakładamy, że menedżer buforów przydziela operatorowi M buforów w pamięci głównej, przy czym wartość M zależy od warunków w systemie (w tym innych przetwarzanych operatorów i zapytań) oraz może zmieniać się dynamicznie. Kiedy operator ma dostęp do M buforów, może używać niektórych z nich do wczytywania stron dysku, innych na strony indeksu, a jeszcze innych — na etapy sortowania lub tablice haszujące. W niektórych systemach DBMS pamięć nie jest przydzielana z jednej puli, ale raczej z odrębnych pul pamięci (z różnymi menedżerami buforów) o odmiennym przeznaczeniu. Operator może otrzymać D buforów z puli na strony wczytane z dysku i H buforów na tablicę haszującą. To podejście zapewnia większe możliwości w zakresie konfigurowania i dostrajania systemu, jednak nie zawsze prowadzi do optymalnego globalnego wykorzystania pamięci.

15.7.2. Strategie zarządzania buforami

Krytyczny wybór dokonywany w menedżerze buforów dotyczy tego, który blok należy usunąć z puli buforów, kiedy potrzebny jest bufor na właśnie zażądany blok. Popularne *strategie wymiany buforów* mogą być znane z zastosowań reguł szeregowania w innym kontekście (na przykład w systemach operacyjnych). Oto niektóre strategie.

Algorytm LRU

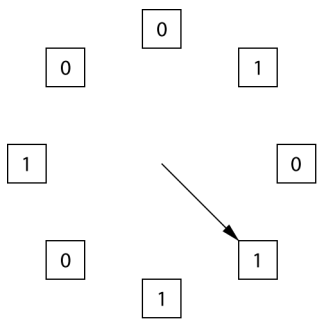
Reguła LRU (ang. *least-recently used*, czyli najdłużej nieużywany) polega na usuwaniu bloku, który nie był wczytywany ani zapisywany przez najdłuższy czas. Metoda ta wymaga, aby menedżer buforów przechowywał tabelę z czasem ostatniego dostępu do bloku w każdym buforze. Przy każdym dostępie do bazy danych trzeba też dodać wpis do tabeli, dlatego przechowywanie informacji jest pracochłonne. Jednak LRU to efektywna strategia. Intuicyjnie można stwierdzić, że długo nieużywane bufory prawdopodobnie będą potrzebne później od tych używanych niedawno.

Pierwszy na wejściu, pierwszy na wyjściu (FIFO)

Kiedy potrzebny jest bufor, a zastosowano strategię FIFO, bufor najdłużej zajmowany przez ten sam blok jest opróżniany, a następnie wykorzystywany na nowy blok. W tym podejściu menedżer buforów musi znać tylko czas wczytania do bufora obecnie zajmującego go bloku. Wpis w tabeli można więc dodać w czasie wczytywania bloku z dysku. Nie trzeba modyfikować tabeli przy dostępie do bloku. Strategia FIFO wymaga mniej pracy niż LRU, ale powoduje więcej błędów. Często używany blok, na przykład blok z korzeniem indeksu w postaci drzewa zbalansowanego, ostatecznie staje się najstarszym blokiem w buforze. Jest wtedy zapisywany z powrotem na dysku, po czym wkrótce trzeba wczytać go do innego bufora.

Algorytm „zegarowy" (drugiej szansy)

Algorytm ten to często stosowane, wydajne przybliżenie strategii LRU. Można wyobrazić sobie, że bufory uporządkowane są w okrąg, co pokazano na rysunku 15.17. „Wskazówka" pokazuje jeden z buforów i porusza się zgodnie z ruchem wskazówek zegara, kiedy trzeba znaleźć bufor, w którym należy umieścić blok z dysku. Każdy bufor jest powiązany z „flagą" o wartości 0 lub 1. Bufory z wartością 0 są narażone na przesłanie zawartości z powrotem na dysk; bufory z wartością 1 są na to odporne. Przy wczytywaniu bloku do bufora flagę należy ustawić na 1. Także dostęp do zawartości bufora powoduje ustawienie flagi na 1.



15.17. *Algorytm „zegarowy" przechodzi po kolejnych buforach i zastępuje flagi 01…1 flagami 10…0*

Inne sztuczki dotyczące algorytmu „zegarowego”

Algorytm „zegarowy” wyboru zwalnianych buforów nie ogranicza się do schematu opisanego w punkcie 15.7.2, gdzie flagi przyjmowały wartości 0 i 1. Do ważnej strony można początkowo przypisać flagę o wartości większej niż 1 i zmniejszać ją o 1 przy każdym przejściu wskazówki. W praktyce można zastosować przyklejanie bloków, przypisując do bloku flagę o nieskończonej wartości. W odpowiednim momencie system zwalnia blok, ustawiając flagę na 0.

Kiedy menedżer buforów potrzebuje bufor na nowy blok, szuka pierwszej wartości 0, poruszając się zgodnie z ruchem wskazówek zegara. Jeśli natrafia na flagę 1, ustawia ją na 0. Dlatego blok jest usuwany z bufora tylko wtedy, jeśli nie używano go przez czas potrzebny wskazówce na wykonanie pełnego obrotu i ustawienie flagi na 0, a następnie wykonanie kolejnego pełnego obrotu oraz znalezienie bufora z niezmodyfikowaną flagą 0. Przykładowo na rysunku 15.17 wskazówka spowoduje ustawienie flagi 1 na 0 w następnym buforze, a następnie przejdzie do bufora z flagą 0, zastąpi jego blok i ustawi flagę na 1.

Kontrola ze strony systemu

Procesor zapytań lub inne komponenty systemu DBMS mogą dawać menedżerowi buforów wskazówki pozwalające uniknąć pewnych błędów, które mogą wystąpić przy ścisłym przestrzeganiu algorytmów LRU, FIFO lub „zegarowego”. W punkcie 13.6.5 wspomniano, że czasem z przyczyn technicznych *nie można przenieść* bloku z pamięci głównej na dysk bez wcześniejszego zmodyfikowania pewnych innych prowadzących do niego bloków. Bloki te są przyklejone, a menedżer buforów musi zmodyfikować strategię wymiany buforów, aby uniknąć usunięcia takich bloków. Daje to możliwość wymuszenia pozostawienia pewnych bloków w pamięci głównej przez oznaczenie ich jako przyklejonych, nawet jeśli nie ma technicznych przyczyn, dla których nie można zapisać ich na dysku. Przykładowo rozwiązanie problemu z algorytmem FIFO (związanego z korzeniem drzewa zbalansowanego) polega na przyklejeniu korzenia, co wymusza stałe utrzymywanie go w pamięci. Podobnie w algorytmach w rodzaju jednoprzebiegowego złączenia przez haszowanie procesor zapytań może przykleić bloki mniejszej relacji, aby zagwarantować, że na stałe pozostaną w pamięci głównej.

15.7.3. Związki między fizycznym operatorem selekcji a zarządzaniem buforami

Optymalizator zapytań ostatecznie wybiera zbiór operatorów fizycznych używanych do wykonania danego zapytania. Wybór może być oparty na założeniu, że przy przetwarzaniu każdego operatora dostępnych jest M buforów. Jednak, jak pokazano, w menedżerze buforów nie zawsze pożądane lub możliwe jest zagwarantowanie dostępności M buforów na czas wykonywania zapytania. Dlatego trzeba zadać dwa powiązane pytania na temat operatorów fizycznych.

(1) Czy algorytm potrafi dostosować się do zmian w wartości M (liczbie buforów dostępnych w pamięci głównej)?

(2) Jeśli oczekiwana liczba M buforów nie jest dostępna, a niektóre bloki, które powinny znajdować się w pamięci, menedżer buforów przeniósł na dysk, w jaki sposób strategia wymiany buforów stosowana przez menedżera wpływa na liczbę dodatkowych potrzebnych operacji wejścia-wyjścia?

Przykład 15.15. Jako ilustracja tych problemów posłuży oparte na blokach złączenie zagnieżdżone z rysunku 15.8. Podstawowy algorytm nie zależy od wartości M, choć wpływa ona na jego wydajność. Dlatego M można ustalić tuż przed rozpoczęciem działania algorytmu.

Możliwe jest nawet to, że M zmieni się między iteracjami pętli zewnętrznej. Przy każdym wczytywaniu do pamięci głównej części relacji S (relacji z pętli zewnętrznej) można wykorzystać wszystkie dostępne w tym czasie bufory oprócz jednego. Ostatni bufor zarezerwowany jest na blok z R (relacji z pętli wewnętrznej). Tak więc liczba powtórzeń pętli zewnętrznej zależy od średniej liczby buforów dostępnych w każdej iteracji. Jednak o ile *średnio* dostępnych jest M buforów, analizy kosztów z punktu 15.3.4 są poprawne. W skrajnym przypadku można mieć szczęście i odkryć, że w pierwszej iteracji dostępne są bufory na zapisanie całej S. Wtedy złączenie zagnieżdżone działa tak jak złączenie jednoprzebiegowe, opisane w punkcie 15.2.3.

Oto inny przykład zależności buforowania i złączenia zagnieżdżonego. Załóżmy, że do wymiany buforów zastosowano strategię LRU, a na bloki z R dostępnych jest k buforów. Przy wczytywaniu kolejno każdego bloku z R pod koniec iteracji pętli zewnętrznej w buforach będzie znajdować się k ostatnich bloków z R. Następnie można wczytać do $M - 1$ buforów przeznaczonych na S nowe bloki z S i ponownie rozpocząć wczytywanie bloków z R (w następnej iteracji w pętli zewnętrznej). Jeśli jednak wczytywanie ponownie zacznie się od początku, trzeba wymienić k buforów przeznaczonych dla R. Samo $k > 1$ nie zapewnia wtedy zmniejszenia liczby dyskowych operacji wejścia-wyjścia.

Lepsza implementacja złączenia zagnieżdżonego (przy stosowaniu strategii LRU do wymiany buforów) polega na pobieraniu bloków z R w naprzemiennej kolejności: od początku do końca i od końca do początku (tak zwane *odbijanie*). W ten sposób przy k buforach dostępnych dla R w każdej (oprócz pierwszej) iteracji pętli zewnętrznej można zaoszczędzić k dyskowych operacji wejścia-wyjścia. Oznacza to, że druga i kolejne iteracje wymagają tylko $B(R) - k$ operacji dla R. Zauważmy, że nawet dla $k = 1$ (czyli bez *dodatkowych* buforów dostępnych dla R) oszczędność wynosi jedną operację na iterację. ■

Zmienna wartość M i strategia wymiany buforów używana w menedżerze buforów mają znaczenie także w innych algorytmach. Oto kilka przydatnych spostrzeżeń.

- Algorytm oparty na sortowaniu można dostosować do zmiany wartości M. Jeśli M maleje, można zmienić rozmiar podlist, ponieważ opisane algorytmy oparte na sortowaniu nie wymagają, aby podlisty miały stałą wielkość. Głównym ograniczeniem przy zmniejszeniu M jest to, że czasem trzeba utworzyć tyle podlist, że nie można przydzielić bufora na każdą z nich w procesie scalania.
- W algorytmach opartych na haszowaniu po zmniejszeniu M można ograniczyć liczbę kubełków, o ile nie staną się one tak duże, że przestaną mieścić się w przydzielonej pamięci głównej. Jednak — inaczej niż w algorytmach opartych na sortowaniu — nie można reagować na zmiany w M w czasie działania algorytmu. Po określeniu liczba kubełków pozostaje stała w czasie pierwszego przebiegu. Jeśli bufory staną się niedostępne, bloki należące do niektórych kubełków trzeba będzie usunąć na dysk.

15.7.4. Ćwiczenia do podrozdziału 15.7

Ćwiczenie 15.7.1. Załóżmy, że chcemy przeprowadzić złączenie $R \bowtie S$, a dostępna pamięć waha się między M i $M/2$. Za pomocą M, $B(R)$ i $B(S)$ podaj warunki, które gwarantują możliwość wykonania poniższych algorytmów:

a) jednoprzebiegowego złączenia,
b) dwuprzebiegowego złączenia opartego na haszowaniu,
c) dwuprzebiegowego złączenia opartego na sortowaniu.

! **Ćwiczenie 15.7.2.** Jak zmniejsza się liczba dyskowych operacji wejścia-wyjścia w złączeniu zagnieżdżonym po udostępnieniu dodatkowych buforów, jeśli strategią wymiany buforów jest:

a) pierwszy na wejściu, pierwszy na wyjściu,

b) algorytm „zegarowy".

!! **Ćwiczenie 15.7.3.** W przykładzie 15.15 zasugerowano, że w czasie złączenia można wykorzystać dodatkowe dostępne bufory, przechowując w nich więcej niż jeden blok z R i przeglądając bloki z R w odwrotnej kolejności w parzystych iteracjach pętli zewnętrznej. Można też jednak udostępnić tylko jeden bufor na R i zwiększyć liczbę buforów dostępnych dla S. Która strategia prowadzi do najmniejszej liczby dyskowych operacji wejścia-wyjścia?

15.8. Algorytmy o więcej niż dwóch przebiegach

Choć dwa przebiegi wystarczają przy operacjach na wszystkich relacjach, oprócz tych największych, należy wiedzieć, że podstawowe techniki omówione w podrozdziałach 15.4 i 15.5 można uogólnić na algorytmy, które — w odpowiedniej liczbie przebiegów — przetwarzają relacje o dowolnym rozmiarze. W tym podrozdziale omówiono uogólnienie podejść opartych na sortowaniu i haszowaniu.

15.8.1. Wieloprzebiegowe algorytmy oparte na sortowaniu

W punkcie 15.4.1 napomknięto, że DWSPS można rozwinąć w algorytm trzyprzebiegowy. Istnieje prosta rekurencyjna technika sortowania, które umożliwia kompletne sortowanie dowolnie dużych relacji lub tworzenie n posortowanych podlist dla dowolnego n.

Załóżmy, że do sortowania relacji R można wykorzystać M buforów w pamięci głównej (przyjmujemy, że R jest sklastrowana). Oto potrzebne operacje.

PRZYPADEK PODSTAWOWY: Jeśli R mieści się w M blokach ($B(R) \leq M$), należy wczytać R do pamięci głównej, posortować za pomocą dowolnego algorytmu sortującego w pamięci głównej i zapisać posortowaną relację na dysku.

INDUKCJA: Jeżeli R nie mieści się w pamięci głównej, trzeba podzielić bloki z R na M grup — $R_1, R_2, \ldots, R_M$. Należy rekurencyjnie posortować R_i dla $i = 1, 2, \ldots, M$, a następnie scalić M posortowanych podlist, tak jak w punkcie 15.4.1.

Jeśli ma miejsce nie tylko sortowanie R, ale też wykonywanie na R operacji jednoargumentowej, na przykład γ lub δ, należy zmodyfikować powyższy algorytm, tak aby przy ostatecznym scalaniu wykonać działania na krotkach z początków posortowanych podlist. Oto te działania.

- Dla δ należy zwrócić jedną kopię każdej specyficznej krotki i pominąć jej pozostałe wystąpienia.
- Dla γ trzeba posortować dane tylko według atrybutów grupujących i w odpowiedni sposób połączyć krotki o danej wartości tych atrybutów, tak jak opisano to w punkcie 15.4.3.

Przy wykonywaniu operacji dwuargumentowej, takiej jak obliczanie części wspólnej lub złączenia, stosowany jest ten sam pomysł, przy czym dwie pierwsze relacje należy najpierw podzielić na łącznie M podlist. Następnie każdą podlistę trzeba posortować za pomocą opisanego algorytmu rekurencyjnego. Na zakończenie każda z M podlist wczytywana jest do jednego bufora i można wykonać operacje w sposób omówiony w odpowiednich punktach podrozdziału 15.4.

M buforów można podzielić między relacje R i S w dowolny sposób. Aby jednak zminimalizować łączną liczbę przebiegów, zwykle należy podzielić bufory proporcjonalnie do liczby bloków zajmowanych przez relacje. Na R należy przeznaczyć $M \times B(R)/(B(R) + B(S))$ buforów, a na S — resztę.

15.8.2. Wydajność wieloprzebiegowych algorytmów opartych na sortowaniu

Rozważmy teraz związek między liczbą potrzebnych dyskowych operacji wejścia-wyjścia, rozmiarem relacji i pojemnością pamięci głównej. Niech $s(M, k)$ będzie maksymalnym rozmiarem relacji, którą można posortować za pomocą M buforów i k przebiegów. Wtedy można obliczyć $s(M, k)$ w następujący sposób.

PRZYPADEK PODSTAWOWY: Jeśli $k = 1$ (dopuszczalny jest jeden przebieg), konieczne jest, aby $B(R) \leq M$. Ujmijmy to inaczej — $s(M, 1) = M$.

INDUKCJA: Przyjmijmy, że $k > 1$. Wtedy można podzielić R na M części, z których każdą można posortować w $k - 1$ przebiegach. Jeśli $B(R) = s(M, k)$, to $s(M, k)/M$ (czyli rozmiar każdej z M części R) nie może przekraczać $s(M, k - 1)$. Tak więc $s(M, k) = M\, s(M, k - 1)$.

Po rozwinięciu rekurencji okazuje się, że:

$$s(M, k) = M\, s(M, k - 1) = M^2 s(M, k - 2) = \ldots = M^{k-1} s(M, 1)$$

Ponieważ $s(M, 1) = M$, okazuje się, że $s(M, k) = M^k$. W k przebiegach można posortować relację R, jeśli $B(R) \leq M^k$. Ujmijmy to inaczej — przy sortowaniu relacji R w k przebiegach minimalną liczbą buforów jest $M = (B(R))^{1/k}$.

Algorytm sortujący w każdym przebiegu wczytuje wszystkie dane z dysku i ponownie je zapisuje. Dlatego k-przebiegowy algorytm sortujący wymaga $2kB(R)$ dyskowych operacji wejścia-wyjścia.

Rozważmy teraz koszt wieloprzebiegowego złączenia $R(X, Y) \bowtie S(Y, Z)$ (jest to reprezentatywna operacja dwuargumentowa na relacjach). Niech $j(M, k)$ to największa liczba bloków, taka że w k przebiegach z wykorzystaniem M buforów można złączyć relacje mające w sumie $j(M, k)$ lub mniej bloków. Oznacza to, że złączenie jest możliwe, jeśli $B(R) + B(S) \leq j(M, k)$.

W ostatnim przebiegu należy scalić M posortowanych podlist z obu relacji. Każda z podlist jest sortowana za pomocą $k - 1$ przebiegów, dlatego nie może być dłuższa niż $s(M, k - 1) = M^{k-1}$, co w sumie daje $M\, s(M, k - 1) = M^k$. Tak więc $B(R) + B(S) \leq M^k$. Po odwróceniu ról parametrów można ponadto stwierdzić, że do obliczenia złączenia w k przebiegach potrzeba $(B(R) + B(S))^{1/k}$ buforów.

Przy określaniu liczby dyskowych operacji wejścia-wyjścia potrzebnych w algorytmach wieloprzebiegowych należy pamiętać, że — inaczej niż przy sortowaniu — dla złączeń i innych operacji na relacjach nie trzeba uwzględniać zapisu końcowego wyniku na dysk. Dlatego sortowanie podlist wymaga $2(k - 1)(B(R) + B(S))$ operacji, a kolejnych $B(R) + B(S)$ operacji potrzeba na wczytanie posortowanych podlist w ostatnim przebiegu. W sumie oznacza to $(2k - 1)(B(R) + B(S))$ operacji.

15.8.3. Wieloprzebiegowe algorytmy oparte na haszowaniu

W operacjach na dużych relacjach istnieje powiązana rekurencyjna technika haszowania. Relację (lub relacje) należy podzielić przez haszowanie na $M - 1$ kubełków, przy czym M to liczba dostępnych buforów w pamięci. Operacje jednoargumentowe należy wykonać osobno na każdym kubełku. Operacje dwuargumentowe, takie jak złączenie, trzeba przeprowadzić na każdej parze odpowiadających sobie kubełków, tak jakby były całymi relacjami. Technikę tę można opisać rekurencyjnie.

PRZYPADEK PODSTAWOWY: Jeśli w operacji jednoargumentowej relacja mieści się w M buforach, należy wczytać ją do pamięci i wykonać operację. Jeżeli w operacji dwuargumentowej jedna z relacji mieści się w $M - 1$ buforach, trzeba wykonać operację, wczytując relację do pamięci głównej, a następnie wczytać drugą relację blok po bloku do M-tego bufora.

INDUKCJA: Jeżeli żadna relacja nie mieści się w pamięci głównej, należy za pomocą haszowania podzielić obie na $M - 1$ kubełków, co opisano w punkcie 15.5.1. Następnie trzeba rekurencyjnie wykonywać operację na każdym kubełku (lub każdej parze) i akumulować dane wyjściowe z wszystkich kubełków (lub par).

15.8.4. Wydajność wieloprzebiegowych algorytmów opartych na haszowaniu

Dalej w tekście obowiązuje założenie, że przy haszowaniu relacji krotki dzielone są możliwie równomiernie między kubełki. W praktyce założenie można częściowo spełnić, wybierając w pełni losową funkcję haszującą, jednak rozkład krotek w kubełkach zawsze będzie nieco nierównomierny.

Najpierw rozważmy operację jednoargumentową, na przykład γ lub δ na relacji R z wykorzystaniem M buforów. Niech $u(M, k)$ będzie liczbą bloków największej relacji, jaką może obsłużyć k-przebiegowy algorytm haszowania. Liczbę u można zdefiniować rekurencyjnie w następujący sposób.

PRZYPADEK PODSTAWOWY: $u(M, 1) = M$, ponieważ relacja R musi mieścić się w M buforach (a więc $B(R) \leq M$).

INDUKCJA: Zakładamy, że pierwszy krok to podział relacji R na $M - 1$ kubełków o równej wielkości. Dlatego można obliczyć $u(M, k)$ w opisany tu sposób. Kubełki w następnym przebiegu muszą być na tyle małe, że można je obsłużyć w $k - 1$ przebiegach. Kubełki mają więc rozmiar $u(M, k - 1)$. Ponieważ R dzielona jest na $M - 1$ kubełków, konieczne jest, aby $u(M, k) = (M - 1)u(M, k - 1)$.

Po rozwinięciu rekurencji okazuje się, że $u(M, k) = M(M - 1)^{k-1}$ lub — w przybliżeniu dla dużego M — $u(M, k) = M^k$. Można też wykonać jedną z jednoargumentowych operacji na relacji R w k przebiegach przy M buforach i założeniu, że $M \geq (B(R))^{1/k}$.

Podobną analizę można przeprowadzić dla operacji dwuargumentowych. Rozważmy złączenie (tak jak w punkcie 15.8.2). Niech $j(M, k)$ będzie górnym ograniczeniem rozmiaru mniejszej z dwóch relacji R i S ze złączenia $R(X, Y) \bowtie S(Y, Z)$. Tak jak wcześniej M to liczba dostępnych buforów, a k to liczba przebiegów.

PRZYPADEK PODSTAWOWY: $j(M, 1) = M - 1$. Oznacza to, że przy stosowaniu jednoprzebiegowego algorytmu złączenia R lub S musi mieścić się w $M - 1$ blokach, co opisano w punkcie 15.2.3.

INDUKCJA: $j(M, k) = (M - 1)j(M, k - 1)$. Oznacza to, że w pierwszym z k przebiegów można podzielić każdą relację na $M - 1$ kubełków i oczekiwać, iż każdy kubełek będzie stanowił $1/(M - 1)$ całej relacji. Możliwe musi być złączenie każdej pary odpowiadających sobie kubełków w $M - 1$ przebiegach.

Rozwijając rekurencję dla $j(M, k)$, ustalamy, że $j(M, k) = (M - 1)^k$. Przy założeniu, że M jest duże, można w przybliżeniu określić, iż $j(M, k) = M^k$. Oznacza to, że złączenie $R(X, Y) \bowtie S(Y, Z)$ można przeprowadzić w k przebiegach za pomocą M buforów, jeśli $(B(R) + B(S)) \leq M^k$.

15.8.5. Ćwiczenia do podrozdziału 15.8

Ćwiczenie 15.8.1. Załóżmy, że $B(R) = 20\,000$, $B(S) = 50\,000$ i $M = 101$. Opisz działanie poniższych algorytmów przy obliczaniu $R \bowtie S$.

a) Trójprzebiegowy algorytm oparty na sortowaniu.

b) Trójprzebiegowy algorytm oparty na haszowaniu.

! **Ćwiczenie 15.8.2.** Istnieje kilka sztuczek do zwiększania wydajności algorytmów dwuprzebiegowych. Określ, czy poniższe sztuczki można wykorzystać w algorytmie wieloprzebiegowym, a jeśli tak, to w jaki sposób.

a) Sztuczka z hybrydowego złączenia przez haszowanie z punktu 15.5.6.

b) Usprawnienie algorytmu opartego na sortowaniu przez zapisywanie bloków obok siebie na dysku (punkt 15.5.7).

c) Usprawnienie algorytmu opartego na haszowaniu przez zapisywanie bloków obok siebie na dysku (punkt 15.5.7).

15.9. Podsumowanie rozdziału 15.

- *Przetwarzanie zapytania*. Zapytania są kompilowane (co obejmuje rozbudowane optymalizowanie), a następnie wykonywane. Poznawanie dziedziny wykonywania zapytań obejmuje naukę metod wykonywania operacji algebry relacji oraz pewnych rozszerzeń zapewniających możliwości SQL-a.
- *Plany zapytań*. Zapytania są kompilowane najpierw do logicznych planów zapytań (które często przypominają wyrażenia algebry relacji), a następnie są przekształcane na fizyczny plan zapytania przez wybranie implementacji każdego operatora, określenie kolejności złączeń i dokonanie innych wyborów, co opisano w rozdziale 16.
- *Skanowanie tabeli*. Dostęp do krotek relacji zapewnia kilka operatorów fizycznych. Operator skanowania tabeli wczytuje każdy blok z krotkami relacji. Przeglądanie indeksu polega na zastosowaniu indeksu do znajdowania krotek, a skanowanie z sortowaniem powoduje posortowanie krotek.
- *Miary kosztów działania operatorów fizycznych*. Zwykle największy wpływ na czas ma liczba dyskowych operacji wejścia-wyjścia potrzebnych do wykonania danej operacji. W użytym tu modelu ważny jest tylko czas dyskowych operacji wejścia-wyjścia, przy czym uwzględniono wyłącznie czas i pamięć potrzebne na wczytanie argumentów, a nie na zapisanie wyniku.

- *Iteratory*. Kilka operacji związanych z wykonywaniem zapytania można wygodnie połączyć, jeśli przeprowadzi się je za pomocą operatora. Mechanizm ten obejmuje trzy metody — rozpoczynającą tworzenie relacji, generującą następną krotkę relacji i kończącą proces.
- *Algorytmy jednoprzebiegowe*. O ile jeden z argumentów operatora algebry relacji mieści się w pamięci głównej, można wykonać operację przez wczytanie mniejszej relacji do pamięci i wczytywanie drugiego argumentu blok po bloku.
- *Złączenie zagnieżdżone*. Ten prosty algorytm złączenia działa nawet wtedy, kiedy żaden argument nie mieści się w pamięci głównej. Wczytuje jak największą część mniejszej relacji do pamięci i porównuje ją z całym drugim argumentem. Proces należy powtarzać do czasu wczytania kolejno wszystkich części mniejszej relacji do pamięci.
- *Algorytmy dwuprzebiegowe*. Oprócz złączenia zagnieżdżonego, większość algorytmów dla argumentów niemieszczących się w pamięci oparta jest na sortowaniu, haszowaniu lub indeksie.
- *Algorytmy oparte na sortowaniu*. Dzielą argumenty na posortowane podlisty o wielkości pamięci głównej. Podlisty te są odpowiednio scalane w pożądany wynik. Przykładowo do scalania krotek wszystkich podlist w posortowanej kolejności służy dwuetapowe wielościeżkowe sortowanie przez scalanie.
- *Algorytmy oparte na haszowaniu*. Tu do podziału argumentów na kubełki służy funkcja haszująca. Następnie operację należy zastosować dla odrębnych kubełków (w operacjach jednoargumentowych) lub dla ich par (dla operacji dwuargumentowych).
- *Haszowanie a sortowanie*. Algorytmy oparte na haszowaniu są często lepsze od opartych na sortowaniu, ponieważ tylko jeden z ich argumentów musi być „mały". Algorytmy oparte na sortowaniu działają dobrze, jeśli istnieje inna przyczyna do przechowywania danych w posortowanej postaci.
- *Algorytmy oparte na indeksach*. Użycie indeksu to doskonały sposób na przyspieszenie selekcji, w której w warunku atrybut z indeksu porównywany jest ze stałą. Złączenia oparte na indeksie to znakomita technika, jeśli jedna z relacji jest mała, a dla drugiej istnieje indeks oparty na atrybutach używanych w złączeniu.
- *Menedżer buforów*. Dostępność bloków pamięci kontroluje menedżer buforów. Kiedy w pamięci potrzebny jest nowy bufor, menedżer stosuje jedną ze znanych strategii wymiany (na przykład LRU) do określenia, który bufor zostanie zapisany na dysku.
- *Radzenie sobie ze zmienną liczbą buforów*. Często nie można z góry przewidzieć liczby buforów w pamięci głównej dostępnych dla operacji. Wtedy wydajność algorytmu wykonywania operacji powinna łagodnie maleć wraz ze zmniejszaniem się liczby dostępnych buforów.
- *Algorytmy wieloprzebiegowe*. Algorytmy dwuprzebiegowe oparte na sortowaniu lub haszowaniu mają naturalne rekurencyjne odpowiedniki, które wymagają trzech lub więcej przebiegów i działają dla większych ilości danych.

15.10. Literatura do rozdziału 15.

[6] i [2] to dwa przeglądy optymalizacji zapytań. [8] zawiera przegląd optymalizacji zapytań rozproszonych.

W [5] opisano wczesne badania metod złączenia. W [3] znajdują się analizy, przegląd i usprawnienia zarządzania pulą buforów.

Pierwszy opis zastosowania technik opartych na sortowaniu znajduje się w [1]. Zalety algorytmów opartych na haszowaniu przedstawiono w [7] i [4]; ta ostatnia praca to źródło hybrydowego złączenia opartego na haszowaniu.

(1) M. W. Blasgen i K. P. Eswaran, *Storage access in relational databases*, „IBM Systems J." 16:4 (1977), s. 363 – 378.

(2) S. Chaudhuri, *An overview of query optimization in relational systems*, „Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems", s. 34 – 43, czerwiec, 1998.

(3) H.-T. Chou i D. J. DeWitt, *An evaluation of buffer management strategies for relational database systems*, „Intl. Conf. on Very Large Databases", s. 127 – 141, 1985.

(4) D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker i D. Wood, *Implementation techniques for main-memory database systems*, „Proc. ACM SIGMOD Intl. Conf. on Management of Data" (1984), s. 1 – 8.

(5) L. R. Gotlieb, *Computing joins of relations*, „Proc. ACM SIGMOD Intl. Conf. on Management of Data" (1975), s. 55 – 63.

(6) G. Graefe, *Query evaluation techniques for large databases*, „Computing Surveys" 25:2 (czerwiec, 1993), s. 73 – 170.

(7) M. Kitsuregawa, H. Tanaka i T. Moto-oka, *Application of hash to data base machine and its architecture*, „New Generation Computing" 1:1 (1983), s. 66 – 74.

(8) D. Kossman, *The state of the art in distributed query processing*, „Computing Surveys" 32:4 (grudzień, 2000), s. 422 – 469.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

SYSTEMY BAZ DANYCH
Kompletny podręcznik

KANON INFORMATYKI

Z kluczowej roli, jaką bazy danych odgrywają w codziennym życiu milionów ludzi, zdajemy sobie sprawę za każdym razem, gdy wpisujemy hasło w wyszukiwarce Google, robimy zakupy w internetowej księgarni czy logujemy się do swojego konta w banku. Szybkie, bezpieczne i niezawodne przetwarzanie oraz przechowywanie ogromnych ilości informacji stało się dziś strategicznym czynnikiem funkcjonowania większości firm, organizacji i instytucji państwowych. Ten ogromny potencjał współczesnych baz danych jest sumą wiedzy i technologii rozwijanych przez kilka ostatnich dziesięcioleci. Owocem tych prac jest przede wszystkim wyspecjalizowane oprogramowanie – systemy zarządzania bazami danych DBMS, czyli rozbudowane narzędzia do wydajnego tworzenia dużych zbiorów informacji i zarządzania nimi. Niestety, mają one jedną zasadniczą wadę – należą do najbardziej złożonych rodzajów oprogramowania.

W związku z tym trzech znanych naukowców z dziedziny IT z Uniwersytetu Stanforda – Hector Garcia-Molina, Jeffrey D. Ullman i Jennifer Widom – postanowiło stworzyć pierwszy kompletny podręcznik, wprowadzający do systemów baz danych. Zawiera on opis najnowszych standardów bazy danych SQL 1999, SQL/PSM, SQL/CLI, JDBC, ODL oraz XML – i to w znacznie szerszym zakresie niż w większości publikacji. Podręcznik został przygotowany w taki sposób, aby po jego przeczytaniu użytkowanie czy projektowanie baz danych, pisanie programów w różnych językach związanych z systemami DBMS oraz ich sprawna implementacja nie stanowiły dla Czytelnika najmniejszego problemu!

HECTOR GARCIA-MOLINA jest profesorem nauk komputerowych i inżynierii elektrycznej na Uniwersytecie Stanforda. Do jego zainteresowań naukowych należą biblioteki cyfrowe oraz integrowanie informacji i zastosowania baz danych w internecie. Jest laureatem wyróżnienia SIGMOD Innovations i członkiem prezydenckiej grupy doradczej do spraw technologii informacyjnych. Obecnie zajmuje stanowisko członka zarządu Oracle Corp.

JEFFREY D. ULLMAN jest emerytowanym profesorem nauk komputerowych na Uniwersytecie Stanforda. Jest autorem lub współautorem szesnastu podręczników informatycznych. Jego zainteresowania naukowe obejmują drążenie danych, integrowanie informacji i nauczanie wspomagane elektronicznie. Jest członkiem National Academy of Engineering i laureatem wielu prestiżowych wyróżnień, m.in. Guggenheim Fellowship i Knuth Prize.

JENNIFER WIDOM jest profesorem nauk komputerowych i inżynierii elektrycznej na Uniwersytecie Stanforda. Jej zainteresowania naukowe obejmują wiele aspektów nietradycyjnego zarządzania danymi. Należy do organizacji ACM i National Academy of Engineering. Jest laureatką wyróżnień ACM SIGMOD Edgar F. Codd Innovations i Guggenheim Fellowship.

Kompleksowe podejście do systemów baz danych — z punktu widzenia projektanta, użytkownika i programisty.

W tej książce znajdziesz obszerne omówienie między innymi:

- modelowania relacyjnego i wysokopoziomowego oraz języka UML i ODL
- zależności funkcyjnych i wielowartościowych oraz algorytmów umożliwiających manipulowanie zależnościami
- różnych aspektów programowania w języku SQL
- architektury trójwarstwowej, kostek danych, modelu relacji zagnieżdżonych i obiektowo-relacyjnych funkcji SQL-a
- XML-a i systemów opartych na tym języku
- zagadnień dotyczących sprawnego implementowania baz danych
- struktur używanych w indeksach, w tym drzew zbalansowanych, struktur haszujących i struktur dla indeksów wielowymiarowych
- wykonywania i optymalizowania zapytań, rejestrowania zdarzeń, kontrolowania współbieżności
- równoległych i rozproszonych baz danych, platformy Map Reduce, bazy danych P2P
- kluczowych technik, takich jak shingling, MinHash i LSH, służących do wyszukiwania podobnych elementów w dużych bazach danych
- algorytmów przeszukiwania sieci WWW, w tym algorytmu PageRank i jego odmian
- modelu strumieniowania danych i rozszerzenia języka SQL

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-3303-6

Cena 99,00 zł