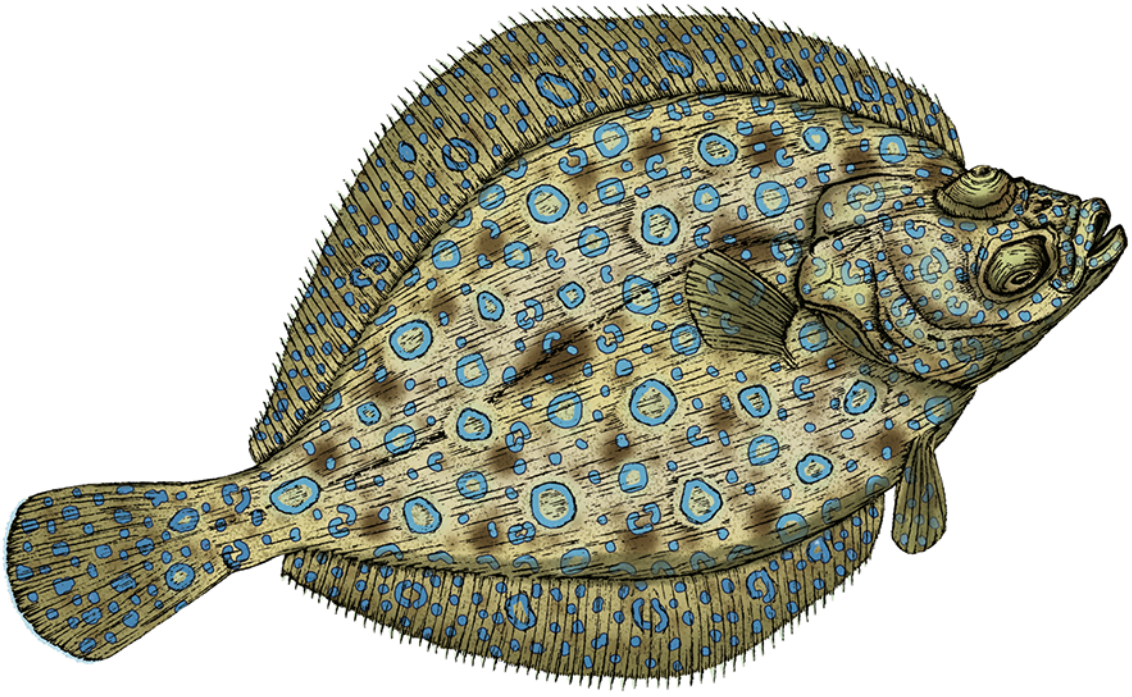
O'REILLY®

Helion

Baza danych od środka

Analiza działania rozproszonych systemów danych



Alex Petrov

Tytuł oryginału: Database Internals: A Deep Dive into How Distributed Data Systems Work

Tłumaczenie: Małgorzata Dąbkowska-Kowalik, Witold Sikorski

ISBN: 978-83-289-1332-5

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/badaod
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Implementowanie B-drzew

W poprzednim rozdziale omówiliśmy ogólne zasady kompozycji formatu binarnego i dowiedzieliśmy się, jak tworzyć komórki, budować hierarchie i łączyć je ze stronami za pomocą wskaźników. Koncepcje te mają zastosowanie zarówno do struktur pamięci masowej z aktualizacją w miejscu, jak i z samym dołączaniem (*append-only*). W tym rozdziale omawiamy niektóre koncepcje specyficzne dla B-drzew.

Podrozdziały w tym rozdziale są podzielone na trzy logiczne grupy. Najpierw omawiamy organizację: jak ustanowić relacje między kluczami i wskaźnikami oraz jak zaimplementować nagłówki i łącza między stronami.

Następnie omawiamy procesy zachodzące podczas schodzenia od korzenia do liścia, a mianowicie jak dokonywać wyszukiwania binarnego oraz jak zbierać tzw. okruszki (ang. *breadcrumbs*) i śledzić węzły nadrzędne na wypadek, gdybyśmy później musieli podzielić lub scalić węzły.

Na koniec omawiamy techniki optymalizacji (równoważenie, dołączanie tylko z prawej strony i masowe ładowanie), procesy konserwacji i zbieranie śmieci.

Nagłówek strony

Nagłówek strony zawiera informacje o stronie, które można wykorzystać do nawigacji, konserwacji i optymalizacji. Zwykle składa się z flag opisujących zawartość strony i jej układ, liczbę komórek na stronie, dolne i górne przesunięcia oznaczające pustą przestrzeń (używane do dołączania przesunięć komórek i danych) oraz inne przydatne metadane.

Na przykład PostgreSQL (*https://databass.dev/links/12*) przechowuje w nagłówku rozmiar strony i wersję układu w nagłówku. W MySQL InnoDB (*https://databass.dev/links/13*) nagłówek strony przechowuje liczbę rekordów sterty, poziom i inne wartości specyficzne dla implementacji. W SQLite (*https://databass.dev/links/14*) nagłówek strony przechowuje liczbę komórek i skrajny prawy wskaźnik.

Magiczne liczby

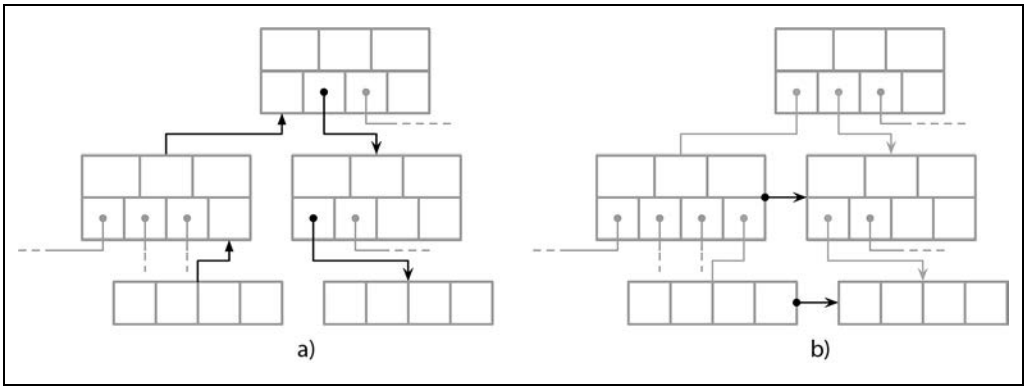
Jedną z wartości często umieszczanych w nagłówku pliku lub strony jest magiczna liczba. Zazwyczaj jest to wielobajtowy blok zawierający stałą wartość, która może być użyta do zasygnalizowania, że blok reprezentuje stronę, określa jej rodzaj lub identyfikuje jej wersję.

Magiczne liczby są często używane do sprawdzania poprawności i czystości kodu [GIAMPAOLO98]. To bardzo mało prawdopodobne, aby sekwencja bajtów w losowym przesunięciu była zgodna z magiczną liczbą. Jeśli tak się stanie, to istnieje duża szansa, że przesunięcie jest poprawne. Na przykład aby sprawdzić, czy strona jest poprawnie załadowana i wyrównana, możemy podczas zapisu umieścić w nagłówku magiczną liczbę `50 41 47 45` (heksadecymalna postać słowa `PAGE`). Podczas odczytu możemy sprawdzić poprawność strony, porównując cztery bajty z nagłówka odczytu z oczekiwaną sekwencją bajtów.

Powiązania między rodzeństwem

Niektóre implementacje przechowują powiązania do przodu i do tyłu, wskazujące strony lewą i prawą rodzeństwa. Powiązania te pomagają zlokalizować sąsiednie węzły bez konieczności przechodzenia z powrotem do węzła nadrzędnego (rodzica). Podejście to zwiększa złożoność operacji dzielenia i scalania, ponieważ przesunięcia rodzeństwa również muszą zostać zaktualizowane. Na przykład gdy węzeł jest dzielony, wskaźnik wsteczny jego prawego rodzeństwa (wcześniej wskazujący węzeł, który został podzielony) musi zostać ponownie powiązany, aby wskazywał nowo utworzony węzeł.

Na rysunku 4.1 widać, że aby zlokalizować węzeł rodzeństwa, musimy odwołać się do węzła nadrzędnego, chyba że rodzeństwo jest połączone. Ta operacja może sięgać aż do korzenia, ponieważ bezpośredni rodzic może pomóc tylko w odwołaniu do *własnych* dzieci. Jeśli przechowujemy powiązania między rodzeństwem bezpośrednio w nagłówku, możemy po prostu iść za nimi, aby zlokalizować poprzedni lub następny węzeł na tym samym poziomie.



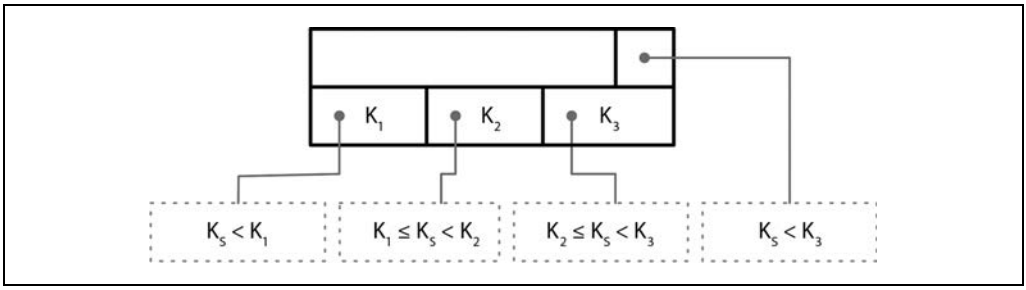
Rysunek 4.1. Lokalizowanie rodzeństwa za pomocą powiązań nadrzędnych (a) i powiązań rodzeństwa (b)

Jedną z wad przechowywania powiązań między rodzeństwem jest to, że muszą one być aktualizowane podczas podziałów i scaleń. Ponieważ aktualizacje muszą odbywać się w węźle rodzeństwa, a nie w węźle podziału bądź łączenia, może to wymagać dodatkowej blokady. Sposób, w jaki powiązania między rodzeństwem mogą być użyteczne w implementacji współbieżnego B-drzewa, omawiamy w punkcie „Drzewa Blink” w rozdziale 5.

Skrajne prawe wskaźniki

Klucze oddzielające B-drzewa mają ścisłe niezmienniki: są używane do dzielenia drzewa na poddrzewa i poruszania się po nich, więc zawsze liczba wskaźników do stron podrzędnych jest o jeden większa od liczby kluczy. Stąd bierze się +1 wspomniane w punkcie „Liczenie kluczy" w rozdziale 2.

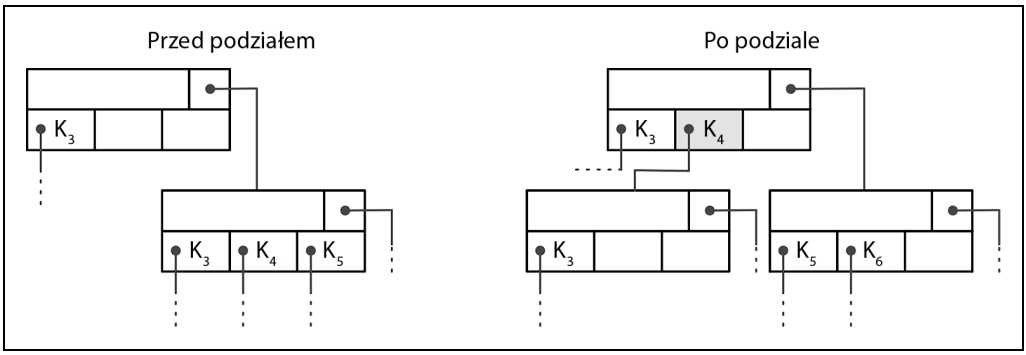
W punkcie „Klucze oddzielające" w rozdziale 2 opisaliśmy niezmienniki kluczy oddzielających. W wielu implementacjach węzły przypominają bardziej klucze pokazane na rysunku 4.2: każdy klucz oddzielający ma wskaźnik podrzędny, podczas gdy ostatni wskaźnik jest przechowywany osobno, ponieważ nie jest sparowany z żadnym kluczem. Można to porównać z rysunkiem 2.10.



Rysunek 4.2. Skrajny prawy wskaźnik

Ten dodatkowy wskaźnik może być przechowywany w nagłówku, tak jak na przykład jest to zaimplementowane w SQLite (*https://databass.dev/links/16*).

Jeśli skrajne prawe dziecko zostanie podzielone, a nowa komórka dołączona do jej rodzica, wskaźnik skrajnego prawego dziecka musi zostać ponownie przypisany. Jak pokazano na rysunku 4.3 — po podziale komórka dołączona do rodzica (pokazana na szaro) przechowuje klucz o podwyższonym poziomie i wskazuje podzielony węzeł. Wskaźnik do nowego węzła jest przypisywany zamiast poprzedniego skrajnego prawego wskaźnika. Podobne podejście zostało opisane i zaimplementowane w SQLite[1].



Rysunek 4.3. Aktualizacja skrajnego prawego wskaźnika podczas podziału węzła. Klucz o podwyższonym poziomie jest pokazany na szarym polu

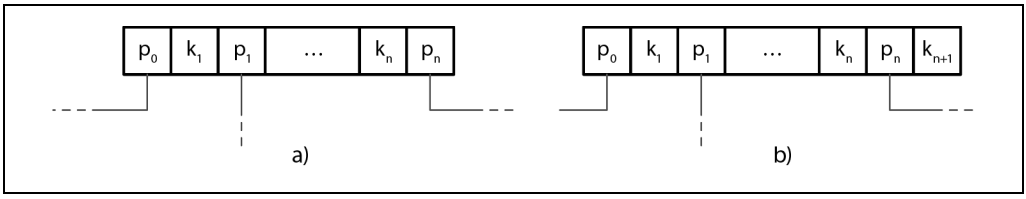
[1] Algorytm ten można znaleźć w funkcji `balance_deeper` w repozytorium projektu (*https://databass.dev/links/15*).

Najwyższe klucze węzłów

Możemy przyjąć nieco inne podejście i przechowywać skrajny prawy wskaźnik w komórce wraz z **najwyższym kluczem** (ang. *high key*) węzła. Najwyższy klucz, zgodnie z nazwą, to najwyższy możliwy klucz, który może być obecny w poddrzewie pod bieżącym węzłem. Podejście to jest stosowane przez PostgreSQL i jest nazywane drzewem Blink (w celu zapoznania się z implikacjami tego podejścia dla współbieżności patrz punkt „Drzewa Blink" w rozdziale 5).

B-drzewa mają N kluczy (oznaczonych przez K_i) i $N+1$ wskaźników (oznaczonych przez P_i). W każdym poddrzewie klucze są ograniczone przez $K_{i-1} \leq K_s < K_i$. Wartość $K_0 = -\infty$ jest domyślna i nie występuje w węźle.

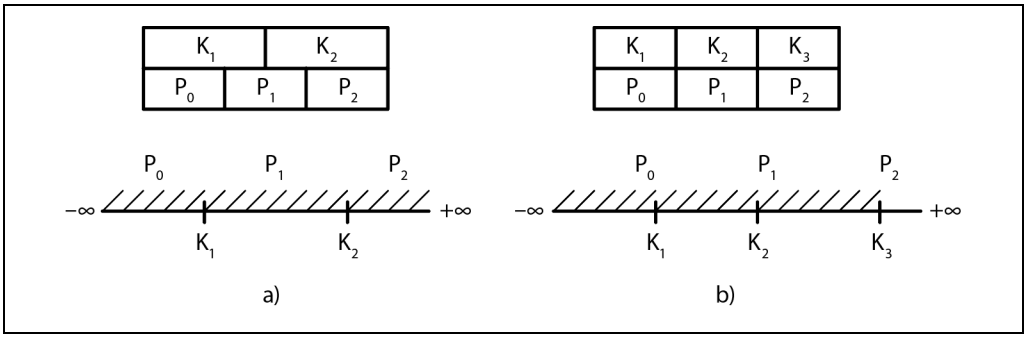
Drzewa Blink dodają klucz K_{N+1} do każdego węzła. Określa on górną granicę kluczy, które mogą być przechowywane we wskazywanym przez wskaźnik P_N poddrzewie, a zatem jest to górne ograniczenie wartości, jakie mogą być przechowywane w bieżącym poddrzewie. Oba podejścia są pokazane na rysunku 4.4: (a) pokazuje węzeł *bez* najwyższego klucza, a (b) pokazuje węzeł z najwyższym kluczem.



Rysunek 4.4. B-drzewa bez najwyższego klucza (a) i z najwyższym kluczem (b)

W tym przypadku wskaźniki mogą być przechowywane parami, a każda komórka może mieć odpowiadający jej wskaźnik, co z kolei może uprościć obsługę prawego wskaźnika, ponieważ nie ma tak wielu przypadków brzegowych do rozważenia.

Na rysunku 4.5 można zobaczyć schematyczną strukturę strony dla obu podejść i to, jak przestrzeń poszukiwań jest dla tych przypadków w różny sposób dzielona: do $+\infty$ w pierwszym przypadku i do górnej granicy K_3 w drugim przypadku.



Rysunek 4.5. Użycie $+\infty$ jako klucza wirtualnego (a) w porównaniu z przechowywaniem najwyższego klucza (b)

Strony przepełnienia

Rozmiar węzła i wartości stopnia rozgałęzienia drzewa są stałe i nie zmieniają się dynamicznie. Byłoby również trudno znaleźć wartość, która byłaby optymalna w każdej sytuacji: jeśli w drzewie występują wartości o zmiennej wielkości i są dość duże, to tylko kilka z nich może zmieścić się na stronie. Jeśli wartości są małe, marnujemy zarezerwowane miejsce.

Algorytm B-drzewa określa, że każdy węzeł przechowuje określoną liczbę elementów. Ponieważ niektóre wartości mają różne rozmiary, możemy znaleźć się w sytuacji, w której zgodnie z algorytmem B-drzewa węzeł nie jest jeszcze *pełny*, ale *nie ma już wolnego miejsca na stronie* o stałym rozmiarze, przechowującej ten węzeł. Zmiana rozmiaru strony wymaga skopiowania już zapisanych danych do nowego obszaru i często jest niepraktyczna. Nadal jednak musimy znaleźć sposób na zwiększenie lub rozszerzenie rozmiaru strony.

Aby zaimplementować węzły o zmiennym rozmiarze bez kopiowania danych do nowego ciągłego obszaru, możemy budować węzły z wielu połączonych stron. Na przykład domyślny rozmiar strony wynosi 4 kB, a po wstawieniu kilku wartości rozmiar danych wzrasta do ponad 4 kB. Zamiast zezwalać na dowolne rozmiary możemy sprawić, by węzły rosły w przyrostach 4 kB; przydzielamy więc stronę rozszerzenia 4 kB i łączymy ją z początkową stroną. Te połączone rozszerzenia stron są nazywane **stronami przepełnienia**. Dla jasności, oryginalną stronę nazywamy w tym podrozdziale **stroną główną**.

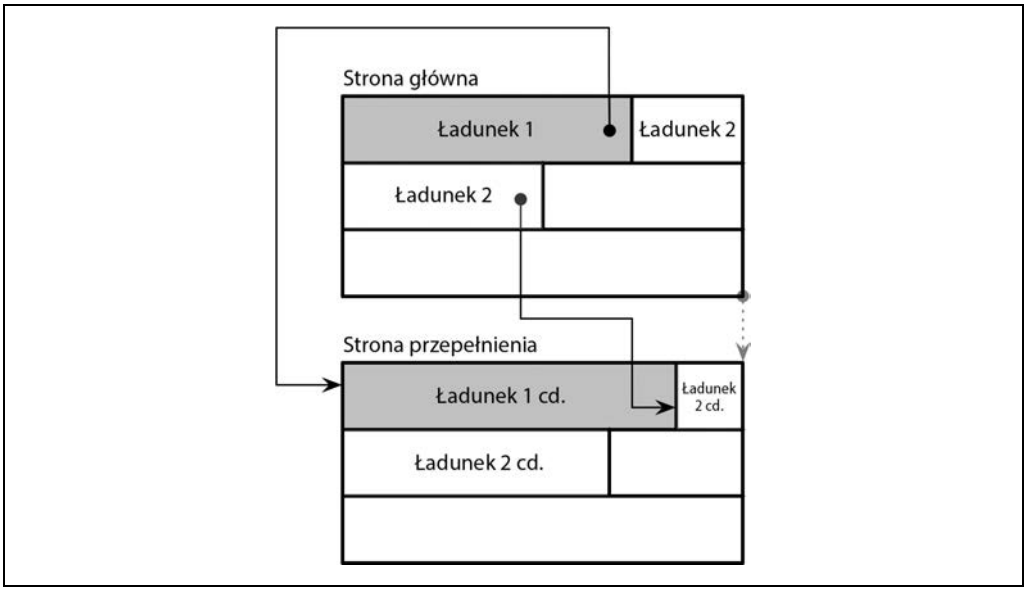
Większość implementacji B-drzew umożliwia bezpośrednie przechowywanie w węźle tylko ustalonej liczby bajtów obciążenia i *przeniesienie* reszty na stronę przepełnienia. Wartość ta jest obliczana przez podzielenie rozmiaru węzła przez stopień rozgałęzienia. Korzystając z tego podejścia, nie możemy doprowadzić do sytuacji, w której strona nie ma wolnego miejsca, ponieważ zawsze będzie miała co najmniej `max_payload_size` bajtów. Więcej informacji na temat stron przepełnienia w SQLite można znaleźć w repozytorium kodu źródłowego SQLite (*https://databass.dev/links/16*). Warto sprawdzić również dokumentację MySQL InnoDB (*https://databass.dev/links/17*).

Gdy wstawione obciążenie jest większe niż `max_payload_size`, węzeł jest sprawdzany pod kątem tego, czy ma już powiązane strony przepełnienia. Jeśli strona przepełnienia już istnieje i ma wystarczającą ilość dostępnego miejsca, są do niej przenoszone dodatkowe bajty z ładunku. W przeciwnym razie przydzielana jest nowa strona przepełnienia.

Na rysunku 4.6 można zobaczyć stronę główną i stronę przepełnienia z rekordami ze strony głównej, wskazującymi na stronę przepełnienia, gdzie znajduje się dalsza część ich ładunku.

Strony przepełnienia wymagają dodatkowej obsługi technicznej, ponieważ mogą zostać pofragmentowane podobnie jak strony podstawowe i musimy być w stanie odzyskać tę przestrzeń, aby zapisać nowe dane, lub odrzucić stronę przepełnienia, jeśli nie jest już potrzebna.

Gdy przydzielana jest pierwsza strona przepełnienia, jej identyfikator jest przechowywany w nagłówku strony głównej. Jeśli pojedyncza strona przepełnienia nie jest wystarczająca, strony przepełnienia są łączone w ten sposób, że identyfikator następnej strony przepełnienia jest przechowywany w nagłówku poprzedniej. W celu zlokalizowania części z przepełnieniem dla danego ładunku może być konieczne przejście przez kilka stron.



Rysunek 4.6. Strony przepełnienia

Ponieważ klucze mają zwykle dużą liczebność, przechowywanie fragmentu klucza ma sens, gdyż większość porównań można wykonać na przyciętej części klucza, która znajduje się na stronie głównej.

W przypadku rekordów danych musimy zlokalizować ich części z przepełnieniem, aby zwrócić je użytkownikowi. Nie ma to jednak większego znaczenia, ponieważ jest to rzadka operacja. Jeśli wszystkie rekordy danych są zbyt duże, warto rozważyć wyspecjalizowane przechowywanie blobów dla dużych wartości.

Wyszukiwanie binarne

Omówiliśmy już algorytm przeszukiwania B-drzewa (patrz podrozdział „Algorytm przeszukiwania B-drzewa" w rozdziale 2) i wspomnieliśmy, że lokalizujemy szukany klucz w węźle przy użyciu algorytmu **wyszukiwania binarnego**. Wyszukiwanie binarne działa *tylko* dla posortowanych danych. Jeśli klucze nie są uporządkowane, nie mogą być przeszukiwane binarnie. Dlatego właśnie niezbędne jest trzymanie kluczy w kolejności i utrzymywanie posortowanego niezmiennika.

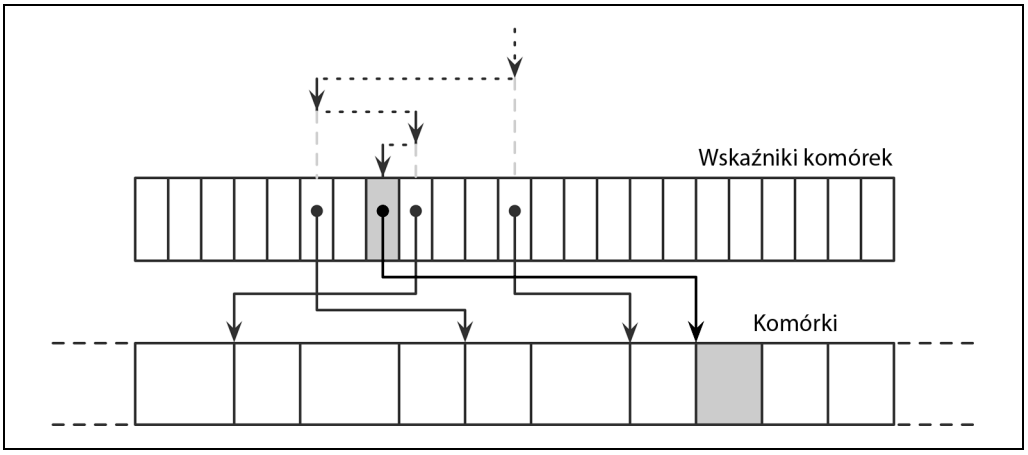
Algorytm wyszukiwania binarnego otrzymuje tablicę posortowanych elementów oraz szukany klucz i zwraca liczbę. Jeśli zwrócona liczba jest dodatnia, wiemy, że szukany klucz został znaleziony, a liczba określa jego pozycję w tablicy wejściowej. Ujemna wartość wskazuje, że szukanego klucza nie ma w tablicy wejściowej, i daje nam **punkt wstawiania**.

Punkt wstawiania jest indeksem pierwszego elementu, który jest *większy niż* podany klucz. Wartość bezwzględna tej liczby jest indeksem, pod którym można wstawić szukany klucz, aby zachować kolejność. Wstawienia można dokonać poprzez przesunięcie elementów o jedną pozycję, zaczynając od punktu wstawiania, aby zrobić miejsce dla wstawionego elementu [SEDGEWICK11].

Większość wyszukiwań na wyższych poziomach nie daje dokładnych dopasowań, a nas interesuje kierunek wyszukiwania, w przypadku którego musimy znaleźć pierwszą wartość, większą niż ta szukana, i podążać za odpowiednim łączem podrzędnym do powiązanego poddrzewa.

Wyszukiwanie binarne ze wskaźnikami kierunku

Komórki na stronie B-drzewa są przechowywane w kolejności wstawiania, a tylko przesunięcia komórek zachowują logiczną kolejność elementów. Aby wykonać wyszukiwanie binarne przez komórki strony, wybieramy przesunięcie środkowej komórki, podążamy za jej wskaźnikiem, aby zlokalizować komórkę, porównujemy klucz z tej komórki z wyszukiwanym kluczem, by zdecydować, czy wyszukiwanie powinno być dalej przeprowadzane w lewo, czy w prawo, i kontynuujemy ten proces rekurencyjnie, aż do znalezienia szukanego elementu lub punktu wstawienia, jak pokazano na rysunku 4.7.



Rysunek 4.7. Wyszukiwanie binarne ze wskaźnikami pośrednimi. Szukany element jest zaznaczony szarym kolorem. Kropkowane strzałki reprezentują wyszukiwanie binarne poprzez wskaźniki komórek. Linie ciągłe reprezentują dostępy do danych, które idą za wskaźnikami komórek i są niezbędne do porównania wartości komórki z wyszukiwanym kluczem

Propagowanie podziałów i scaleń

Jak omówiliśmy w poprzednich rozdziałach, podziały i scalenia B-drzewa mogą przenosić się na wyższe poziomy. Aby to zrobić, musimy być w stanie przejść łańcuch z powrotem do węzła korzenia od dzielącego się liścia lub pary łączących się liści.

Węzły B-drzewa mogą zawierać wskaźniki do węzłów nadrzędnych. Ponieważ strony z niższych poziomów są zawsze przenoszone do pamięci podręcznej, gdy są przywoływane z wyższego poziomu, nie jest nawet konieczne przechowywanie tych informacji na dysku.

Podobnie jak wskaźniki rodzeństwa (patrz punkt „Powiązania między rodzeństwem” wcześniej w tym rozdziale) wskaźniki rodzica muszą być aktualizowane za każdym razem, gdy zmienia się rodzic. Dzieje się tak we wszystkich przypadkach, gdy klucz oddzielający z identyfikatorem strony

jest przenoszony z jednego węzła do drugiego: podczas podziału, łączenia lub równoważenia węzła nadrzędnego.

Niektóre implementacje (na przykład WiredTiger (*https://databass.dev/links/20*)) używają wskaźników rodzica do przechodzenia przez liście, aby uniknąć zakleszczeń, które mogą wystąpić podczas używania wskaźników rodzeństwa (patrz [MILLER78], [LEHMAN81]). Zamiast używać wskaźników rodzeństwa do przechodzenia przez węzły liścia, algorytm wykorzystuje wskaźniki rodzica, podobnie jak na rysunku 4.1.

Aby zaadresować i zlokalizować rodzeństwo, możemy podążać za wskaźnikiem z węzła nadrzędnego i rekurencyjnie zejść z powrotem do niższego poziomu. Ilekroć dotrzemy do końca węzła nadrzędnego po przejściu przez całe rodzeństwo mające wspólnego rodzica, wyszukiwanie będzie kontynuowane rekurencyjnie w górę; ostatecznie dotrzemy do korzenia, by z powrotem przejść w dół do poziomu liścia.

Okruszki

Zamiast przechowywania i utrzymywania wskaźników do węzłów nadrzędnych możliwe jest śledzenie węzłów na ścieżce do docelowego węzła liścia i podążanie za łańcuchem węzłów nadrzędnych w odwrotnej kolejności w przypadku kaskadowych podziałów podczas wstawiania lub scaleń podczas usuwania.

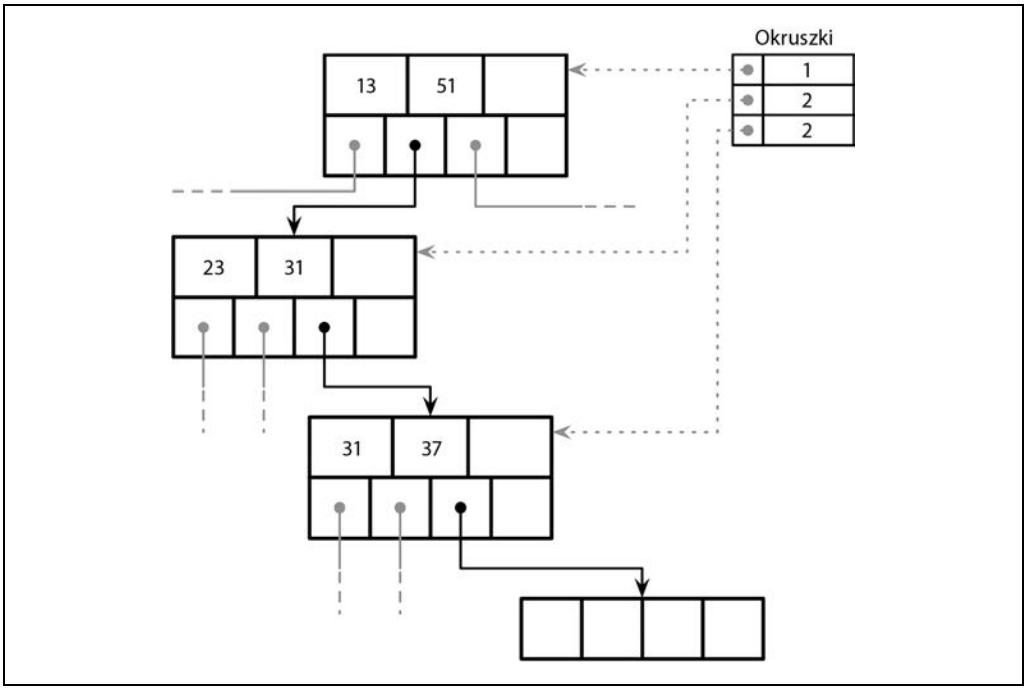
Podczas wykonywania operacji, które mogą skutkować zmianami strukturalnymi B-drzewa (wstawianie lub usuwanie), najpierw przechodzimy drzewo od korzenia do liścia, aby znaleźć węzeł docelowy i punkt wstawiania. Ponieważ nie zawsze wiemy z góry, czy operacja spowoduje podział, czy scalenie (przynajmniej nie do momentu zlokalizowania docelowego węzła liścia), musimy oznaczać drogę tzw. **okruszkami** (ang. *breadcrumbs* — dosł. okruszki chleba).

Okruszki zawierają odniesienia do węzłów, przez które przeszliśmy od korzenia, i są używane do śledzenia ich w odwrotnej kolejności podczas propagacji podziałów lub scaleń. Najbardziej naturalną strukturą danych jest stos. Na przykład PostgreSQL przechowuje okruszki na stosie określanym wewnętrznie jako BTStack[2].

Jeśli węzeł zostanie podzielony lub scalony, okruszki mogą być użyte do znalezienia punktów wstawienia dla kluczy przeciągniętych do rodzica i powrotu w górę drzewa w celu propagacji zmian strukturalnych do węzłów wyższego poziomu, jeśli to konieczne. Ten stos jest przechowywany w pamięci.

Rysunek 4.8 przedstawia przykład przechodzenia od korzenia do liścia ze zbieraniem okruszków zawierających wskaźniki do odwiedzonych węzłów i indeksy komórek. Jeśli docelowy węzeł liścia jest podzielony, element na szczycie stosu jest zdejmowany w celu zlokalizowania jego bezpośredniego rodzica. Jeśli węzeł nadrzędny ma wystarczającą ilość miejsca, dołączana jest do niego nowa komórka o indeksie komórki z okruszka (zakładając, że indeks jest nadal prawidłowy). W przeciwnym razie węzeł nadrzędny również jest dzielony. Ten proces jest kontynuowany rekurencyjnie, dopóki stos nie zostanie opróżniony i nie dotrzemy do korzenia albo na danym poziomie nie będzie żadnego podziału.

[2] Więcej informacji można znaleźć w repozytorium projektu: *https://databass.dev/links/21*.



Rysunek 4.8. Okruszki zebrane podczas wyszukiwania, zawierające przemierzone węzły i indeksy komórek. Linie przerywane reprezentują logiczne łącza do odwiedzonych węzłów. Liczby w tabeli okruszków reprezentują indeksy śledzonych wskaźników potomnych

Przywracanie równowagi

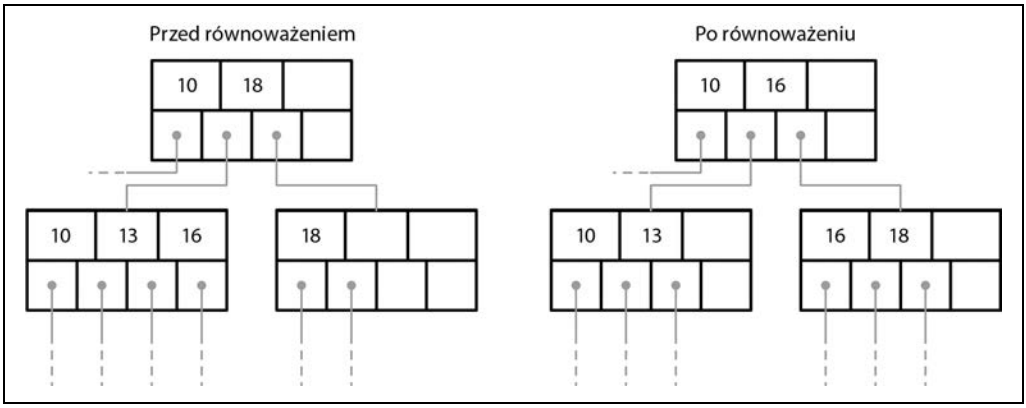
Niektóre implementacje B-drzewa próbują odłożyć operacje dzielenia i scalania, aby amortyzować ich koszty poprzez **przywracanie równowagi** elementów w obrębie poziomu lub przenoszenie przed ostatecznym podziałem lub scaleniem elementów z węzłów o większej zajętości do węzłów o mniejszej zajętości tak długo, jak to możliwe. Pomaga to poprawić zajętość węzłów i może zmniejszyć liczbę poziomów w drzewie przy potencjalnie wyższym koszcie równoważenia.

Równoważenie obciążenia może być wykonywane podczas operacji wstawiania i usuwania [GRAEFE11]. Aby poprawić wykorzystanie przestrzeni, zamiast rozdzielać węzeł przy jego przepełnieniu, możemy przenieść niektóre elementy do jednego z węzłów rodzeństwa i zrobić miejsce na wstawienie. Podobnie podczas usuwania — zamiast scalać węzły rodzeństwa, możemy przenieść niektóre elementy z sąsiednich węzłów, aby dzięki temu węzeł był co najmniej w połowie zapełniony.

B*-drzewa dystrybuują dane między sąsiednimi węzłami, dopóki oba rodzeństwa nie zostaną zapełnione [KNUTH98]. Tak że zamiast dzielić pojedynczy węzeł na dwa w połowie puste węzły, algorytm dzieli dwa węzły na trzy węzły, z których każdy jest zapełniony w dwóch trzecich. SQLite używa tego wariantu w implementacji (*https://databass.dev/links/22*). Podejście to poprawia średnią zajętość poprzez odkładanie podziałów, ale wymaga dodatkowej logiki śledzenia

i równoważenia. Wyższe wykorzystanie oznacza również bardziej wydajne wyszukiwania, ponieważ wysokość drzewa jest mniejsza i mniej stron musi zostać przeszukanych na ścieżce do przeszukiwanego liścia.

Na rysunku 4.9 pokazano dystrybucję elementów między sąsiednimi węzłami, gdzie lewy węzeł rodzeństwa zawiera więcej elementów niż prawy. Elementy z węzła zajętego w większym stopniu są przenoszone do węzła mniej zajętego. Ponieważ równoważenie zmienia niezmiennik min/max węzłów rodzeństwa, to aby go zachować, musimy zaktualizować klucze i wskaźniki w węźle nadrzędnym.



Rysunek 4.9. Równoważenie B-drzewa: rozdzielanie elementów między węzłem zajętym w większym stopniu i węzłem zajętym w mniejszym stopniu

Równoważenie obciążenia jest przydatną techniką stosowaną w wielu implementacjach baz danych. Na przykład SQLite implementuje algorytm **równoważenia rodzeństwa** (ang. *balance-siblings*) (*https://databass.dev/links/23*), który jest nieco zbliżony do tego, co opisaliśmy w tym punkcie. Równoważenie może dodać pewną złożoność do kodu, ale ponieważ jego przypadki użycia są odizolowane, może być zastosowane na późniejszym etapie jako optymalizacja.

Dołączanie tylko z prawej strony

Wiele systemów baz danych automatycznie wykorzystuje monotonicznie zwiększane wartości jako główne klucze indeksów. Ten przypadek otwiera możliwość optymalizacji, ponieważ wszystkie wstawienia mają miejsce na końcu indeksu (w skrajnym prawym liściu), więc większość podziałów występuje w skrajnym prawym węźle na każdym poziomie. Ponadto klucze są zwiększane monotonicznie, a zatem biorąc pod uwagę, że liczba wykonywanych dołączeń w stosunku do aktualizacji i usunięć jest mała, strony niebędące liśćmi są również mniej pofragmentowane niż w przypadku kluczy ustawionych losowo.

PostgreSQL nazywa ten przypadek **szybką ścieżką** (*https://databass.dev/links/24*). Gdy wstawiony klucz jest większy od pierwszego klucza po skrajnie prawej stronie, a ta ma wystarczająco dużo miejsca, aby pomieścić nowo wstawiony wpis, nowy wpis jest wstawiany w odpowiednim miejscu w buforowanym prawym liściu i cała ścieżka odczytu może zostać pominięta.

SQLite ma podobną koncepcję i nazywa ją **szybkim równoważeniem** (ang. *quickbalance*) (*https://databass.dev/links/25*). Gdy wpis jest wstawiany na prawym końcu, a węzeł docelowy jest pełny (tj. staje się największym wpisem w drzewie po wstawieniu), zamiast ponownego równoważenia lub dzielenia węzła alokowany jest nowy skrajny prawy węzeł i dodawany jest jego wskaźnik do rodzica (więcej informacji na temat implementacji równoważenia w SQLite można znaleźć w podrozdziale „Przywracanie równowagi" wcześniej w tym rozdziale). Mimo że pozostawia to nowo utworzoną stronę prawie pustą (zamiast w połowie pustej w przypadku podziału węzła), jest bardzo prawdopodobne, że węzeł wkrótce się zapełni.

Ładowanie masowe

Jeśli mamy wstępnie posortowane dane i chcemy je załadować masowo lub musimy przebudować drzewo (na przykład w celu defragmentacji), możemy jeszcze bardziej rozwinąć koncepcję dołączania tylko z prawej strony. Ponieważ dane wymagane do utworzenia drzewa są już posortowane, podczas masowego ładowania musimy tylko dołączyć elementy w najbardziej wysuniętej na prawo lokalizacji w drzewie.

W tym przypadku możemy całkowicie uniknąć podziałów i scaleń i skomponować drzewo od dołu do góry, wypisując je poziom po poziomie lub wypisując węzły wyższego poziomu, gdy tylko mamy wystarczającą liczbę wskaźników do już zapisanych węzłów niższego poziomu.

Jednym z podejść do implementacji masowego ładowania jest zapisywanie danych wstępnie posortowanych według stron na poziomie liścia (zamiast wstawiania poszczególnych elementów). Po zapisaniu strony liścia propagujemy jej pierwszy klucz do rodzica i korzystamy z normalnego algorytmu do budowania wyższych poziomów B-drzewa [RAMAKRISHNAN03]. Dołączane klucze są posortowane, zatem wszystkie podziały w tym przypadku występują w skrajnym prawym węźle.

Ponieważ B-drzewa są zawsze budowane od najniższego poziomu (liścia), kompletny poziom liścia może zostać wypisany przed utworzeniem węzłów wyższego poziomu. Dzięki temu można mieć wszystkie wskaźniki potomne pod ręką do czasu skonstruowania wyższych poziomów. Główne korzyści płynące z takiego rozwiązania są takie, że nie musimy wykonywać żadnych podziałów ani scalania na dysku, a w pamięci na czas tworzenia musimy przechowywać tylko minimalną część drzewa (tj. wszystkich rodziców aktualnie wypełnianego węzła liścia).

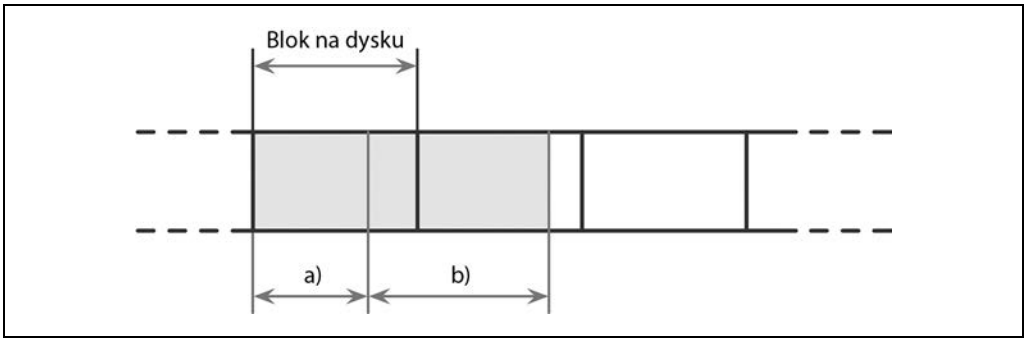
Niezmienne B-drzewa mogą być tworzone w taki sam sposób, ale w przeciwieństwie do zmiennych B-drzew nie wymagają dodatkowej przestrzeni dla późniejszych modyfikacji, ponieważ wszystkie operacje na drzewie są ostateczne. Wszystkie strony mogą być całkowicie zapełnione, co poprawia zajętość i przekłada się na lepszą wydajność.

Kompresja

Przechowywanie surowych, nieskompresowanych danych może powodować znaczne koszty, ale wiele baz danych oferuje różne sposoby kompresji w celu zaoszczędzenia miejsca. Oczywisty kompromis dotyczy tutaj szybkości dostępu i współczynnika kompresji: większe współczynniki kompresji mogą poprawić rozmiar danych, co pozwala pobrać większą ilość danych w jednym dostępie, ale może to wymagać więcej pamięci RAM i cykli procesora do ich kompresji i dekompresji.

Kompresja może być dokonywana na różnych poziomach szczegółowości. Nawet jeśli dotyczy całych plików i może dać lepsze współczynniki kompresji, ma ograniczone zastosowanie, ponieważ cały plik musi zostać zdekompresowany przy aktualizacji, a bardziej szczegółowa kompresja jest zwykle lepsza dla większych zbiorów danych. Kompresja całego pliku indeksu jest zarówno niepraktyczna, jak i trudna do efektywnego wdrożenia: aby zaadresować konkretną stronę, cały plik (lub jego część zawierająca metadane kompresji) musi być dostępny (w celu zlokalizowania części skompresowanej), zdekompresowany i udostępniony.

Inną możliwością jest kompresja danych według stron. To dobrze pasuje do naszej dyskusji, ponieważ w algorytmach, które omawialiśmy do tej pory, używane były strony o stałym rozmiarze. Strony mogą być kompresowane i dekompresowane niezależnie od siebie, co pozwala na połączenie kompresji z ładowaniem i zapisywaniem strony w pamięci. Jednak skompresowana strona w tym przypadku może zajmować tylko ułamek bloku dyskowego, a ponieważ transfery są zwykle wykonywane w jednostkach wielkości bloków dyskowych, może być konieczny transfer dodatkowych bajtów [RAY95]. Na rysunku 4.10 można zobaczyć skompresowaną stronę (a) zajmującą mniej miejsca niż blok dysku. Gdy ładujemy tę stronę, przesyłamy również dodatkowe bajty, które należą do drugiej strony. W przypadku stron, które obejmują wiele bloków dysku, jak na tym samym rysunku (b), musimy odczytać dodatkowy blok.



Rysunek 4.10. Kompresja i wypełnianie bloków

Innym podejściem jest kompresja samych danych — wierszowa (kompresja całych rekordów danych) albo kolumnowa (kompresja poszczególnych kolumn). W tym przypadku zarządzanie stronami i kompresja są oddzielone.

Większość baz danych z otwartym kodem źródłowym, które zostały przeanalizowane podczas pisania tej książki, zawiera metody kompresji z wykorzystaniem dostępnych bibliotek, takich jak Snappy (*https://databass.dev/links/26*), zLib (*https://databass.dev/links/27*), lz4 (*https://databass.dev/links/28*) i wiele innych.

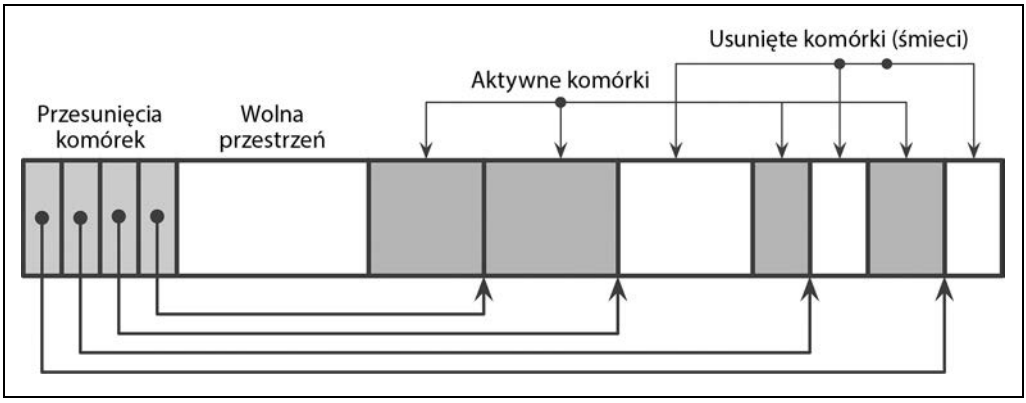
Ponieważ algorytmy kompresji dają różne wyniki w zależności od zbioru danych i potencjalnych celów (takich jak na przykład współczynnik kompresji, wydajność lub narzut pamięci), w tej książce nie będziemy zagłębiać się w porównywanie implementacji i ich szczegóły. Istnieje wiele dostępnych opracowań, które oceniają różne algorytmy kompresji dla różnych rozmiarów bloków (na przykład Squash Compression Benchmark (*https://databass.dev/links/29*)), zwykle

koncentrując się na czterech metrykach: narzut pamięci, wydajność kompresji, wydajność dekompresji i współczynnik kompresji. Te wskaźniki są ważne i trzeba je wziąć pod uwagę przy wyborze biblioteki kompresji.

Odkurzanie i konserwacja

Do tej pory rozmawialiśmy głównie o operacjach związanych z użytkownikami B-drzew. Jednakże istnieją też inne procesy, zachodzące równolegle z zapytaniami, które utrzymują integralność, odzyskują przestrzeń, zmniejszają obciążenie i utrzymują strony w kolejności. Wykonywanie tych operacji w tle pozwala nam zaoszczędzić trochę czasu i uniknąć płacenia ceny za czyszczenie podczas wstawiania, aktualizacji i usuwania.

Opisany projekt stron podzielonych na obszary (patrz podrozdział „Strony podzielone na obszary" w rozdziale 3) wymaga konserwacji stron w celu utrzymania ich w dobrym stanie. Na przykład późniejsze podziały i scalenia w węzłach wewnętrznych lub wstawienia, aktualizacje i usunięcia na poziomie liści mogą spowodować, że strona będzie miała wystarczającą ilość przestrzeni *logicznej*, ale nie będzie miała wystarczającej ilości przestrzeni *ciągłej*, ponieważ będzie pofragmentowana. Na rysunku 4.11 pokazano przykład takiej sytuacji: strona ma jeszcze dostępną przestrzeń logiczną, ale jest pofragmentowana i podzielona między dwa usunięte (śmieciowe) rekordy i wolną przestrzeń pozostającą między wskaźnikami nagłówka/komórki i komórkami.



Rysunek 4.11. Przykład pofragmentowanej strony

Nawigacja po B-drzewach odbywa się od poziomu korzenia. Rekordy danych, do których można dotrzeć, idąc za wskaźnikami w dół od węzła głównego, są **aktywne** (adresowalne). Nieadresowalne rekordy danych są nazywane **śmieciami**: te rekordy nie są nigdzie przywoływane i nie mogą być odczytywane ani interpretowane, więc ich zawartość jest praktycznie nieważna.

To rozróżnienie można zobaczyć na rysunku 4.11: komórki, do których wskaźniki nadal istnieją, są adresowalne, w przeciwieństwie do komórek usuniętych lub nadpisanych. Wypełnianie zerami obszarów śmieci jest często pomijane ze względu na wydajność, ponieważ ostatecznie te obszary i tak są nadpisywane przez nowe dane.

Fragmentacja spowodowana aktualizacjami i usunięciami

Zastanówmy się, w jakich okolicznościach strony wchodzą w stan, w którym mają nieadresowalne dane i muszą zostać skompaktowane. Usunięcia na poziomie liści kasują tylko komórki z nagłówka, a sama komórka pozostaje nienaruszona. Po wykonaniu tej czynności komórka nie jest już *adresowalna*, jej zawartość nie pojawi się w wynikach zapytania, a wykasowanie zawartości lub przenoszenie sąsiednich komórek nie jest konieczne.

Gdy strona jest dzielona, przycinane są tylko przesunięcia, a ponieważ reszta strony nie jest adresowalna, komórki, których przesunięcia zostały obcięte, nie są osiągalne, więc zostaną nadpisane za każdym razem, gdy pojawią się nowe dane, lub zostaną odzyskane ze śmieci, gdy zostanie uruchomiony proces odkurzania.

W niektórych bazach danych wykorzystywane jest odśmiecanie. Bazy te pozostawiają usunięte i zaktualizowane komórki w celu kontroli współbieżności w wielu wersjach (patrz punkt „Wielowersyjna kontrola współbieżności" w rozdziale 5). Komórki pozostają dostępne dla współbieżnie wykonujących się transakcji do momentu zakończenia aktualizacji i mogą zostać odzyskane, gdy żaden inny wątek nie uzyska do nich dostępu. Niektóre bazy danych utrzymują struktury śledzące **rekordy-widma**, które są odzyskiwane, gdy wszystkie transakcje, które mogły je zobaczyć, zostają zakończone [WEIKUM01].

Ponieważ usuwanie odrzuca tylko przesunięcia komórek i nie przenosi pozostałych komórek ani fizycznie nie usuwa komórek docelowych, aby zająć zwolnione miejsce, zwolnione bajty mogą zostać rozproszone na całej stronie. W takim przypadku mówimy, że strona jest **pofragmentowana** i wymaga defragmentacji.

Aby dokonać zapisu, często potrzebujemy ciągłego bloku wolnych bajtów, w którym zmieści się komórka. By ponownie połączyć uwolnione fragmenty i naprawić tę sytuację, musimy *przepisać* stronę.

Operacje wstawiania zostawiają krotki w tej samej kolejności, w której zostały wstawione. Nie ma to tak znaczącego wpływu, ale naturalnie posortowane krotki mogą pomóc we wstępnym pobieraniu pamięci podręcznej podczas kolejnych odczytów.

Aktualizacje dotyczą głównie poziomu liści: wewnętrzne klucze stron są używane do nawigacji i definiują jedynie granice poddrzewa. Dodatkowo aktualizacje są wykonywane na podstawie poszczególnych kluczy i generalnie nie powodują zmian strukturalnych w drzewie, poza tworzeniem stron przepełnienia. Jednak na poziomie liści operacje aktualizacji nie zmieniają kolejności komórek i starają się unikać przepisywania stron. Oznacza to, że ostatecznie może zostać zapisanych wiele wersji komórki, z których tylko jedna będzie adresowalna.

Defragmentacja stron

Proces, w którym jest odzyskiwana przestrzeń i przepisywane są strony, nazywany jest **kompaktowaniem**, **odkurzaniem** (ang. *vacuum*) lub po prostu **konserwacją**. Przepisywanie stron może być wykonywane synchronicznie przy zapisie, jeśli strona nie ma wystarczającej ilości wolnego

miejsca (aby uniknąć tworzenia niepotrzebnych stron przepełnienia), ale kompaktowanie jest najczęściej określane jako odrębny asynchroniczny proces przechodzenia przez strony, odśmiecania i przepisywania ich zawartości.

Proces ten odzyskuje przestrzeń zajmowaną przez nieaktywne komórki i przepisuje komórki w ich logicznej kolejności. Gdy strony są przepisywane, mogą również zostać przeniesione na nowe pozycje w pliku. Nieużywane strony w pamięci stają się dostępne i są zwracane do pamięci podręcznej stron. Identyfikatory nowo dostępnych stron na dysku są dodawane do **listy wolnych stron** (ang. *free page list*, czasami nazywanej *freelist*[3]). Informacje te muszą być przechowywane, aby przetrwać awarie i ponowne uruchamianie węzła oraz aby zagwarantować, że wolne miejsce nie zostanie utracone lub nie wycieknie.

Podsumowanie

W tym rozdziale omówiliśmy koncepcje charakterystyczne dla implementacji B-drzewa na dysku, takie jak:

Nagłówek strony
: Jakie informacje są w nim zwykle przechowywane.

Wskaźniki skrajne prawe
: Są one niesparowane z kluczami oddzielającymi i uczymy się, jak sobie z nimi radzić.

Duże klucze
: Określają maksymalny dozwolony klucz, który może być przechowywany w węźle.

Strony przepełnienia
: Umożliwiają przechowywanie rekordów o zbyt dużym i zmiennym rozmiarze przy użyciu stron o stałym rozmiarze.

Omówiliśmy też kilka szczegółów związanych z przejściami od korzenia do liścia:

- Jak wykonać wyszukiwanie binarne za pomocą wskaźników pośrednich.
- Jak śledzić hierarchie drzew przy użyciu wskaźników nadrzędnych lub okruszków.

Na koniec omówiliśmy kilka technik optymalizacji i konserwacji:

Przywracanie równowagi
: Przenosi elementy między sąsiednimi węzłami, aby zmniejszyć liczbę podziałów i scaleń.

Dołączanie tylko z prawej strony
: Dołącza nową komórkę najbardziej na prawo, zamiast ją dzielić, przy założeniu, że szybko się zapełni.

[3] Na przykład SQLite utrzymuje listę stron (*https://databass.dev/links/30*), które nie są używane przez bazę danych, gdzie strony *pnia* są przechowywane na połączonej liście i przechowują adresy zwolnionych stron.

Ładowanie masowe

Technika wydajnego budowania od podstaw B-drzew z posortowanych danych.

Zbieranie śmieci

Proces, który przepisuje strony, ustawia komórki w kolejności klucza i odzyskuje miejsce zajmowane przez nieadresowalne komórki.

Koncepcje te powinny wypełnić lukę między podstawowym algorytmem B-drzewa a rzeczywistą implementacją i pomóc lepiej zrozumieć, jak działają systemy przechowywania oparte na B-drzewach.

Dalsze lektury

Więcej o koncepcjach wspomnianych w tym rozdziale można dowiedzieć się z następujących źródeł:

B-drzewa oparte na dysku

Goetz Graefe, *Modern B-Tree Techniques*, „Foundations and Trends in Databases", 3, 2011, nr 4 (kwiecień), s. 203 – 402, *https://doi.org/10.1561/1900000028*.

Christopher G. Healey, *Disk-Based Algorithms for Big Data (1st Ed.)*, CRC Press, Boca Raton 2016.

Aby wybrać odpowiednie narzędzie do pracy, musisz zrozumieć idee i algorytmy stojące za ich projektem.

Michael Klishin, współpracownik RabbitMQ

W ciągu ostatnich 15 lat powstało tak wiele baz danych i narzędzi, że łatwo się pogubić, jeśli próbuje się zrozumieć przypadki użycia, szczegóły i specyfiki. Większość opracowań na temat systemów baz danych nie opisuje implementacji mechanizmu pamięci masowej. Tymczasem znajomość tych wewnętrznych aspektów jest bardzo ważna dla programistów, inżynierów, architektów i menedżerów.

Ta książka ułatwi Ci zgłębienie koncepcji kryjących się za działaniem nowoczesnych baz danych. Dzięki niej zrozumiesz, w jaki sposób struktury dyskowe różnią się od tych w pamięci i jak działają algorytmy efektywnego utrzymywania struktur B-drzewa na dysku. Poznasz implementacje pamięci masowej o strukturze dziennika. Znajdziesz tu również wyjaśnienie zasad organizacji węzłów w klaster baz danych i specyfiki środowisk rozproszonych. Dowiesz się, jak algorytmy rozproszone poprawiają wydajność i stabilność systemu i jak uzyskać ostateczną spójność danych. Ponadto w książce zaprezentowano koncepcje antyentropii i plotek, służące do zapewniania zbieżności i rozpowszechniania danych, a także mechanizm transakcji utrzymujący spójność logiczną bazy.

Obowiązkowa lektura dla każdego, kto korzysta z jakiejkolwiek bazy danych!

Nate McCall, przewodniczący PMC

Najważniejsze zagadnienia:

- klasyfikacja i taksonomia pamięci masowej
- silniki pamięci masowej oparte na B-drzewie i niezmienna struktura dziennika
- struktura plików bazy danych
- pamięć podręczna stron i pule buforów
- systemy rozproszone: złożone wzorce komunikacji węzłów i procesów
- klastry baz danych

Alex Petrov jest inżynierem infrastruktury danych, entuzjastą baz danych i systemów pamięci masowej. Jako członek PMC aktywnie wspiera rozwój projektu Apache Cassandra. Specjalizuje się w pamięciach masowych, systemach rozproszonych i algorytmach.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1332-5
9 788328 913325
Cena: 89,00 zł