Politechnika Wrocławska

Dziedzina: Dziedzina nauk inżynieryjno-technicznych
Dyscyplina: informatyka techniczna i telekomunikacja

Rozprawa Doktorska

INDEKSOWANIE BAZ DANYCH NA NOWOCZESNYCH TYPACH PAMIĘCI

Mgr inż. Michał Kukowski

promotor: prof. dr hab. Jacek Cichoń
promotor pomocniczy: dr inż. Wojciech Macyna

Słowa kluczowe: bazy danych, flash, ssd, pcm, indeksowanie danych

Wrocław 2023

Streszczenie

Praca poświęcona jest indeksowaniu baz danych na różnych nowoczesnych typach pamięci, takich jak pojedyncze kości flash, dyski SSD oraz pamięć zmiennofazowa PCM. W dzisiejszych czasach indeks to nie tylko prosta struktura danych wyposażona w podstawowe procedury, to często duży, rozbudowany system, który uwzględnia wiele czynników m.in. charakterystykę pamięci. Efektywny sposób indeksowania rekordów nie tylko powinien minimalizować czas potrzebny na wykonanie kwerendy, ale także powinien wydłużać czas życia nowych nośników pamięci. W hurtowniach danych codziennie zapisuje się terabajty informacji. W tym kontekście, minimalizacja zużycia pamięci jest bardzo istotna nie tylko z punktu widzenia ekonomicznego, ale także z ekologicznego.

Pierwszym głównym celem pracy jest analiza popularnych algorytmów i systemów wykorzystywanych w silnikach baz danych do indeksowania rekordów. Drugim celem jest opracowanie nowych metod indeksowania oraz implementacja zaprojektowanych struktur danych. W pracy zaproponowano kilka nowych algorytmów i systemów porządkowania danych, realizujących indeksowanie rekordów. Każdy z algorytmów przeanalizowano, dołączając analizę teoretyczną złożoności obliczeniowej oraz dowód poprawności.

W rozdziale **5** omówiono algorytmy indeksujące rekordy na pamięci flash. Przedstawiono także nowy algorytm indeksowania FA-Tree (ang. Flash Aware Tree) [1], który osiągnął 20 krotnie lepszy czas obsługi kwerendy od popularnego drzewa B+, a także zużył 6 krotnie mniej pamięci.

W rozdziale **6** omówiono sposoby indeksowania danych zapisanych na dyskach SSD. Przedstawiono nowe algorytmy indeksowania danych takie jak wierszowy indeks FALSM-Tree (ang. Flash Aware LSM-Tree) [2], który znacząco poprawia osiągi bazy danych podczas wstawiania wielu elementów do indeksu zapisanego na dysku SSD. Podczas eksperymentów FALSM-Tree osiągnął 6 krotnie lepszy czas oraz zapisał 6 razy mniej bajtów niż klasyczny indeks LSM. W tym samym rozdziale omówiono także całkiem nowy kolumnowy indeks CFT (ang. Columned FD-Tree) [3]. Mimo kolumnowego ułożenia, indeks CFT osiągał czas modyfikacji struktury mocno zbliżony do czasu wykonania modyfikacji przez oryginalne wierszowe podejście z wykorzystaniem drzewa FD, jednocześnie znacząco poprawił czas obsługi kwerend wyszukujących. Również w rozdziale **6** przedstawiono nowy mechanizm indeksowania częściowego LAM (ang. Lazy Adaptive Merging) [4]. System ten jest w pełni przystosowany do charakterystyki dysków SSD. Nie tylko osiągał 2 krotnie lepsze wyniki od algorytmu AM podczas tworzenia samego indeksu, ale także wykonywał kwerendy modyfikujące tabelę średnio o 15% szybciej.

W rozdziale **7** omówiono indeksowanie baz danych używających pamięć zmiennofazową PCM do przechowywania danych. Przedstawiono także nową strukturę danych BB+-Tree (ang. Buffered B+-Tree) oraz nowy sposób częściowego indeksowania dostosowanego do pamięci PCM - PAM (ang. PCM Adaptive Merging) [5]. Wyniki eksperymentów pokazały ogromną przewagę nowego przedstawionego systemu nad oryginalnym algorytmem

Adaptive Merging. Algorytm PAM stworzył indeks nawet 5 krotnie szybciej, modyfikując także 5 krotnie mniej pamięci, co znacząco wydłużyło żywotność tego nośnika pamięci.

Abstract

The Phd thesis is devoted to indexing databases on various modern types of memory, such as single flash memory chips, solid-state drives (SSDs), and phase-change memory (PCM). Nowadays, an index is not just a simple data structure equipped with basic procedures, but often a large, complex system that takes into account many factors, including memory characteristics. An efficient way of indexing records should not only minimize the time required to execute a query but also extend the lifespan of new memory media. In data warehouses, terabytes of information are written every day. In this context, minimizing memory usage is essential not only from an economic point of view but also from an ecological one.

The first main goal of the paper is to analyze popular algorithms and systems used in database engines for indexing records. The second goal is to develop new indexing methods and implement designed data structures. Several new algorithms and data ordering systems were proposed in the paper, realizing record indexing. Each of the algorithms was analyzed, including theoretical computational complexity analysis and proof of correctness.

Chapter **5** discusses algorithms for indexing records on flash memory. A new indexing algorithm, FA-Tree (Flash Aware Tree) [1], is also presented, which achieved a 20-fold improvement in query handling time over the popular B+ tree and used six times less memory.

Chapter **6** discusses ways of indexing data stored on SSDs. New data indexing algorithms, such as row-based FALSM-Tree (Flash Aware LSM-Tree) [2], are presented, which significantly improve database performance when inserting multiple elements into an index stored on an SSD. During experiments, FALSM-Tree achieved six times better time and wrote six times fewer bytes than the classical LSM index. The same chapter also presents a completely new column-based CFT (Columned FD-Tree) index [3] Despite its column-based layout, the CFT index achieved modification time comparable to the time of modification using the original row-based approach with the use of the FD tree, while significantly improving the query search time. Chapter **6** also presents a new partial indexing mechanism called LAM (Lazy Adaptive Merging) [4] This system is fully adapted to the characteristics of SSDs. It not only achieved twice as good results as the AM algorithm when creating the index itself but also executed query modifications to the table on average 15% faster.

Chapter **7** discusses indexing databases using PCM to store data. A new data structure, BB+-Tree (Buffered B+-Tree), and a new partial indexing method adapted to PCM memory - PAM (PCM Adaptive Merging) [5] - are also presented. The results of the experiments showed a huge advantage of the new system over the original Adaptive Merging algorithm. The PAM algorithm created an index even five times faster, while modifying five times less memory, significantly extending the lifespan of this memory medium.

Spis treści

Wstęp

Praca poświęcona jest indeksowaniu baz danych na nowoczesnych typach pamięci. Za nowoczesny model pamięci w tej pracy uznajemy: pojedyncze kości flash, dyski SSD oraz pamięć zmiennofazową PCM. Sposób przechowywania danych oraz zarządzania nimi podczas indeksowania, jak również przenoszenie, defragmentacja czy buforowanie musi uwzględniać ograniczenia wynikające z funkcjonowania nośnika pamięci, na której zapisane są rekordy. Dotyczy to w szczególności ograniczeń liczby możliwych usunięć danych z komórek pamięci, a także asymetrii pomiędzy kosztami zapisu i odczytu. Efektywny sposób indeksowania rekordów nie tylko powinien minimalizować czas potrzebny na wykonanie kwerendy, ale także powinien minimalizować zużycie nowoczesnych modeli pamięci [6], [7]. Algorytm indeksowania musi być także bezpieczny. Oznacza to, że wszystkie dane, które są w nim zapisane muszą zostać nienaruszone podczas awarii systemu. Do przechowywania baz danych zazwyczaj używa się nośników trwałych takich jak dyski HDD, SDD czy PCM. Gdy odłączymy zasilanie, dane nie ulegają degradacji. Jednak algorytmy indeksujące, bardzo często, aby przyspieszyć operacje wykonywane na strukturze danych, używają różnych technik buforowania, zbierania statystyk i potrzebnych metadanych [8]. Informacje te zapisywane są w ulotnej pamięci RAM, a więc w chwili awarii systemu, dane te są tracone. Zatem bardzo ważnym aspektem takich algorytmów jest zdolność przywrócenia stanu przed awarią (ang. crash recovery), co zazwyczaj realizowane jest przez odtwarzanie wszelkich utraconych informacji na podstawie innych informacji zapisanych na dysku.

W hurtowniach danych z setkami tabel i tysiącami kolumn istnieje wiele sposobów na indeksowanie bazy danych. Zbyt mała liczba indeksów powoduje kosztowne skany nieposortowanych partii danych za każdym razem, zaś zbyt duża ich liczba powoduje czasowy narzut na aktualizacje podczas dodawania i usuwania rekordów. Ciężko jest idealnie zaprojektować bazę danych, znajdując idealną kombinację indeksów, ponieważ nie jesteśmy w stanie przewidzieć wszystkich potrzeb użytkownika [9]. Obecnie stosuje się 3 główne podejścia do indeksowania rekordów: wierszowe, kolumnowe i częściowe. Wybór techniki indeksowania zależy od wielu czynników: pamięci, na której przechowywane są dane, liczby kolumn w rekordzie, wielkości bazy danych oraz dominującego wzorca kwerend [10], [11], [12]. Wierszowy sposób indeksowania zapisuje rekordy obok siebie, jeden po drugim. Ten sposób zapisu wykorzystywany jest, gdy w użyciu bazy danych przeważają modyfikacje tabeli. Każdy zapis wykonuje pojedynczą operację na dysku, ponieważ cały rekord przechowywany jest w ciągłym obszarze pamięci. Kolumnowe podejście stosuje się wtedy, gdy rekord składa się z wielu kolumn, ale tylko kilka z nich jest używanych w jednej kwerendzie. W takim przypadku, pojedynczy rekord jest zapisany w wielu miejscach, a kolumny kolejnych rekordów przechowywane są w ciągłych obszarach pamięci. Tak więc jeśli chcemy wczytać wybrane kolumny dla wielu kluczy, to odczytujemy tylko potrzebne segmenty dysku. Niestety, gdy chcemy zapisać nowy rekord, zamiast pojedynczego zapisu, mamy tyle zapisów, ile kolumn w rekordzie, ponieważ każda kolumna to osobny obszar fi-

zycznej pamięci. Zatem ten sposób przechowywania danych używa się wtedy, gdy odczyty są dominującą operacją wykonywaną na tabeli. Trzeci sposób, indeksowanie częściowe, jest najmłodszym sposobem porządkowania danych. Polega on na indeksowaniu tylko tej części rekordów, która jest potrzebna lub będzie potrzebna w najbliższym czasie. Idealnie sprawdza się w ogromnych bazach danych, w których na co dzień korzysta się z niewielkiego podzbioru rekordów. Aby zminimalizować czasowy narzut wynikający z utrzymania indeksu, rekordy dodawane i usuwane są z indeksu automatycznie, w zależności od ich przydatności. Zazwyczaj nowe dane są wstawiane natychmiastowo do indeksu, a dane o które poprosił użytkownik w kwerendzie, gdy nie były w indeksie, są kopiowane do niego, aby ich ponowny odczyt był o wiele szybszy. Cały mechanizm częściowego indeksowania można porównać do buforowania danych w pamięci podręcznej komputera (ang. cache memory).

Celem pracy jest analiza dotychczasowych algorytmów i struktur danych wykorzystywanych do indeksowania relacyjnych baz danych, opracowanie nowych metod indeksowania oraz implementacja zaproponowanych struktur danych. W pracy zaproponowano kilka nowych algorytmów i systemów porządkowania danych realizujących indeksowanie rekordów.

Struktura pracy jest następująca. W rozdziale **2** przeanalizujemy dokładnie różne sposoby indeksowania baz danych, ich wady i zalety, a także spróbujemy sformułować problemy do rozwiązania wynikające z przechowywania różnych typów indeksów na nowoczesnych modelach pamięci trwałej. W rozdziale **3** omówimy funkcjonowanie pamięci flash, dysków HDD, SDD oraz PCM, a także sposoby zarządzania tymi pamięciami dostępne z poziomu systemu operacyjnego Linux. W tym rozdziale, również przeprowadzimy dogłębną analizę charakterystyk każdej z poszczególnych pamięci oraz jej wpływ na funkcjonowanie indeksów pracujących na tych pamięciach. W rozdziale **4** omówimy całą platformę testową, która została zaimplementowana z myślą o wprowadzeniu jednolitego, deterministycznego środowiska do przeprowadzenia wszystkich potrzebnych eksperymentów umieszczonych w tej pracy. W skład platformy testowej wchodzi nowatorski symulator SIPS (ang. Storage and Index Performance Simulator), który nie tylko odpowiedzialny jest za symulację modeli pamięci, ale także integruje algorytmy indeksujące bazę danych z symulowanym modelem pamięci. Dzięki temu program potrafi monitorować stan dysków, indeksów oraz kwerend, a co za tym idzie, dostarczać wszelkie niezbędne dane do przeprowadzenia analizy każdego z przedstawionych w tej pracy algorytmów. Platforma testowa zawiera także 3 zestawy kwerend przygotowane na podstawie najbardziej popularnych i rzetelnych zestawów eksperymentalnych jak TPC-C i TPC-H. Wszystkie kwerendy testowe zostały dostosowane do warunków symulatora oraz implementacji algorytmów indeksujących, które nie są zintegrowane z silnikiem baz danych, ponieważ celem eksperymentów jest sprawdzenie wydajności samego sposobu indeksowania, a nie wpływu optymalizacji silnika baz danych na efektywność wykonywania kwerend. W rozdziale **5** omówimy algorytmy indeksujące dane, przechowywane na pojedynczych kościach flash. Najczęściej takie bazy danych stosowane są w urządzeniach wbudowanych (ang. embedded devices) i w tzw. urządzeniach internetu rzeczy (ang. IoT - Internet of Things), które ze względu na koszty produkcji używają kości flash zamiast dysków SSD. W tym samym rozdziale przedstawimy nowy algorytm indeksowania na pamięci flash FA-Tree (ang. Flash Aware Tree) [1], przeanalizujemy jego działanie oraz omówimy wyniki eksperymentów przeprowadzonych z użyciem platformy testowej. Zaproponowana struktura danych FA-Tree osiągnęła nawet 20 krotnie lepszy czas obsługi kwerend zorientowanych

na modyfikacje bazy od klasycznego indeksu opartego na drzewie B+. Jednocześnie indeks FA-Tree zużył 6 krotnie mniej pamięci od drzewa B+. Nowa struktura została również porównania z popularnym indeksem LSM-Tree, który ze względu na duże zużycie pamięci operacyjnej nie jest często wykorzystywany w urządzeniach wbudowanych z ograniczonymi zasobami sprzętowymi (pamięć operacyjna, moc obliczeniowa). W rozdziale **6** omówimy sposoby indeksowania danych, zapisanych na dyskach SSD. Przedstawimy nowe algorytmy indeksowania danych jak wierszowy indeks FALSM-Tree (ang. Flash Aware LSM-Tree) [2], czyli modyfikację drzewa LSM, w pełni dostosowaną do dysków SSD jak i wspierającą ważną w ostatnich czasach operację dodawania zbiorczego rekordów (ang. bulkload). Dzięki wprowadzeniu nowej metody dodawania rekordów, czas obsługi takiej kwerendy został zredukowany aż 6 krotnie względem oryginalnego indeksu LSM. Warto również wspomnieć, że testy na zestawach TPC-C wykazały, że nowy algorytm nadpisuje nawet 6 krotnie mniej pamięci, a więc znacząco wydłuża czas życia dysków SSD. W tym samym rozdziale omówimy także całkiem nowy kolumnowy indeks CF-Tree (ang. Column Flash Tree) [3]. Indeks ten jest zbiorem struktur danych typu FD-Tree połączonych ze sobą specjalnie zaprojektowanymi metadanymi. Dzięki zaproponowanemu algorytmowi scalania atrybutów w rekord, indeks CFT w czasie $O(\log n)$ jest w stanie dla danego atrybutu zwrócić odpowiadający mu inny atrybut lub cały rekord. Tak szybki algorytm scalający kolumny jest niezbędny, aby uznać nową metodę indeksowania za przydatną na obecnym rynku baz danych. W rozdziale **6** przedstawiono pełny dowód oraz analizę złożoności obliczeniowej algorytmu scalającego. Dodatkowo przeprowadzono liczne eksperymenty pokazujące realną przydatność nowej metody. Mimo kolumnowego ułożenia, indeks CFT osiąga czas modyfikacji struktury mocno zbliżony do czasu wykonania modyfikacji przez oryginalne wierszowe podejście z wykorzystaniem drzewa FD. Maksymalne odchylenie na niekorzyść kolumnowego indeksu wynosiło 6%. Oczywistym faktem jest, iż kolumnowe podejście znacznie szybciej obsługiwało kwerendy wyszukujące zakres kluczy, gdy użytkownik nie potrzebował wszystkich kolumn. Czas obsługi takiej kwerendy przez strukturę CFT malał liniowo wraz z malejącą liczbą bajtów jakie musiał wczytać by wykonać zapytanie od użytkownika. W tym samym rozdziale przedstawimy nowy mechanizm indeksowania częściowego LAM (ang. Lazy Adaptive Merging) [4]. Mimo, iż nowy algorytm korzysta z idei tworzenia indeksu podobnej do oryginalnego Adaptive Merging [13], to jest to całkiem nowy system w pełni przystosowany do charakterystyki dysków SSD. Zaproponowany system LAM buduje indeks 2 razy szybciej niż oryginalny algorytm AM. Dodatkowo obsługuje on o 15% szybciej wszelkie kwerendy modyfikujące tabelę, jak dodawanie lub usuwanie rekordów. W rozdziale **7** omówimy indeksowanie baz danych używających pamięci zmiennofazowej PCM (ang. Phase Change Memory) do przechowywania danych. Przedstawimy także nową strukturę danych BB+-Tree (ang. Buffered B+-Tree) oraz nowy sposób częściowego indeksowania dostosowanego do pamięci PCM - PAM (ang. PCM Adaptive Merging) [5]. Tak jak we wcześniej wymienionych rozdziałach, przeanalizujemy także eksperymenty przeprowadzone na nowych metodach indeksowania. Wyniki pokazują ogromną przewagę nowego przedstawionego systemu nad oryginalnym algorytmem Adaptive Merging. Algorytm PAM tworzy indeks nawet 5 krotnie szybciej, modyfikując także 5 krotnie mniej pamięci, co znacząco wydłuży żywotność nośnika danych. W ostatnim rozdziale **8** podsumujemy całą pracę oraz przedstawimy możliwe dalsze kierunki badań.

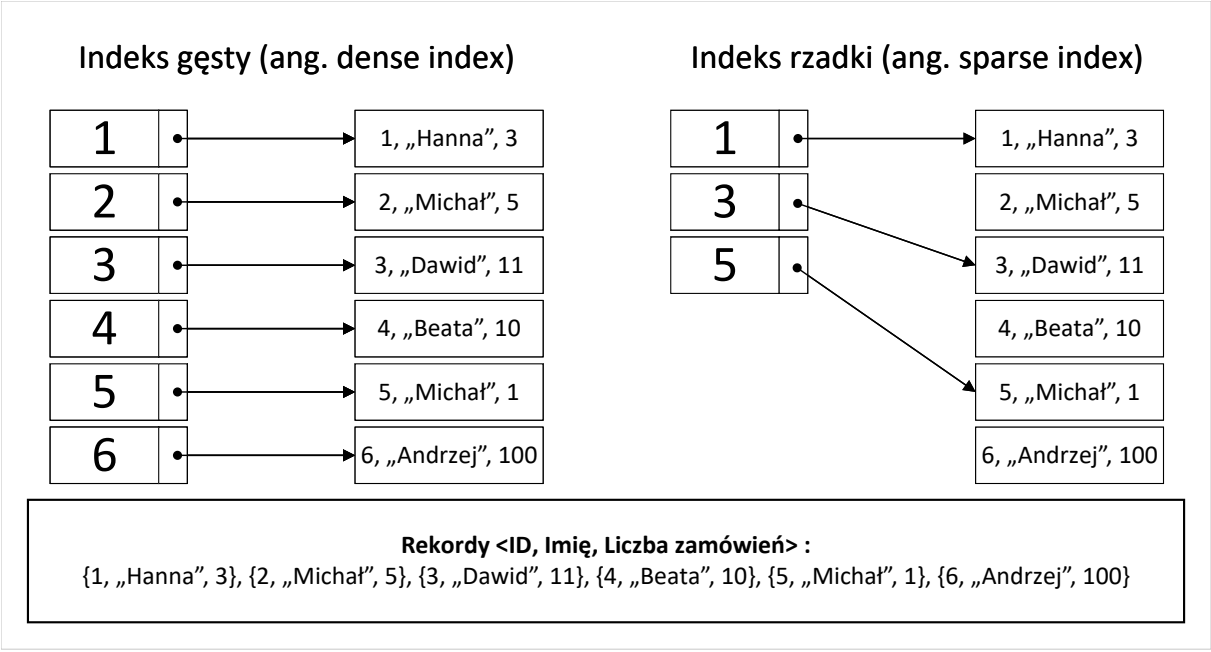
Indeksowanie

W tym rozdziale omówimy kilka sposobów indeksowania, przedstawimy wady i zalety każdego z podejść oraz spróbujemy przeanalizować ich działanie w kontekście nowoczesnych modeli pamięci.

Indeks to uporządkowana struktura danych, która umożliwia szybki dostęp do rekordu o wybranym kluczu. Indeksy są zakładane na pojedynczych atrybutach lub zbiorze atrybutów. Wybór leży po stronie administratora baz danych i zależny jest od przewidywanego wzorca kwerend. Tabela, która nie jest posortowana po atrybucie znajdującym się w kwerendzie, musi zostać wczytana w całości (ang. scan). Taka operacja ma liniową złożoność obliczeniową, a jej koszt, używając notacji asymptotycznej, zapisujemy jako $\mathrm{O}(n)$, gdzie n to liczba rekordów w tabeli. Aby przyspieszyć ten proces wykorzystuje się indeksowanie. Na zbiorze rekordów stosuje się porządek liniowy, a więc zapisuje się je w taki sposób, aby szybciej znaleźć rekord o wybranym kluczu. Gdy w kwerendzie użytkownik poda atrybut, po którym posortowana jest tabela, zamiast pełnego skanowania, wystarczy użyć struktury danych indeksu i w odpowiedni sposób znaleźć potrzebny rekord (ang. seek). Indeksy najczęściej realizowane są jako struktury drzewiaste, a złożoność obliczeniowa takiego wyszukiwania wynosi $\mathrm{O}(\log n)$. Wynika to z faktu, że aby znaleźć szukany rekord należy przejść po drzewie od korzenia do liści [14].

Indeksy dzielą się na kilka grup [15]. Z punktu widzenia charakterystyki atrybutu indeksowego wyróżnia się indeks podstawowy (ang. primary index) oraz indeks wtórny (ang. secondary index). Indeks podstawowy jest założony na atrybucie, który jest kluczem tabeli. Taki atrybut musi przyjmować wartości unikalne w obrębie wszystkich wartości danej tabeli. Tabela może posiadać tylko jeden indeks podstawowy, ponieważ istnieje tylko jeden klucz. Indeks wtórny założony jest na dowolnym atrybucie tabeli, który nie jest kluczem głównym. W przeciwieństwie do indeksu podstawowego, wartości w indeksie wtórnym mogą się powtarzać. Tabela może posiadać wiele indeksów wtórnych. Należy jednak pamiętać, że każdy indeks powiększa koszt manipulacji rekordów [16]. Wstawianie oraz usuwanie danych z tabeli wydłuża się wraz z każdym posiadanym indeksem, ponieważ zawartość każdego indeksu musi odzwierciedlać nowy stan tabeli.

Biorąc pod uwagę liczbę kluczy w indeksie względem wszystkich rekordów, indeksy dzielimy na gęste (ang. dense index) i rzadkie (ang. sparse index). Rysunek 2.1 przedstawia różnicę pomiędzy tymi indeksami. Indeks gęsty posiada w strukturze klucz do każdego rekordu w tabeli, indeks rzadki posiada tylko niektóre klucze z tabeli. Indeks gęsty posiada szybsze wyszukiwanie niż rzadki, ponieważ w samej strukturze danych mamy wszystkie klucze i jesteśmy w stanie dokładnie określić pozycję rekordu, którego szukamy. Ponieważ zazwyczaj indeks zapisuje klucz oraz wskaźnik do miejsca na dysku, przechodzenie po indeksie jest o wiele szybsze niż przechodzenie po fizycznych rekordach. Całkiem odwrotna sytuacja ma miejsce w przypadku indeksu rzadkiego. Aby zaoszczędzić pamięć potrzebną na zapisywanie dodatkowego indeksu, rekordy grupuje się w wirtualne bloki o ustalo-

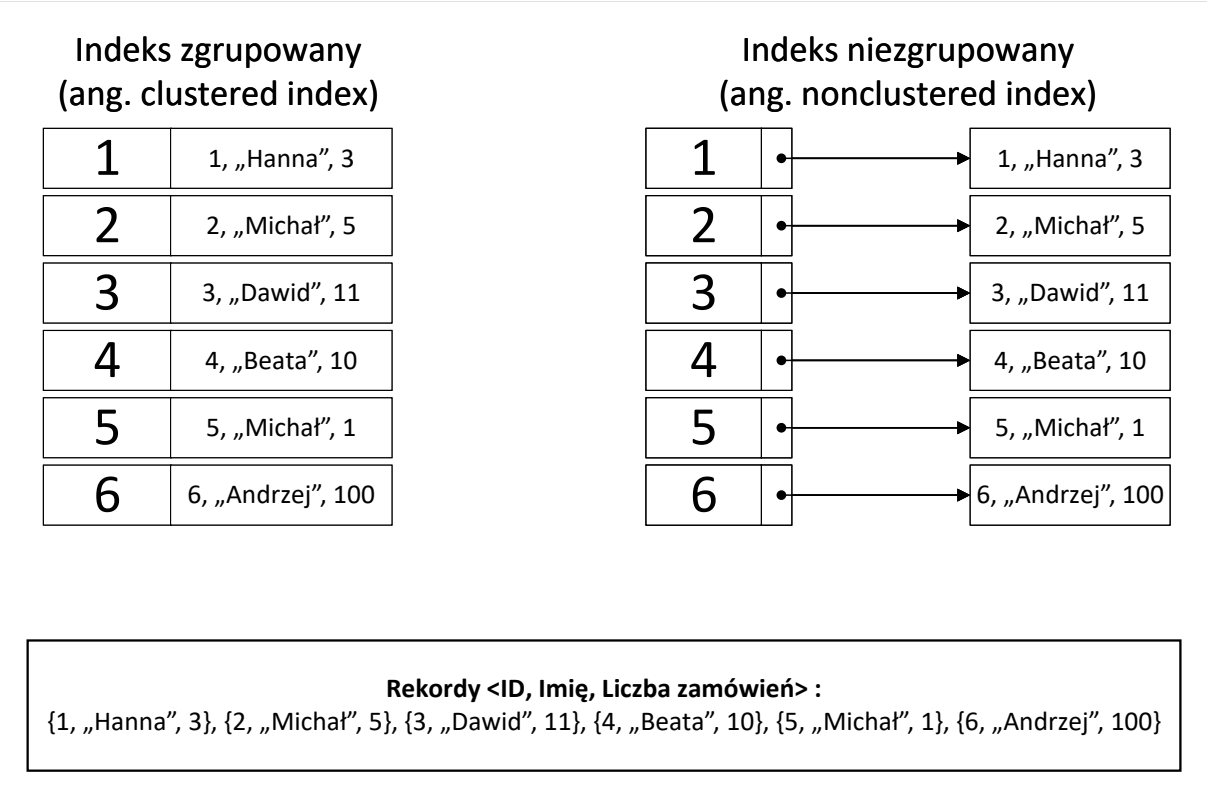


Rysunek 2.1: Indeksy gęste i rzadkie

nej wielkości (na rysunku wirtualny blok ma pojemność dwóch rekordów), a następnie wstawia się do indeksu pierwszy klucz z każdego bloku. Gdy chcemy wyszukać rekord za pomocą indeksu, najpierw znajdujemy blok, który musimy odczytać, a następnie wczytujemy cały wirtualny blok z dysku. Wyszukiwanie w indeksie rzadkim jest wolniejsze niż w gęstym. Indeks rzadki zużywa mniej pamięci oraz wymaga mniej czasu na aktualizację podczas wstawiania lub usuwania danych, ponieważ często wystarczy wykonać zmianę na wirtualnym bloku, pozostawiając strukturę indeksu bez zmian.

Ze względu na położenie rekordów w pamięci wyróżniamy indeksy zgrupowane (ang. clustered index) oraz niezgrupowane (ang. nonclustered index). Rysunek 2.2 przedstawia oba wymienione typy indeksów. Indeks zgrupowany zapisuje rekordy w strukturze indeksu, a więc może istnieć tylko jeden taki indeks dla tabeli. Jeśli chcemy stworzyć kolejny indeks zgrupowany, musimy skopiować tabelę i dopiero na kopii ułożyć rekordy fizycznie według innego porządku. Zaletą takiego indeksu jest bardzo szybki dostęp do wyszukiwanych danych, gdyż nie musimy wykonywać dwóch operacji na dysku, a więc odczytu strony, na której znajduje się klucz oraz odczytu fizycznego rekordu znajdującego się w miejscu wskazywanym przez wskaźnik zapisany obok klucza. Tak zapisany indeks jest o wiele większy niż normalny, ponieważ zawiera w sobie całą tabelę. Pamiętajmy jednak, że koszty operacji wykonywanych na indeksie skalują się wraz z liczbą rekordów w indeksie a nie wielkością indeksu. W związku z tym taki zapis nie wpływa negatywnie na pracę indeksu. Indeks niezgrupowany obok klucza zapisuje tylko wskaźnik do adresu gdzie fizycznie zapisany jest rekord. Takich indeksów można mieć kilka dla danej tabeli w zależności od potrzeb. Indeksy te charakteryzują się nieco wolniejszym czasem dostępu do rekordu, mniejszym kosztem aktualizacji struktury oraz mniejszym zużyciem pamięci od indeksów zgrupowanych.

W tej pracy najbardziej skupimy się na typach indeksowania podzielonych ze względu na sposób ułożenia danych w pamięci. Rozważane będzie ułożenie rekordów wierszowo i kolumnowo. W pracy poruszymy także nowy sposób indeksowania - indeksowanie częściowe



Rysunek 2.2: Indeksy zgrupowane i niezgrupowane

jako pewien pomysł automatyzacji wyboru danych do uporządkowania. Wymienione powyżej trzy typy indeksów, ich sposób działania oraz ich wady i zalety przedstawimy w dalszej części tego rozdziału.

2.1 Indeksowanie wierszowe

Indeksy zapisujące rekordy wierszowo to najstarszy rodzaj omawianych w tej pracy indeksów. Zaletą tego typu indeksów jest prostota implementacji (o wiele łatwiej stworzyć indeks wierszowy niż kolumnowy) oraz szybkość utrzymania porządku na takim indeksie. Cały rekord zapisany jest w ciągłym obszarze pamięci. Tak więc dodawanie nowego rekordu, usuwanie starego jak i modyfikacje wykonują pojedyncze operacje na dysku. Z drugiej strony taki rodzaj indeksowania posiada kilka wad. Po pierwsze, wierszowe ułożenie rekordów nie sprzyja kompresji. Im mniejsza dziedzina tym łatwiej zastosować kompresję danych. Zatem idealnym ułożeniem byłaby agregacja podobnych wartości z tych samych kolumn. Dodatkowo zazwyczaj tabela składa się z kilku lub nawet kilkunastu atrybutów. Większość kwerend nie będzie potrzebowała ich wszystkich. Mimo to, ze względu na ułożenie w pamięci, musimy wczytać cały rekord, niezależnie od zapytania.

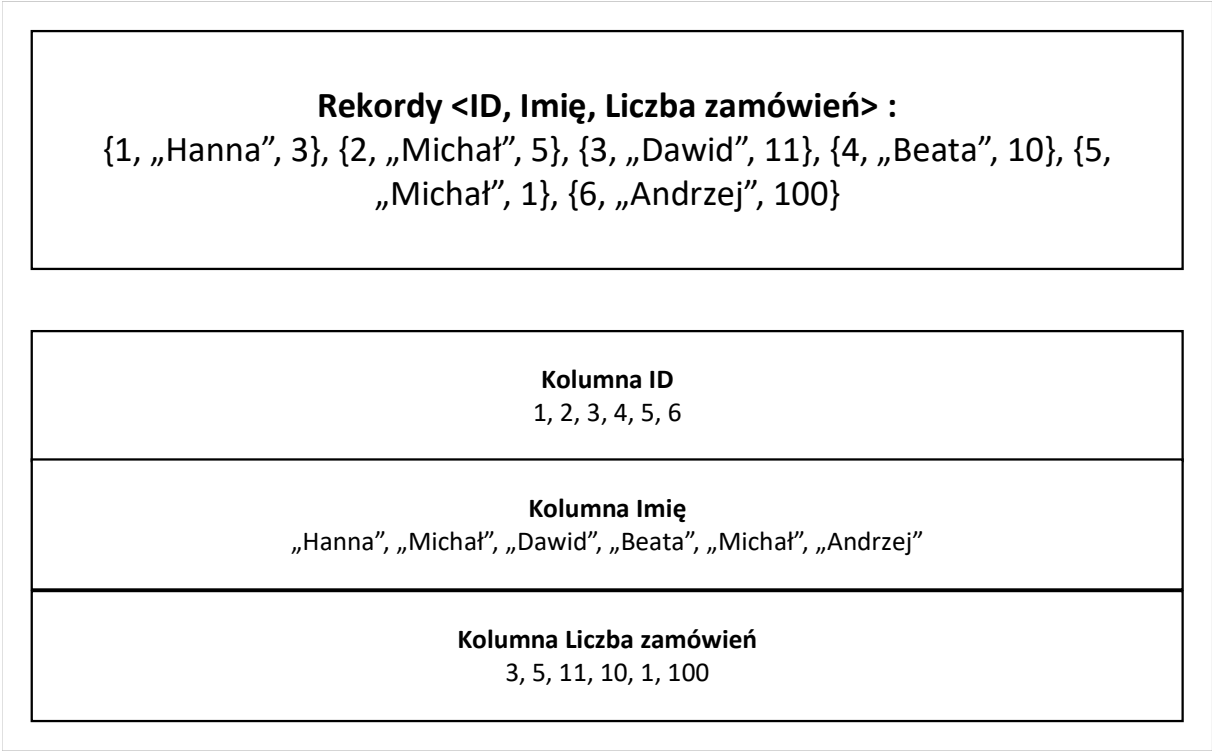
Za najprostszą formę indeksu wierszowego możemy uznać posortowaną tablicę. Koszt wyszukiwania w takiej tablicy jest równy $\mathrm{O}(\log n)$, ponieważ można użyć wyszukiwania binarnego. Największą wadą takiego podejścia jest koszt wstawiania i usuwania danych, który jest równy $\mathrm{O}(n)$. Taka forma indeksu jest bardzo nieefektywna, dlatego nie używa się jej w bazach danych. Pierwszym nietrywialnym indeksem był ISAM [17] (ang. Indexed Sequential Access Method). ISAM to dwupoziomowy indeks statyczny używany do prze-

chowywania rekordów na dyskach HDD. Poziom pierwszy indeksuje cylindry. Dane na tym poziomie zawierają poszukiwany klucz i adres do ścieżki dyskowej. Poziom drugi indeksuje ścieżki. ISAM jest indeksem statycznym, co oznacza, że nie posiada mechanizmów modyfikowania struktury. Usunięcie rekordu powoduje powstanie pustego miejsca na dysku, a nowe rekordy są dodawane do bloków przepełnienia (nowych miejsc w pamięci). W konsekwencji struktura jest bardzo nieefektywna pamięciowo oraz może być stosowana właściwie tylko do dysków HDD. W [18] opracowano wirtualną maszynę współpracującą ze strukturą ISAM (VSAM), dzięki czemu indeks ISAM można zaaplikować na dowolny sprzęt i dysk. VSAM niestety nie rozwiązał reszty problemów oryginalnego pomysłu. Dopiero wprowadzenie indeksów opartych na drzewach B+ [19] przyspieszyło wszystkie operacje wykonywane na indeksach. Drzewo B+ to samoorganizujące się drzewo posiadające ustaloną liczbę węzłów potomnych, często nazywanych potocznie dziećmi węzła. Taka struktura zapewnia logarytmiczny czas dostępu do dowolnego klucza oraz logarytmiczne dodawanie i usuwanie rekordów. Drzewo B+ jest najpopularniejszym indeksem baz danych zapisanych na dysku HDD. Jego strukturę można łatwo dopasować do parametrów dysku. Ustalając rozmiar węzła drzewa na rozmiar sektora dysku HDD osiągamy bardzo dobrą wydajność spowodowaną precyzyjnym buforowaniem obszarów dysku w pamięci RAM komputera. Kolejną strukturą danych, która jest bardzo często wykorzystywana to Filtr Bloom'a [20]. Filtr Bloom'a jest to probabilistyczna struktura danych, która jest bardzo efektywna pamięciowo i w czasie stałym potrafi określić przynależność rekordu do zbioru. Wadą jest prawdopodobieństwo pojawienia się fałszywie błędnej odpowiedzi pozytywnej. Oznacza to, że jeśli filtr określi obecność elementu w zbiorze, informacja ta może być nieprawdziwa. Jednak gdy filtr stwierdzi, że elementu w zbiorze nie ma, to mamy pewność, że tak jest. Struktura ta wykorzystywana jest do redukcji zapytań do indeksu, które są kosztowne, ponieważ wczytują dane prosto z dysku.

Ze względu na odmienną charakterystykę każdej z pamięci, obecnie używa się wielu struktur danych indeksujących rekordy wierszowo. Gdy tabela zapisana jest w pamięci operacyjnej RAM, używa się drzewa czerwono czarnego [21], drzewa AVL [22] lub drzewa TRIE [23] [24]. Do popularnych indeksów działających efektywnie na pamięci flash i dyskach SSD należą między innymi FD [25] i LA [26]. Każde z nich w inny sposób redukuje liczbę zapisów oraz wymazań na dyskach SSD, co czyni je o wiele lepszym wyborem od drzewa B+ w kontekście pracy na pamięci flash. Z kolei do pracy na pamięci PCM często wybiera się jedną z modyfikacji drzewa B+ ([27], [28], [29]), które kosztem dodatkowych odczytów, redukują liczbę wolnych zapisów.

2.2 Indeksowanie kolumnowe

W ostatnich latach kolumnowe przechowywanie danych w bazie stało się coraz bardziej popularne. Wynika to z faktu, że większość zapytań do baz danych nie potrzebuje danych ze wszystkich kolumn. Zatem w wierszowej implementacji marnujemy odczyty z dysku na wczytanie całego wiersza i wydobycie tylko potrzebnych atrybutów. W podejściu kolumnowym na jednej stronie w pamięci przetrzymywane są wartości pojedynczej kolumny dla kilku rekordów, dzięki czemu podczas wyszukiwania wczytujemy tylko potrzebne informacje. Niestety, kolumnowe podejście jest wolniejsze podczas zapisywania nowych rekordów, ponieważ musimy zapisać podzielony rekord na kolumny w kilku miejscach na dysku [10], [11], [12].



Rysunek 2.3: Kolumnowe ułożenie rekordów

Rysunek 2.3 przedstawia przykład kolumnowego ułożenia danych. Każda kolumna zapisana jest w osobnym miejscu w pamięci. Gdy użytkownik poprosi w kwerendzie o zestaw liczby zamówień dla każdego klienta, chcąc zobaczyć tylko ich imiona, wtedy z dysku odczytamy tylko potrzebne strony. Sektor dysku, na którym zapisane są ID użytkowników nie zostanie wczytany podczas tego procesu, co przyspieszy obsługę takiej kwerendy. Przechowywanie kolumnowe niesie ze sobą również trudność w scalaniu wszystkich atrybutów w rekord. Dlatego struktura danych indeksu musi prócz standardowych operacji zawierać także operacje scalania, która dla danego atrybutu (niekoniecznie klucza) zwróci cały rekord. Jest to najtrudniejsza operacja do zaprojektowania ze względu na nietrywialne struktury danych użyte do indeksowania np. drzewo B+. Sytuacja na rysunku jest o wiele prostsza. Atrybut danego rekordu jest na tym samym indeksie tablicy. Zatem znając pozycję jednego atrybutu możemy określić położenie innych atrybutów należących do danego rekordu i scalić je ze sobą. Taki algorytm dostał opracowany w jednej z pierwszych struktur zaprojektowanych do kolumnowego ułożenia tabeli DSM [30] (ang. Decomposition Storage Model). Struktura DSM ma jednak wiele wad. Przede wszystkim jest nieposortowana. Tak więc, gdy chcemy dodać kolejny rekord do DSM, musimy pobrać z niego wszystkie wartości atrybutów, a następnie każdą z nich dodać na koniec listy związanej z odpowiednią kolumną. Dzięki temu zawsze mamy pewność, że atrybuty tego samego rekordu są umieszczane na tej samej pozycji listy. Niestety z tego powodu każda kwerenda musi wykonać pełen skan potrzebnych kolumn.

Za pierwszy silnik w pełni wspierający kolumnowy zapis tabeli uznaje się C-Store [31]. Silnik ten zapisywał tabelę w dwóch osobnych sekcjach: sekcji zorientowanej na szybkie odczyty, która była zapisana kolumnowo, oraz sekcji zoptymalizowanej pod częste zapisy i aktualizacje danych. Druga sekcja zapisana była wierszowo aby umożliwić szybkie wsta-

wianie i usuwanie rekordów. C-Store sam decydował kiedy przenieść dane z jednej sekcji do drugiej. Od tamtego czasu zaproponowano wiele nowych technik usprawniających algorytmy stworzone w C-Store [32], [33], [34], [35]. Dodano także wiele nowych usprawnień, między innymi takie, które skupiają się równoległym odczycie kilku kolumn [36], [37], [38] oraz na kompresji danych w obrębie każdej z nich [39], [40], [41], [42], [43], [44]. Obecnie ze względu na wiele zastosowań i możliwych optymalizacji ułożenie kolumnowe tabeli jest bardzo popularne. Większość nowoczesnych silników baz danych wspiera ten sposób zapisu tabeli. Należą do nich między innymi: MonetDB [45], PostGreSQL [46], VectorWise [47], Druid [48] i eXtremeDB [49]. Najpopularniejszą strukturą danych używaną obecnie do indeksowania rekordów zapisanych kolumnowo jest PDT [50] (ang. Positional Delta Tree). Indeks ten wykorzystywany jest w bazach danych zapisanych w pamięci operacyjnej komputera (ang. in-memory database) oraz na dyskach twardych HDD. PDT zapisuje różnice pomiędzy starymi wartościami a nowymi, dodatkowo zapisując pozycję na której należy zaaplikować tę zmianę. Dzięki temu, że w strukturze zapisujemy nie tylko dane, ale także pozycje wszystkich danych dla każdego z atrybutów, możemy w łatwy sposób scalić je w cały rekord. Dodatkowo technika zapisywania samych zmian służy do agregowania wielu z nich w jedną dużą zmianę (ang. batch).

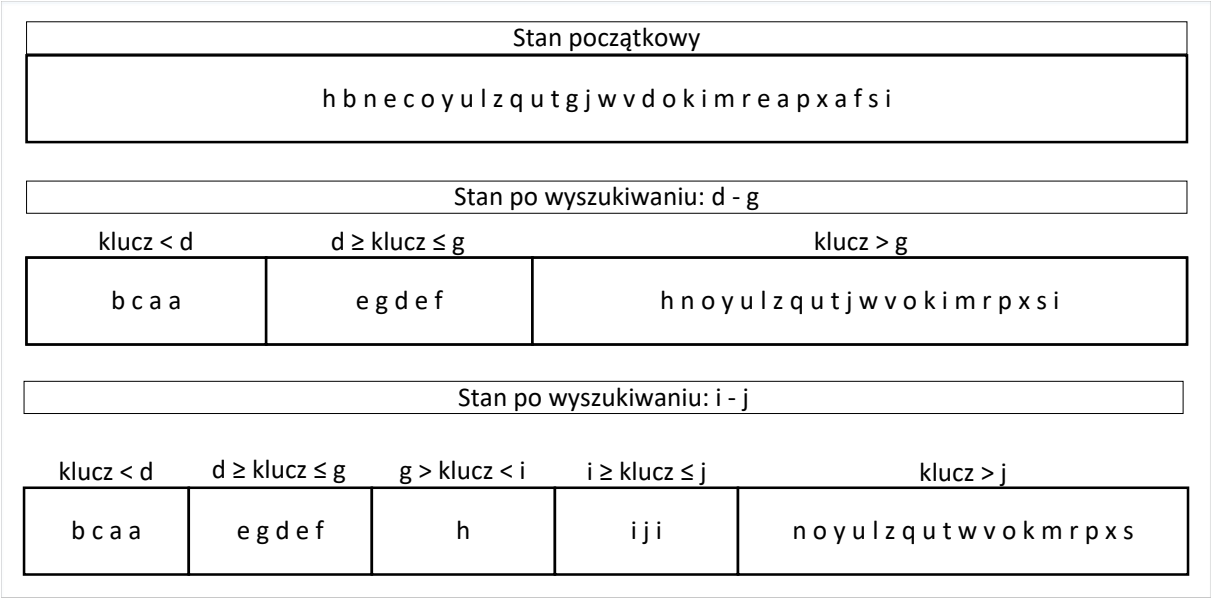
Kolumnowy zapis baz danych używa się najczęściej w sytuacjach, w których dominują odczyty. Zapis ten przyspiesza odczyt danych poprzez lepszą kompresję oraz skanowanie tylko potrzebnych kolumn, a co za tym idzie zmniejsza się liczba bajtów jaką trzeba odczytać z dysku. Z drugiej strony zapis nowego rekordu jest wolniejszy, ponieważ jeden rekord trzeba zapisać w kilku miejscach. Gdy baza danych zapisana jest w pamięci operacyjnej komputera lub na dysku HDD, zysk z odczytu przewyższa niewielką stratę na zapisie. Jednak nowoczesne pamięci takie jak flash, SSD czy PCM posiadają odmienną charakterystykę. Zapis na tych pamięciach jest kilkukrotnie wolniejszy niż odczyt, a więc klasyczne struktury danych nie osiągają dobrych rezultatów w ułożeniu kolumnowym. Algorytm musi nie tylko spełniać wszystkie wymogi kolumnowego zapisu takie jak posiadanie szybkiej operacji scalania, ale musi także wykorzystywać techniki przyspieszające zapis danych adekwatne do używanych pamięci. Dobrymi przykładami jest system PAX [51], który polega na wirtualnym podziale strony dysku SSD na mini strony. Każda mini strona zawiera tylko dane jednej kolumny. Mamy zatem ułożenie kolumnowe kilku rekordów w obrębie jednej fizycznej strony. Taki mechanizm nie zmniejsza liczby bajtów wczytywanych z dysku, gdyż najmniejszą jednostką na której pracujemy jest strona, ale przyspiesza pracę procesora ze względu na lepsze wykorzystanie pamięci procesora (ang. cache memory). Kolejnym przykładem jest struktura FBDSM [52] (ang. Flash-based Decomposition Storage Model), która dostosowała oryginalny pomysł DSM [30] do pamięci flash. Zamiast jednej tabeli, są dwie. Pierwsza to tabela główna PT, w której zapisywane są dane dodane do tabeli, które nie zostały usunięte ani zmienione. Druga tabela to tabela LT, która zawiera wszystkie zmiany na tabeli PT a więc: usunięcia rekordów oraz zmianę wartości. Dzięki temu nadpisywanie danych jest odroczone w czasie oraz realizowane później za pomocą paczek danych, co jest o wiele szybsze w przypadku pamięci flash. Struktura ta jednak nie rozwiązuje głównego problemu, czyli braku relacji porządku. Ponadto tabela nie jest posortowana po kluczu, a po czasie, aby zachować pozycje atrybutów dla danego rekordu, co umożliwia szybkie scalanie kolumn w jeden pełny rekord.

2.3 Indeksowanie częściowe

Stworzenie idealnego zbioru indeksów zawierającego wszystkie potrzebne indeksy o wszystkich potrzebnych atrybutach do obsługi każdej z kwerend to bardzo trudne zadanie, przed którym stają administratorzy baz danych. Nie da się przewidzieć wszystkich możliwych kwerend jakie użytkownik końcowy będzie chciał wykonać na tabelach. Z tego więc powodu dąży się do automatyzacji procesu tworzenia indeksów [53], [54], [55]. Stosuje się podejścia, które próbują dostosowywać porządek bazy danych i kolejne indeksy wraz ze zmianą kwerend. Kolumny po których musimy sortować dane podczas zapytania są monitorowane i analizowane. Dzięki temu silnik bazy stwierdza, czy potrzebny jest nowy indeks lub czy istniejący przestał być już używany. Wtedy następuje reorganizacja, tzn. dodawany jest nowy indeks poprzez kopiowanie tabeli i sortowanie jej po innym kluczu lub usuwany jest stary indeks. Takie podejście ma kilka wad. Po pierwsze, czas potrzebny na analizę może być zbyt długi. Gdy silnik zacznie tworzyć indeks, wzorzec kwerendy może ulec zmianie, przez co nie zostanie on użyty, a czas na jego tworzenie zostanie zmarnowany. Po drugie, sam czas tworzenia indeksu jest ogromny, ponieważ trzeba posortować całą tabelę często niemieszczącą się w pamięci RAM.

W przeciągu kilku ostatnich lat pojawiło się kolejne podejście do automatyzacji procesu indeksowania. Tym razem nie skupiano się na doborze atrybutów jakie powinny być uporządkowane, ale wybraniu podzbioru tabeli, która jest na tyle często używana, że opłaca się uporządkować ten podzbiór i stworzyć na nim częściowy indeks. W [56] [57] zaproponowano Indeks Cracking jako nową metodę automatycznego, iteracyjnego tworzenia indeksu. W wyniku działania kwerendy, część tabeli jest reorganizowana, aby dostosować się do wzorca obecnego zapytania. Proces ten próbuje zamortyzować koszt tworzenia indeksu, dzieląc go na wiele części. Z początku czas będzie większy niż zwykły skan, ponieważ musimy dodać kilka operacji potrzebnych na powiększenie posortowanego zbioru. Jednak z czasem będzie się on zmniejszał, ponieważ będziemy mogli korzystać z częściowego indeksu. Gdy cały proces się zakończy, otrzymamy w pełni posortowaną tabelę. Indeks Cracking używa metody podziału danych na partycje (ang. partition) względem elementów z zakresu zapytania i działa podobnie do sortowania szybkiego (ang. quicksort) [58]. Za każdym razem partycja, która zawiera klucze z zakresu zapytania dzielona jest na kolejne tak, aby jedna z nich zawierała tylko elementy mniejsze od zakresu kwerendy, druga wartości z danego zakresu, a trzecia dane o kluczach większych niż zakres. Zatem w większości przypadków stworzone zostaną dwie nowe partycje.

Rysunek 2.4 przedstawia trzy etapy tego procesu. Każda litera obrazuje wartość nowego klucza rekordu, który może się powtarzać, ponieważ indeksy częściowe są zazwyczaj indeksami wtórnymi. Górny prostokąt pokazuje układ danych przy inicjalizacji procesu zaraz po skopiowaniu. Środkowy prostokąt pokazuje partycje powstałe w wyniku zapytania od d do g. Elementami podziału są zatem wartości d i g. Pierwsza partycja zawiera wartości mniejsze od d, środkowa partycja zawiera elementy, które należało wczytać, czyli pomiędzy d i g, a ostatnia partycja zawiera pozostałe rekordy, czyli te z kluczami większymi niż g. Warto zwrócić uwagę, że partycje nie są posortowane wewnętrznie, a kolejność elementów dodawana do partycji jest taka sama jak ich wczytania. Ponieważ stan początkowy zawierał (idąc od lewej strony) rekordy $\{b,\ c,\ a,\ a\}$, to właśnie w takiej kolejności zostały umieszczone w partycji pierwszej. Ostatni stan przedstawia sytuację stworzoną przez kwerendę po zakresie i - j. Pierwsze dwie partycje są rozłączne z nowym zakresem wyszukiwania, a więc nie zostały zmienione podczas wyszukiwania. Ostatnia partycja zo-



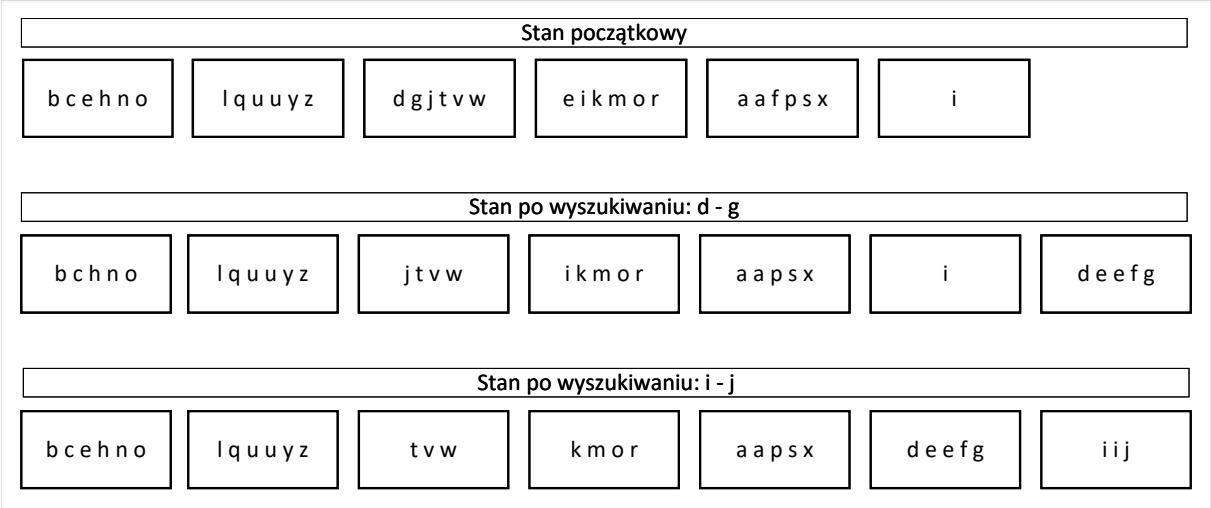
Rysunek 2.4: Indeks Cracking

stała podzielona na tych samych zasadach co poprzednio, czyli na zbiory o elementach mniejszych od i, elementach pomiędzy i a j oraz na elementy większe niż j.

Chociaż Indeks Cracking jest bardzo dobrym podejściem, gdy z góry nie znamy wszystkich potrzebnych kluczy, to jednak metoda ta ma kilka wad. Przede wszystkim częste reorganizacje na małych częściach tabeli są stosunkowo wolne w przypadku gdy używamy pamięć blokową taką jak dyski HDD lub SSD. Po drugie szybkość tworzenia indeksu zależna jest od wzorca kwerend. Tak jak w sortowaniu szybkim, złożoność obliczeniowa zależna jest od wyboru elementu rozdzielającego partycję (ang. pivot). I chociaż istnieją algorytmy wyboru, to jednak nie możemy ich tutaj zastosować, ponieważ punkt podziału jest ustalony przez użytkownika (krańce zakresu kwerendy). Trzecią wadą jest dość długi czas potrzebny na osiągnięcie takich samych efektów jak pełny indeks. Niestety w najgorszym przypadku, gdy nigdy nie będziemy mogli skorzystać z partycji (gdy element rozdzielający partycję był zawsze najmniejszym lub największym elementem tabeli) metoda ta będzie gorsza niż zwykły skan. Dodatkowo ciężko jest dodać nowe dane do indeksu, który już jest częściowo posortowany, tak aby nie zaburzyć partycji. Wiele optymalizacji zostało zaproponowanych, aby zwiększyć przydatność algorytmu Indeks Cracking. W [59] dostosowano metodę do czysto-kolumnowego zapisu tabeli. W [60] wprowadzono lepszy algorytm wyboru elementu rozdzielającego, który nie zależy tylko i wyłącznie od kluczy wyszukiwania, dzięki czemu zmniejszona została liczba niepotrzebnych reorganizacji. W [61] dodano algorytmy wstawiania, aktualizacji oraz usuwania danych podczas reorganizacji partycji.

Kolejna metoda automatycznego tworzenia indeksu podczas kwerend została zaproponowana w [13]. Adaptive Merging w przeciwieństwie do Indeks Cracking korzysta z algorytmu scalania posortowanych list tak jak sortowanie przez scalanie (ang. mergesort) [62]. Gdy po raz pierwszy musimy wyszukać elementy biorąc pod uwagę wartości z danej kolumny, kopiujemy tabelę, sortujemy wczytując tyle danych na raz, ile mieści się w buforze. Następnie sortujemy wartości w obrębie danej partycji. W kolejnych zapytaniach będziemy usuwać dane z posortowanych partycji oraz ze zbioru wczytanych danych tworzyć nowe partycje. W ten sposób uzyskamy po pewnym czasie zbiór partycji, które nie

tylko są posortowane wewnętrznie, ale także tworzą posortowany ciąg. W takim momencie możemy scalić je w jedną partycję otrzymując pełny indeks. Partycje mogą być przetrzymywane w strukturze Partition B+-Tree [63]. W takim przypadku operacje na partycjach wykonywane są podobnie do zwykłego B drzewa [19]. Właśnie dlatego podczas całego procesu mamy wymierne korzyści z posiadania partycji. Podczas zapytania możemy równolegle odczytywać miejsca zapisu partycji, a w niej za pomocą wyszukiwania binarnego (ang. binary search) znaleźć potrzebne dane.



Rysunek 2.5: Adaptive Merging

Rysunek 2.5 przedstawia trzy kroki tego procesu. Aby zobrazować różnice pomiędzy tym podejściem a Indeks Cracking, dane oraz kwerendy pozostały takie same jak w poprzednim przykładzie. Zatem górny prostokąt pokazuje układ danych przy inicjalizacji procesu zaraz po skopiowaniu i sortowaniu paczek danych. Każda partycja posiada swoją maksymalną pojemność (ang. capacity) wynikającą wprost z rozmiaru bufora, w którym można posortować dane. W naszym przykładzie pojemność partycji została ustawiona na 6 rekordów. Zatem podczas inicjalizacji dane są kopiowane, dzielone na 6-elementowe zbiory, sortowane a następnie zapisywane do nowych partycji. Drugi stan obrazuje sytuację powstałą w wyniku zapytania d - g. Każda partycja, która może zawierać dane potrzebne do wyniku kwerendy jest skanowana, następnie rekordy z kluczami o wartościach z przedziału d - g są usuwane z poszczególnych partycji, sortowane w buforze a następnie zapisane do całkiem nowej partycji. Ostatni stan powstał w wyniku kwerendy f - j. System powtórzył wszystkie kroki wymienione wcześniej, usunął potrzebne dane, posortował je i zapisał do nowej partycji. Warto zwrócić uwagę, że do bufora trafił ostatni rekord z partycji numer 6, która zawierała tylko jeden element - i. Ponieważ w wyniku tej kwerendy partycja stała się pusta, została usunięta z systemu. Metoda ta dostosowana jest do pamięci blokowej: nie dzielimy partycji na wiele nowych, każde zapytanie tworzy jedną dodatkową partycję, która zapisywana jest sekwencyjnie. Łatwo zauważyć, że im większy bufor, tym mniej będzie partycji oraz szybciej stworzymy pełny indeks. Dlatego podczas pierwszego kroku można posortować większe zbiory używając sortowania dostosowanego do pamięci zewnętrznej [64]. W [65] przedstawiono zrównoleglone wersje algorytmu sortowania, scalania i odczytu danych z partycji. W [66], [67] połączono metodę z pewnymi procedurami wykonywanymi w Indeks Cracking. Niestety wymienione optymalizacje, skupiają się zapisie całej tabeli w pamięci RAM, przez co są nieoptymalne dla nowoczesnych typów pamięci takich jak flash, SSD czy PCM.

Charakterystyka pamięci trwałych

W tym rozdziale zajmiemy się analizą charakterystyk poszczególnych pamięci wykorzystywanych w dalszej części pracy. Bardzo istotną kwestią jest to, aby przy wyborze struktury do indeksowania bazy danych brać pod uwagę cechy pamięci, na której ta baza jest zapisana. Często występuje sytuacja, gdy struktura indeksu, który dobrze działa na klasycznych dyskach twardych HDD, nie będzie dobrym wyborem dla nowszych modeli dysków SSD. W takim przypadku wydajność indeksu wynika z braku wykorzystania pełnego potencjału używanej pamięci. Dobrym przykładem jest popularne drzewo B+ [68]. Jest to podstawowa struktura danych używana do przechowywania danych w sposób uporządkowany na dyskach HDD. Jednak ze względu na odmienne właściwości pamięci flash, drzewo B+ nie jest dobrym wyborem, gdy baza danych zapisana jest na dyskach SSD [69]. Różne pamięci posiadają różny czas dostępu do danych (ang. latency) oraz szybkość przetwarza danych (ang. bandwidth) (zobacz tabelkę 3.1). Rysunek 3.1 przedstawia hierarchię pamięci - im pamięć jest wyżej w hierarchii, tym jest szybsza, ale jednocześnie droższa i o mniejszej pojemności.

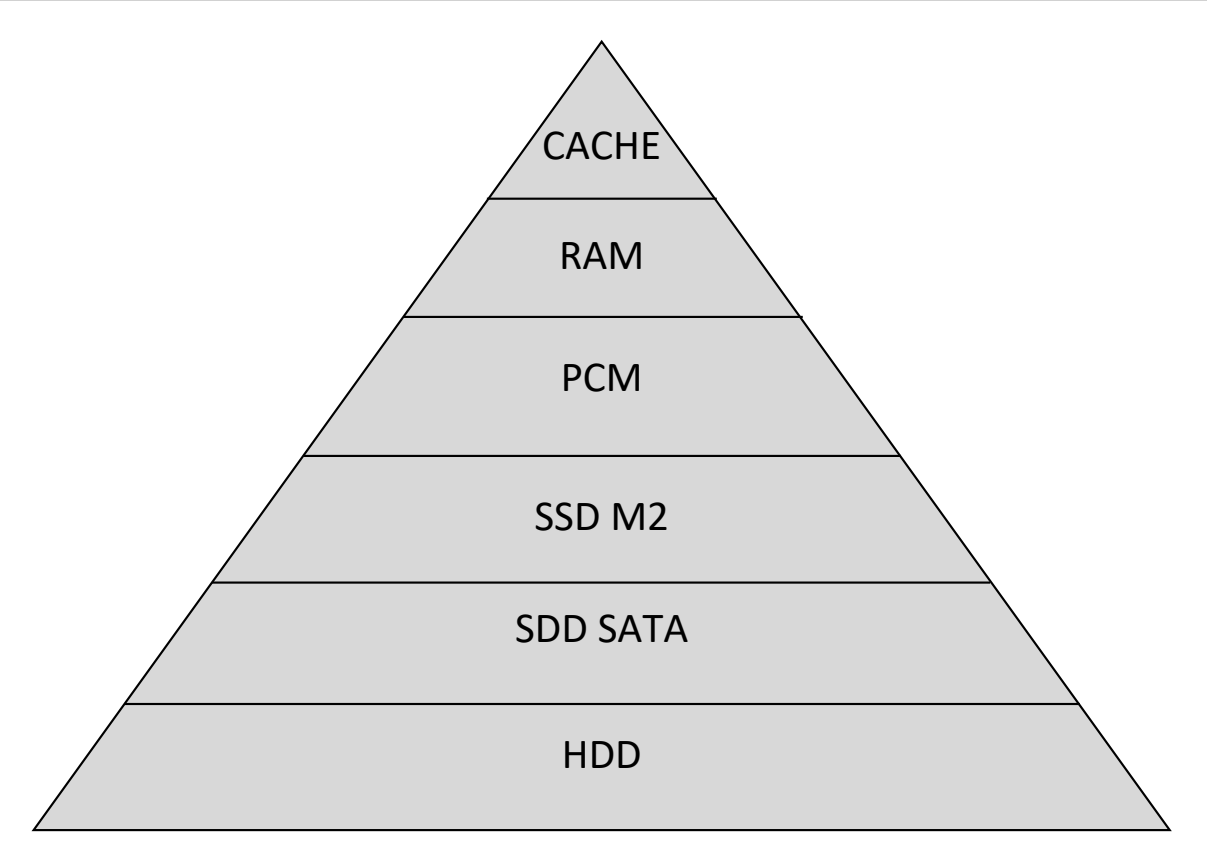
Pamięć podręczna procesora (ang. cache) zbudowana jest często z SRAM (ang. Static Random Access Memory), która charakteryzuje się bardzo szybkim czasem dostępu. Procesor, zanim zacznie pracować na danych, musi wczytać potrzebne fragmenty pamięci (czy to RAM czy to PCM) do swojej pamięci podręcznej.

Tak samo jak pamięć główna komputera, pamięć podręczna jest typem pamięci ulotnej. Dane zapisane w tych pamięciach ulegają degradacji przy wyłączeniu zasilania. Obie te pamięci są bajtowo adresowalne. Możemy odczytywać i zapisać pojedyncze bajty. W celu optymalizacji system dzieli pamięć podręczną na bloki (ang. cache line) zwykle wielkości 64B, a pamięć RAM na strony o wielkości 4KB. Pamięci te cechują się bardzo szybkim czasem dostępu i dużą przepustowością danych. Pamięć zmiennofazowa PCM (ang. Phase Change Memory) to nowy rodzaj pamięci trwałej. Dane umieszczone na niej nie ulegają zniszczeniu po odłączeniu zasilania, a więc nadaje się do przechowywania dużych ilości danych. PCM tak samo jak pamięć RAM jest bajtowo adresowalna. Cechuje się szybkim czasem dostępu i dużą asymetrią pomiędzy szybkością odczytu i zapisu. Pozostałe

Pamięć	Czas dostępu	Szybkość odczytu	Szybkość zapisu
L1 CACHE	1,2ns	105GB/s	58GB/s
RAM	20ns	20GB/s	12GB/s
PCM	350ns	8,1GB/s	5,6GB/s
SSD M2 Gen 3.0	0,005ms	3,5GB/s	3,2GB/s
SSD SATA	0,36ms	550MB/s	500MB/s
HDD	4,17ms	160MB/s	150MB/s

Tabela 3.1: Porównanie rodzajów pamięci

rodzaje pamięci przedstawione na rysunku (dyski SSD) to trwałe pamięci blokowe. Praca zarówno z dyskami SSD, jak i dyskami HDD jest trudniejsza niż w przypadku pamięci bajtowo adresowalnych, ponieważ najmniejszą jednostką dostępną dla użytkownika jest strona, w przypadku dysku SDD, oraz sektor w przypadku dysków HDD. Wielkość stron to zazwyczaj 2KB-8KB, a wielkość sektora to 512B-4096B. Takie ograniczenie zmusza użytkownika do własnego buforowania danych i agregacji kilku operacji w jedną.



Rysunek 3.1: Hierarchia pamięci (względem szybkości)

3.1 Zarządzanie dyskami z poziomu systemu operacyjnego Linux

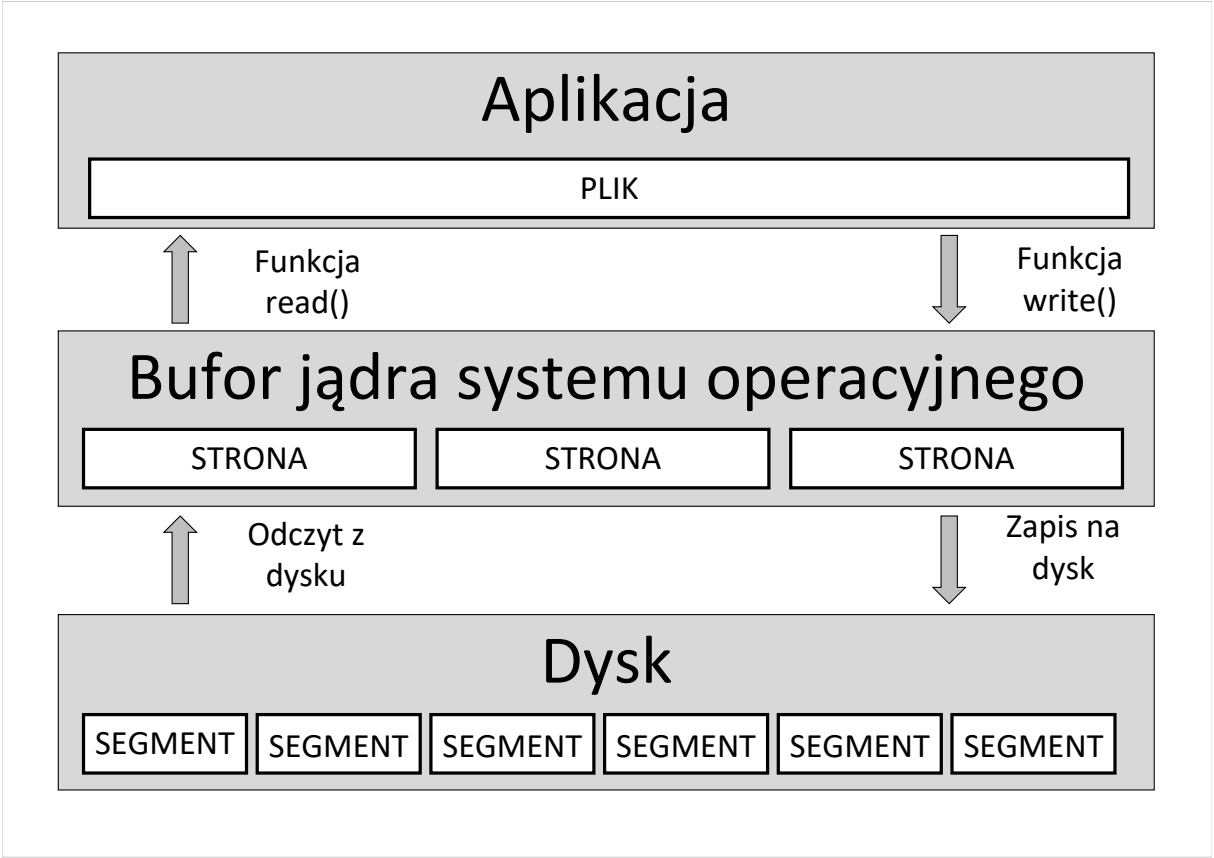
Aby w pełni zrozumieć wpływ charakterystyki pamięci na algorytmy indeksowania, musimy najpierw przeanalizować jak system operacyjny zarządza dyskami oraz wykonuje na nich operacje. Wszystkie eksperymenty przedstawione w dalszej części tej pracy zostały przeprowadzone na systemie operacyjnym Linux. Dlatego też omówimy sobie działanie tego systemu. Linux jest systemem otwartym (ang. opensource), oznacza to, że każdy z nas może obejrzeć i przeanalizować kod źródłowy jądra tego systemu [70].

System operacyjny Linux dostarcza nam kilkanaście dostępnych operacji na plikach [71], które bezpośrednio przekładają się na pracę z dyskami. Możemy je podzielić na następujące kategorie:

- **systemowe operacje WE/WY (ang. syscalls)**: open, write, read, flush, close

- **standardowe operacje WE/WY (ang. standard IO)**: fopen, fwrite, fread, fflush, fclose
- **wektorowe operacje WE/WY (ang. vectored IO)**: writev, readv,
- **operacje mapowania do pamięci (ang. memory-mapped IO)**: mmap, msync, munmap

Warto wspomnieć, że system operacyjny nie może wykonać operacji na pliku w inny sposób niż poprzez systemowe operacje, które wykonają odpowiedni kod jądra systemu operacyjnego. Wszystkie pozostałe operacje takie jak: standardowe operacje WE/WY, wektorowe operacje czy operacje mapowania wykonają pośrednio operacje systemowe z wykorzystaniem dodatkowego buforowania danych na poziomie aplikacji (biblioteki).



Rysunek 3.2: Operacje systemowe WE/WY

Jądro systemu operacyjnego, które przyjmuje operacje systemowe, nie wykonuje operacji *read()* i *write()* bezpośrednio na pliku (o ile nie dodamy takiej opcji). Zamiast tego dokonuje wewnętrznego buforowania danych (patrz rysunek 3.2). Bufor ten nazywamy Page Cache. Gdy chcemy wykonać operację na pliku lub jego fragmencie, jądro systemu sprawdza, czy sektor nie znajduje się w buforze. Gdy wykonujemy funkcję *read()*, najpierw sprawdzany jest bufor danych Page Cache. Jeśli dane znajdują się w buforze, potrzebne strony są od razu kopiowane do adresu podanego przez użytkownika aplikacji. Jednak gdy potrzebny sektor nie został znaleziony, system zgłasza Page Fault. W takim przypadku dane są wczytywane bezpośrednio z dysku, trafiają do bufora Page Cache i zostają skopiowane do aplikacji. Analogicznie postępujemy w przypadku funkcji *write()*. Dane od

użytkownika trafiają do Page Cache. Niezależnie czy segment znajdował się buforze czy też nie, operacja *write()* wpisuje zmienione dane fragmentu pliku jako strony Page Cache. Taka strona jest oznaczona jako *dirty* i oczekuje na wpisanie do dysku. Operację tę możemy wykonać manualnie za pomocą funkcji *flush()* lub *close()*. Możemy także czekać, aż system operacyjny sam podejmie decyzję o wyrzuceniu strony z bufora Page Cache i wpisaniu zmian w sektorach fizycznie na dysk.

Operacja *flush()* zostanie wywołana, gdy wolne miejsce w buforze Page Cache się skończy. Wtedy system operacyjny musi podjąć decyzję, które strony należy wyrzucić z bufora, a które zostawić. Istnieje kilka podejść do tego problemu. Najbardziej popularne to FIFO (ang. First In First Out), MRU (ang. Most Recently Used) i LRU (ang. Last Recently Used) [72]. Dodatkowo, gdy użytkownik aplikacji wie w jaki sposób będzie pracować z plikiem, może ustawić odpowiednią strategię za pomocą funkcji fadvise(). Najczęściej używa się trzech opcji: *FADV_SEQUENTIAL*, *FADV_RANDOM* oraz *FADV_WILLNEED*. Pierwsza opcja sugeruje, że fragmenty plików będą odczytywane sekwencyjnie, jeden po drugim. W takiej sytuacji system może przygotować się na kolejne operacje i odczytywać segmenty dysku wcześniej (ang. prefetch). Druga opcja jest całkowitym przeciwieństwem pierwszej. Ponieważ wiemy, że zawsze będziemy odczytywać pojedyncze sektory dysku umieszczone w różnych miejscach, nie powinniśmy wykonywać operacji *prefetch()* i odczytywać więcej sektorów jednocześnie. Trzecia opcja mówi systemowi, że fragment pliku będzie potrzebny w najbliższej przyszłości i nie należy usuwać strony z Page Cache. Pamiętajmy jednak, że jeśli niepoprawnie ustawimy strategię, czyli niezgodnie z faktycznym użyciem pliku, to możemy spowolnić działanie systemu. Dlatego należy korzystać z funkcji *fadvise()* tylko wtedy, gdy wiemy w jaki sposób będziemy pracować na pliku. Dobrym przykładem wykorzystania funkcji *fadvise()* jest baza danych RocksDB. Dzięki komunikacji z jądrem systemu Linux, operacje na tej bazie zostały przyspieszone o kilkanaście procent [73].

Kolejną bardzo ważną opcją jest *O_DIRECT*. Możemy użyć tego parametru przy otwieraniu pliku za pomocą funkcji *open()*. Opcja ta wyłącza używanie bufora Page Cache. Oznacza to, że wszystkie operacje będą wykonywane bezpośrednio na dysku. Funkcja ta przydatna jest wtedy, gdy chcemy sami buforować dane z poziomu aplikacji. Metodę operacji na plikach z pominięciem struktury Page Cache wykorzystują MySQL [74] w silniku InnoDB [75] oraz PostgreSQL [46].

3.2 Dysk twardy (HDD)

Dysk twardy HDD (ang. Hard Disk Drive) to jeden z najbardziej popularnych rodzajów pamięci masowej wykorzystywanych na przełomie XX i XXI wieku. Podstawowym elementem twardego dysku jest znajdujący się w obudowie wirujący talerz lub zespół talerzy w przypadku nowoczesnych modeli. Konstrukcja dysku wykonana jest zazwyczaj z aluminium, a jego wnętrze pokrywa miękka warstwa zabezpieczająca, która zapobiega uszkodzeniom urządzenia oraz odpowiada za częściowe tłumienie hałasu. Talerz zamocowany jest na osi napędzanej silnikiem elektrycznym. Zapis i odczyt danych zapewniają głowice. Na każdą powierzchnię talerza przypada po jednej głowicy dla odczytu i dla zapisu. Umieszczone są one na elastycznych ramionach (tzw. pozycjonerach), które ustawiają głowice w odpowiedniej pozycji względem obracających się talerzy. Napęd ramion głowic realizowany jest najczęściej cewkami wzorowanymi na układach magnetodynamicz-

	512B sektor	4KB sektor
Dane użytkownika	512B	4096B
Kod korekcyjny (ECC)	50B	100B
Sumaryczna wielkość bloku	577B	4211B
Efektywność kodów korekcyjnych	0,887	0.973

Tabela 3.2: Porównanie formatów dysku HDD

nych stosowanych m.in. w głośnikach. Takie rozwiązanie umożliwia szybkie i precyzyjne umieszczanie głowic w zadanej pozycji, co skutkuje czasami dostępu do danych na poziomie kilkudziesięciu milisekund. Razem z mechaniczną częścią dysku twardego znajduje się układ sterujący pracą dysku i przetwarzający dane. W skład układu wchodzą pamięć ROM (ang. Read-only Memory) zawierająca oprogramowanie proceduralne, interfejs SATA (ang. Serial Advanced Technology Attachment) oraz bufor danych zrealizowany za pomocą pamięci RAM.

Dysk twardy ze względu na swoją budowę jest urządzeniem blokowym. Oznacza to, że system operacyjny może odczytać lub zapisać jeden lub więcej sektorów. Sektor dzieli się na dwie części. Pierwsza część przechowuje dane i jest dostępna dla użytkownika. Druga część przechowuje kod korekcyjny ECC (ang. Error Correcting Code) i jest dostępna tylko z poziomu oprogramowania samego dysku. Wykorzystywana jest po to, aby korygować błędy podczas odczytu. W starszych modelach wielkość sektora wynosiła 512B. Nowsze modele ze względu na większą pojemność dysku oraz na zwiększoną efektywność kodów korekcyjnych posiadają format AF [76] (ang. Advanced Format) cechujący się sektorami o wielkości 4KB. Tabelka 3.2 opisuje różnice pomiędzy formatami sektorów. Dyski HDD charakteryzują się o wiele wolniejszym odczytem i zapisem losowym niż sekwencyjnym. Wynika to z faktu, że zanim przeprowadzimy operację na sektorze danych, musimy poczekać aż głowica przesunie się w odpowiednie miejsce i zacznie właściwą pracę. Zatem o wiele szybciej zapiszemy duży plik o wielkości 1MB niż 10 plików o wielkości 100KB. Mimo, że minimalną jednostką zapisu jest sektor, to podczas nadpisywania danych kontroler dysku sprawdzi bieżący stan i nadpisze tylko te bajty, które się zmieniły.

Podsumowując dyski twarde HDD cechują się dużą pojemnością i stosunkowo niską ceną. Jednak ze względu na ich budowę (obrotowe talerze i głowice) jest to niezwykle delikatne i wrażliwe urządzenie. Podczas pracy z dyskami HDD należy zwrócić szczególną uwagę na tryb zapisu i odczytu. Sekwencyjne operacje są o wiele szybsze od losowych. Warto jednak nadmienić, że w przeciwieństwie do nowoczesnych dysków SSD, nadpisywanie danych można wykonać w miejscu (bez potrzeby usuwania bieżących danych w sektorze). Dzięki temu operacja ta jest tak samo szybka jak zwykły zapis. Właściwości tego nośnika pamięci masowej bardzo dobrze wykorzystuje drzewo B+ [77], które przeanalizujemy w kolejnych rozdziałach.

3.3 Pamięć typu flash

Pamięci typu flash są popularną odmianą pamięci nieulotnych EEPROM (ang. Electrically Erasable Programmable Read-only Memory). Są obecne w niemal każdym urządzeniu elektronicznym - od kilkukilobajtowych pamięci przechowujących oprogramowanie sterowników (ang. firmware) przez kilkumegabajtowe magazyny w systemach wbudowa-

	NOR	NAND
Maksymalna pojemność	niska	wysoka
Cena za bit	wysoka	niska
Niezawodność	bardzo wysoka	wysoka
Liczba błędów spowodowana degradacją	niska	średnia
Pobór energii w czasie pracy	duży (około 160mA)	mały (około 50mA)
Pobór energii w stanie spoczynku	mały (około 200μA)	duży (około 1mA)
Odczyt losowy	szybki	wolny
Zapis losowy	wolny	szybki
Kasowanie	bardzo wolne	wolne
Preferowane użycie	pamięć programu	pamięć masowa

Tabela 3.3: Porównanie typów pamięci flash NOR i NAND

nych, aż do dużych kart pamięci i dysków SSD (ang. Solid State Data) o pojemnościach dochodzących do kilkunastu terabajtów. Pamięć Flash została opracowana w laboratoriach Toshiby przez Dr. Fujio Masuoka na początku lat '80 ubiegłego wieku. Zasada działania opiera się na przechowywaniu informacji w tranzystorach polowych MOSFET [78]. Flash jest pamięcią blokową. Operacje odczytu i zapisu można wykonać tylko na dużych ciągłych obszarach pamięci - stronach. Zazwyczaj strona ma pojemność 2KB-8KB. Flash posiada również inne ograniczenia wynikające z architektury pamięci. Nie można zmienić wartości bitu z '0' na '1'. Aby nadpisać stronę należy wykonać operację usuwania (ang. erase). Operacja ta musi zostać wykonana na całym bloku. Blok zawiera od 32 do 128 stron. Z tego ograniczenia wynika ogromna asymetria pomiędzy szybkością losowego odczytu a losowego zapisu [79]. W przeciwieństwie do dysków twardych, pamięć flash nie posiada żadnych ruchomych części. Tak więc dostęp do dowolnego miejsca w pamięci jest tak samo szybki. Jednak operacja zapisu może wywołać kosztowne wymazanie całego bloku, które jest stosunkowo wolne. Z tego powodu wiele metod optymalizacji zostało zaproponowanych dla pamięci flash. Niektóre z nich skupiaja się na częściowej zmianie architektury komórek pamięci [80], [81], [82]. Innym nurtem optymalizacji jest implementacja dodatkowego buforowania w warstwie sterowników [83], [84]. Ze względu na tak duże asymetrie pomiędzy szybkością odczytu i zapisu oraz odmienną od klasycznych dysków HDD charakterystykę, wiele klasycznych algorytmów i struktur danych zostało dostosowanych do tego rodzaju pamięci [85], [86], [87].

3.3.1 Flash typu NOR

Pamięć flash ze względu na wykorzystywaną architekturę bramek dzieli się na pamięć NAND i NOR. Tabelka 3.3 przedstawia główne różnice pomiędzy tymi typami. Pamięć NOR jest starszym typem pamięci flash. W konfiguracji obwodu wewnętrznego pamięci NOR poszczególne komórki pamięci są połączone równolegle. W związku z tym dostęp do danych można uzyskać w losowej kolejności. Architektura pamięci NOR zapewnia wystarczającą liczbę linii adresowych do zmapowania całego zakresu pamięci. Daje to zaletę w postaci szybkiego losowego dostępu i krótkich czasów odczytu, co czyni ją idealną do zapisywania wykonywanego kodu dla niewielkich urządzeń wbudowanych. Główną wadą pamięci NOR jest większy rozmiar komórki pamięci, co skutkuje wyższym kosztem produkcji oraz niższymi prędkościami zapisu i usuwania danych z pamięci.

3.3.2 Flash typu NAND

W architekturze NAND bloki są połączone sekwencyjnie. Rozmiary bloków wynoszą od 8KB do 32KB i są mniejsze od bloków w architekturze NOR wynoszących 64KB-256KB. To pozwala na zwiększenie prędkości odczytu, zapisu i usuwania. Ponadto urządzenia NAND są połączone za pomocą skomplikowanego szeregowo połączonego interfejsu, który może się różnić w zależności od producenta. Strukturalnie architektura NAND została zaprojektowana z myślą o zoptymalizowanej litografii o wysokiej gęstości, jako kompromis między możliwością dostępu swobodnego do mniejszych rozmiarów bloków. To sprawia, że pamięć NAND jest tańsza pod względem kosztu na wolumin. Teoretycznie gęstość bitów na komórkę pamięci NAND jest dwukrotnie większa niż w pamięci NOR.

Pamięć ta ma znacznie mniejszy rozmiar komórki i znacznie wyższe prędkości zapisu i usuwania danych w porównaniu do pamięci NOR. Do wad należy zaliczyć niższą prędkość odczytu oraz konieczność implementacji interfejsu pośredniczącego z uwagi na konieczność odwzorowania operacji WE/WY. Interfejs ten nie pozwala na losowy dostęp do pamięci, co nieznacznie utrudnia jej obsługę. Należy zauważyć, że wykonanie kodu z pamięci NAND realizuje się poprzez kopiowanie zawartości pamięci do RAM. Jest to więc zasadnicza różnica w stosunku do pamięci NOR, gdzie kod wykonuje się bezpośrednio. Kolejną poważną wadą jest obecność uszkodzonych bloków. NAND zazwyczaj ma tylko 98% dobrych bitów, gdy jest dostarczany w stanie fabrycznym. Dodatkowo, w całym czasie eksploatacji modelu powstają dodatkowe uszkodzenia w blokach pamięci. Dlatego też konieczne jest wykorzystanie funkcji korekcji błędów (ECC) w urządzeniu, a także dedykowanego kontrolera, który eliminuje uszkodzone bloki z dalszej eksploatacji. Tabelka 3.4 przedstawia różnicę pomiędzy typami pamięci NAND: SLC(ang. Single-Level Cell), MLC(ang. Multi-Level Cell), TLC(ang. Triple-Level Cell) i QLC(ang. Quad-Level Cell).

Pamięć NAND z komórkami jednopoziomowymi (SLC) przechowuje tylko po 1 bicie informacji na komórkę. W efekcie komórka przechowuje wartość 0 lub 1, co pozwala szybciej zapisywać i odczytywać dane. Pamięć typu SLC zapewnia najlepszą wydajność i najwyższą wytrzymałość rzędu 10^5 cykli usuwań, przez co może służyć dłużej niż inne rodzaje pamięci NAND. Jednak niska gęstość zapisu danych sprawia, że SLC jest najdroższym rodzajem pamięci NAND i dlatego nie jest często stosowana w urządzeniach powszechnego użytku. Zwykle wykorzystuje się ją w serwerach i innych zastosowaniach przemysłowych, które wymagają szybkości i wytrzymałości. Pamięci wielopoziomowe typu MLC, TLC i QLC potrafią przechowywać odpowiednio 2, 3 i 4 bity w pojedynczej komórce. Większa gęstość danych utrudnia ich zapis i odczyt oraz zwiększa wrażliwość na błędy podczas zapisu. Z drugiej strony takie wykorzystanie komórek pamięci zwiększa pojemność nośnika danych niewielkim kosztem. Pamięci wielopoziomowe ze względu na cenę stosuje się głównie w urządzeniach powszechnego użytku, w których wytrzymałość nośnika danych nie jest kluczowa.

3.3.3 FTL

Każda komórka pamięci flash ma ograniczoną żywotność. Operacja usuwania jest głównym czynnikiem powodującym niszczenie pamięci. Każdy blok może wytrzymać określoną liczbę cykli usuwania. Wytrzymałość pamięci zależy od jej typu. Pamięć SLC potrafi wytrzymać nawet 100tys. usuwań, a pamięć QLC tylko tysiąc. System operacyjny posiada program (logger) do zapisywania wszystkich ważnych informacji z pamięci ulotnej RAM do pliku znajdującego się na pamięci masowej, np. flash. Dzięki temu w razie awarii

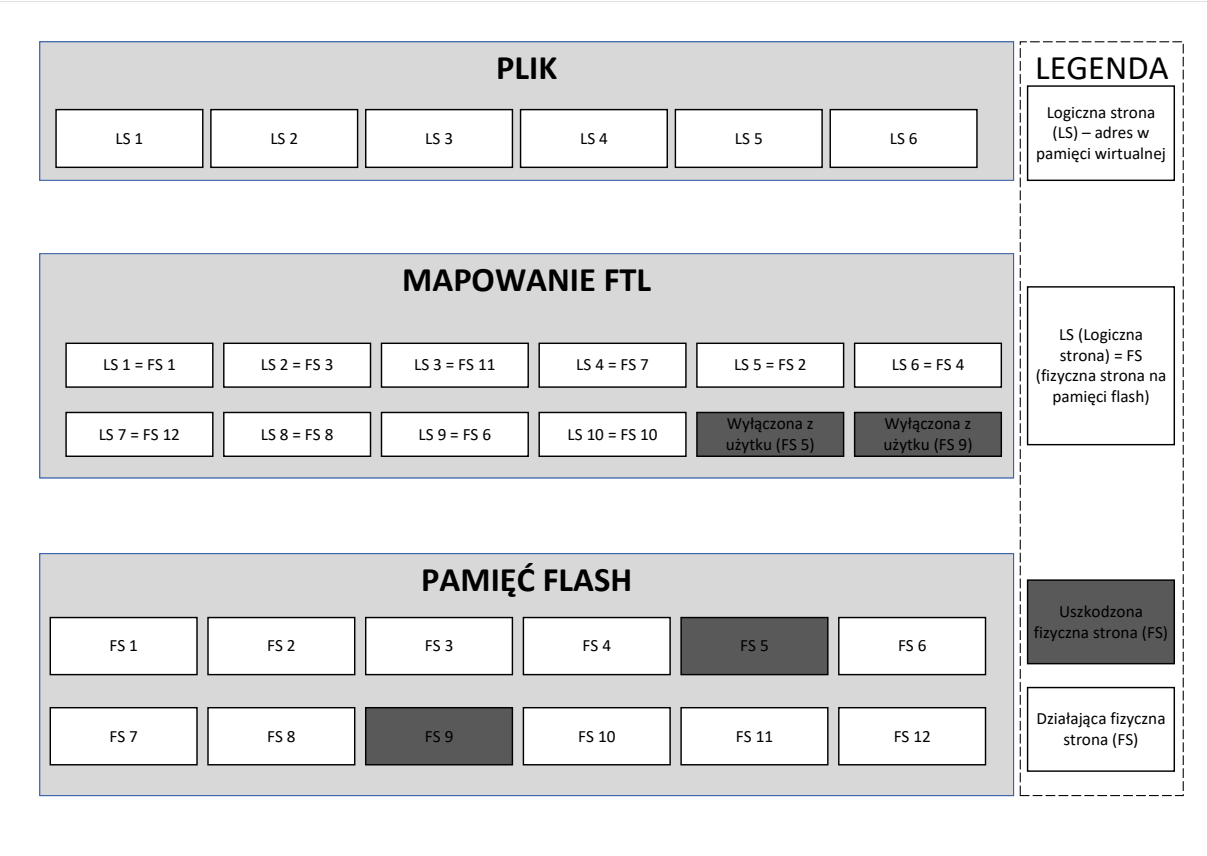
	SLC	MLC	TLC	QLC
Liczba cykli kasowania	>100000	>30000	<5000	1000
Czas przechowywania danych	>20lat	>10lat	<3lata	<1rok
Niezawodność	bardzo wysoka	wysoka	niska	niska
Degradacja danych	niska	średnia	wysoka	wysoka
Cena za 1GB	wysoka	średnia	niska	niska
Pojemność	niska	średnia	wysoka	bardzo wysoka

Tabela 3.4: Porównanie typów pamięci flash NAND

systemu lub braku zasilania, informacje nie zostaną utracone. Gdyby wyłączyć wszystkie mechanizmy maksymalizujące żywotność pamięci flash w systemie operacyjnym, w ciągu sekundy niszczylibyśmy kilka bloków, zmniejszając tym samym pojemność naszej pamięci. Aby uniknąć takiej sytuacji wprowadzono mechanizm wirtualizacji pamięci flash. FTL (ang. Flash Translation Unit) jako mechanizm równomiernego rozłożenia pracy na całej pamięci flash został opatentowany w USA w 1995 [88]. Jego działanie opierało się na prostej wirtualizacji pamięci fizycznej.

Wyróżniamy dwa główne rodzaje systemów FTL. Pierwszy to mapowanie stronicowe. Każda fizyczna strona posiada swój odpowiednik logiczny w systemie FTL. Takie mapowanie zużywa sporo pamięci RAM (około 6MB RAM na każdy 1GB pamięci flash) [89]. Zaletą tego sposobu jest wręcz idealne rozłożenie pracy na całą pamięć flash, ponieważ posługujemy się w wirtualizacji minimalną jednostką dostępu do tej pamięci. Ze względu na duże zużycie pamięci RAM, co jest problemem w przypadku urządzeń wbudowanych, opracowano drugi sposób wirtualizacji - wirtualizację blokową [90]. FTL blokowy mapuje całe bloki fizyczne na sektory logiczne pamięci wirtualnej. Tablica mapująca jest stosunkowo mała (około 0.2MB RAM na każdy 1GB pamięci flash) [89]. Z drugiej strony wydajność tego systemu jest niższa od FTL stronicowego. Ze względu na wady i zalety każdej metody wprowadzono również metody hybrydowe [91], [92], [93], które wykorzystywane są w dużych skomplikowanych systemach operacyjnych jak Windows czy Linux.

Rysunek 3.3 przedstawia działanie prostego mechanizmu FTL stronicowego. Każda strona fizyczna (FS) posiada swój odpowiednik logiczny LS w tablicy mapowania FTL. Jeśli strona fizyczna jest uszkodzona (FS5 oraz FS9 zaznaczone ciemnym kolorem), zostaje przesunięta na koniec w tablicy mapowania i jest wyłączona z użytku. Dzięki temu dostępna pamięć ma ciągłe adresy logiczne. Gdy pracujemy na pliku, wykorzystujemy adresy logiczne zamiast fizycznych. Wydawać by się mogło, że plik jest zapisany w ciągłym miejscu pamięci, jednak zazwyczaj nie jest to prawdą. W rzeczywistości plik może być fizycznie rozmieszczony w kilku miejscach. Gdy nadpiszemy fragment pliku, wtedy FTL pod logiczny adres strony podstawi pierwszy wolny adres strony fizycznej. Wolna strona to fragment pamięci flash, która nie posiada żadnych danych użytkownika, a więc została w przeszłości skasowana i nie zapisana ponownie. Dzięki temu mapowanie FTL drastycznie zmniejsza liczbę wykonywanych operacji kasowania. Gdybyśmy pisali po fragmencie LS1 10 razy, to musielibyśmy wywołać 10 razy operację kasowania na FS1 (ponieważ LS1=FS1) oraz wszystkich stronach znajdujących się w tym samym bloku. Ponieważ za każdym razem bierzemy nową stronę, która jest pusta, wykonujemy kasowanie tylko wtedy, gdy skończą nam się puste strony. Po każdym usuwaniu do puli pustych stron dodajemy nawet 32-64 nowe strony (czyli wszystkie skasowane strony należące do bloku), ponieważ tyle zazwyczaj znajduje się w pojedynczym bloku. W naszym przykła-



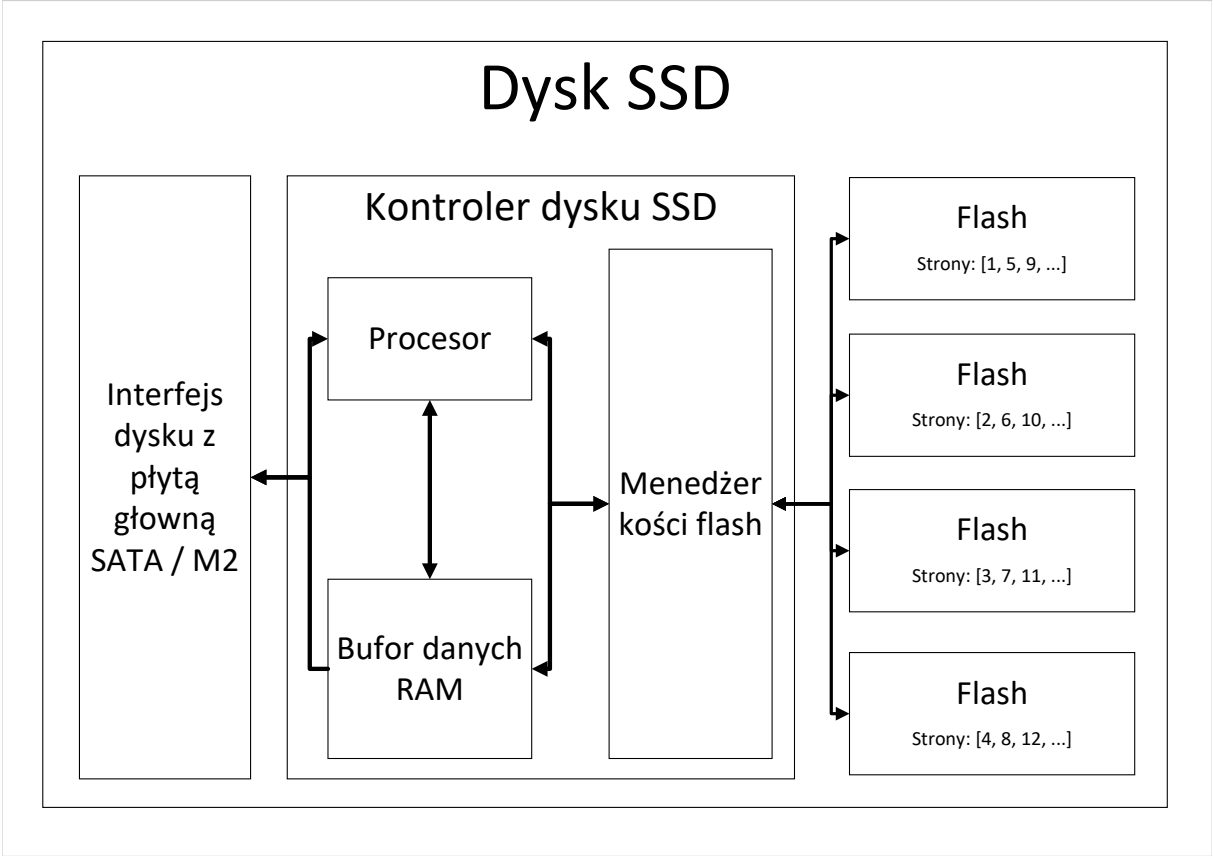
Rysunek 3.3: Działanie FTL

dzie, zamiast 10 operacji wymazania, przeprowadzilibyśmy tylko jedną. Zatem taki prosty mechanizm wirtualizacji pamięci pozwala na przeprowadzenie nawet o 64 usunięć mniej. Oprócz powyższej funkcjonalności, FTL musi równomiernie rozkładać strony do usuwania (ang. wear-leveling), tak aby nie zniszczyć szybko pamięci. Gdy mapowanie wybiera pustą stronę, bierze stronę znajdującą się w bloku o najmniejszym liczniku kasowań EC (ang. erase counter). Używanie licznika EC jest klasycznym podejściem, które jest wykorzystywane w większości implementacji FTL ze względu na jego dokładność [94], [95], [96], [97]. Alternatywą do przechowywania wartości EC na każdym bloku jest używanie estymowanych liczników [98], [99], które nie wymagają przechowywania fizycznie w bloku. Jednak wadą tego podejścia jest o wiele mniejsza dokładność, co dyskredytuje je do zastosowań komercyjnych.

3.4 SSD

Dyski SSD (ang. Solid State Disk) posiadają kompletnie odmienną charakterystykę od tradycyjnych dysków twardych (HDD) [100], [101]. Nie zawierają żadnych ruchomych części, a więc nie trzeba czekać aż głowica dysku przesunie się na odpowiednie miejsce tak jak w przypadku dysków HDD. Dzięki temu czas odczytu i zapisu jest bardzo szybki. SSD zyskały na popularności głównie ze względu na: szybko rosnącą pojemność (obecnie kilku terabajtów), spadek kosztów produkcji, dobrą odporność na czynniki fizyczne, mały pobór mocy oraz bardzo dużą szybkość odczytu i zapisu.

Dyski SSD mogą zapisywać dane z wykorzystaniem wszystkich rodzajów pamięci typu NAND. Wybór pamięci zależy od zastosowań dysku. Pamięć SLC używana jest w drogich modelach przeznaczonych do użytku profesjonalnego, jak serwery czy hurtownie danych, gdzie szybkość wykonywanych operacji oraz bezpieczeństwo danych są bardzo ważne. Z kolei pamięć MLC stosowana jest powszechnie stosowana w dyskach przeznaczonych do laptopów, czy komputerów osobistych. Ponieważ dyski SSD używają pamięci flash, to ich charakterystyka jak i ich ograniczenia są identyczne jak w przypadku pamięci NAND . Obecnie dyski używają dwóch typów interfejsów: wolniejszego SATA (ang. Serial Advanced Technology Attachment) [102], [103] o przepustowości maksymalnej 750MB/s oraz szybszego M.2 (ang. Next Generation Form Factor) [104] wykorzystującego złącza PCIe4 (ang. Peripheral Component Interconnect Express) o maksymalnej przepustowości wynoszącej aż 4GB/s. Szybszy interfejs M.2 może zostać użyty w protokole NVMe (ang. Non-Volatile Memory Express) [105], [106], który jest obecnie najpopularniejszym protokołem do komunikacji pomiędzy SSD a głównym procesorem. Dyski SSD mają wbudowany procesor oraz bufor danych, aby maksymalnie wykorzystać przepustowość używanego interfejsu.



Rysunek 3.4: Budowa dysków SSD

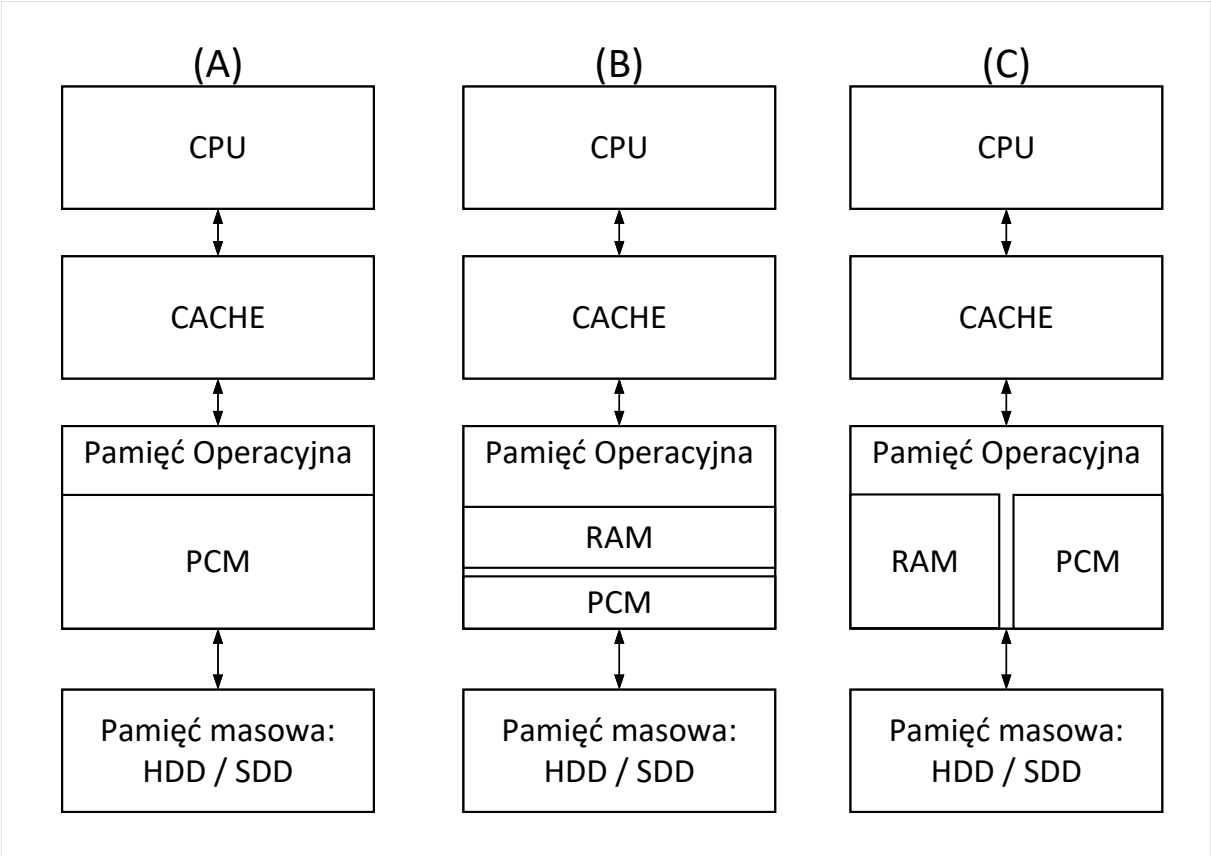
Rysunek 3.4 przedstawia uproszczony model dysku SSD. Przykładowy dysk ma wbudowane 4 kości flash typu NAND. Menedżer pamięci flash często ma wbudowany algorytm FTL [107], który rozwiązuje problem szybkiego zużywania się pamięci (ang. wear-leveling). Aby w pełni wykorzystać potencjał używanych kości pamięci, menedżer mapuje strony równolegle tak, aby móc jednocześnie pracować na wielu kościach. W naszym przykła-

dzie, strony logiczne 1,5,9 ... wskazują na kolejne strony kości pierwszej, strony 2, 6, 10 ... wskazują na strony kości drugiej. Analogiczne mapowanie przeprowadzone jest dla pozostałych kości. Dzięki takiemu ułożeniu pamięci wirtualnej możemy równolegle wczytywać lub zapisywać dane, gdy użytkownik z góry zna zakres adresów, na których pracuje. Ponieważ taki układ w pamięci wirtualnej widzianej przez użytkownika końcowego jest sekwencyjny (fizycznie równoległy), tryb pracy kontrolera, który pozwala na odczyt lub zapis na wielu kościach jednocześnie nazywamy trybem sekwencyjnym. Praca na ciągłym obszarze pamięci jest o wiele wydajniejsza właśnie dzięki użyciu trybu sekwencyjnego przez kontroler [108]. Dlatego efektywne algorytmy projektowane z myślą o pracy na dysku SSD maksymalizują użycie tego trybu jednocześnie minimalizując liczbę losowych dostępów [109], [110]. Aby przyspieszyć tryb losowy, kontroler dysku posiada także swój własny bufor RAM. Za jego pomocą implementowany jest podobny mechanizm buforowania jak w przypadku pamięci podręcznej procesora (ang. cache) [111]. Gdy użytkownik prosi o dane z podanego adresu, procesor wbudowany w dysk SSD przegląda cały bufor danych w poszukiwaniu potrzebnych stron. Jedynie strony, które nie znalazły się w buforze, są wczytywane z pamięci flash [112]. Cały mechanizm jest bardzo podobny do omawianego wcześniej mechanizmu Page Cache.

Kolejnym ważnym mechanizmem przyspieszającym działanie dysków SSD jest automatycznie usuwanie bloków (ang. garbage collection) [113] w momencie, gdy dysk nie jest używany przez użytkownika lub jest używany minimalnie. Algorytm FTL dzieli strony na dwa podzbiory: strony puste - usunięte wcześniej, gotowe do ponownego użytku, oraz strony brudne (ang. dirty), które nie są już używane i oczekują na usunięcie. Ponieważ operacja zapisu jest najszybsza wtedy, gdy strona jest pusta i gotowa do użytku, procesor wbudowany w kontroler dysku SSD usuwa zużyte strony, gdy tylko jest taka możliwość (znikome wykorzystanie dysku). Algorytm odśmiecania pamięci jest bardzo ważnym elementem dysków SSD. Zaproponowano wiele różnych implementacji tego algorytmu jak i wiele optymalizacji skupiających się głównie na dostosowaniu się do sposobu używania dysków, tak aby operacja usuwania bloków nie odbywała się podczas pełnego obciążenia nośnika pamięci [114], [115], [116], [117], [118]. Gdy dysk jest pełny (wszystkie strony mają zapisane dane od użytkownika), szybkość działania dysku drastycznie maleje [119], ponieważ nie ma pustych stron. Przed każdym zapisem musimy czekać na operację kasowania bloku. Z tego powodu producenci dysków SSD zalecają zapis danych do maksymalnie 80% pojemności SSD. Nowe systemy operacyjne wspierają także operacje TRIM, czyli manualne odśmiecanie pamięci. System operacyjny ma więcej danych o obłożeniu całego komputera [120], [121], [122], dlatego często potrafi lepiej zdecydować, kiedy warto włączyć kasowanie zużytych stron. Gdy podejmie taką decyzję, wysyła komendę TRIM do dysku SSD, a ten niezależnie od używanego algorytmu odśmiecania, zaczyna usuwać niepotrzebne już bloki.

3.5 PCM

Pamięć zmiennofazowa PCM (ang. Phase Change Memory) [123], [124] jest pamięcią nieulotną opartą na nośniku krystalicznym. Autorem koncepcji i prototypów pamięci zmiennofazowej jest firma Ovonyx, od której pochodzi nazwa tej pamięci: ovonic unified memory (OUM) [125]. Obecnie rozwojem tego rodzaju pamięci zajmuje się firma Intel [126].



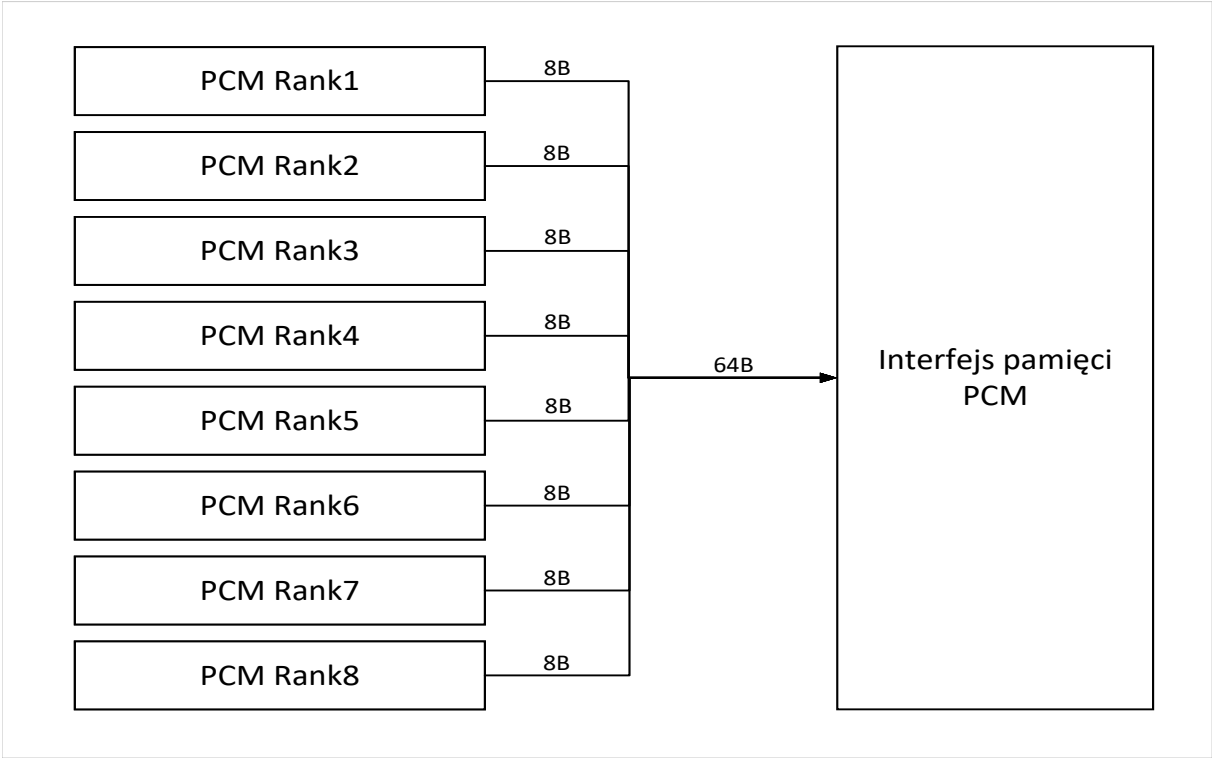
Rysunek 3.5: Możliwe zastosowania pamięci PCM

Podobnie jak pamięć flash, PCM nie traci danych po odłączeniu zasilania. Pamięć ta jest dużo bardziej wydajna od dysków HDD i SSD. Wartości dotyczące szybkości zapisu i odczytu bitów informacji szacuje się średnio na poziomie kilkadziesiąt razy wyższym niż w przypadku NAND. Dokładne informacje na temat szybkości pamięci przedstawia tabelka 3.1. Podczas zapisu danych nie istnieje konieczność wymazywania całego bloku komórek. Dlatego pamięć ta może zostać wykorzystana nie tylko jako pamięć masowa, ale także jako pamięć operacyjna komputera [127], [128], [129].

Rysunek 3.5 przedstawia możliwe zastosowania pamięci PCM zaproponowane w [130], [131], [132]. W propozycji *(A)* pamięć operacyjna jest w pełni realizowana za pomocą PCM. Pamięć zmiennofazowa jest wolniejsza od pamięci RAM, jednak pojedyncza kość pamięci jest pojemniejsza. Obecnie kości Intel Optane DC posiadają pojemność od 64GB aż do 256GB. Dlatego gdy potrzebujemy pamięci operacyjnej o dużej pojemności, warto wybrać PCM jako pamięć główną komputera. W [130] zaproponowano algorytm wykorzystujący charakterystykę pamięci PCM, który posiada czas wykonania zestawu operacji tylko o 20% większy od operacji przeprowadzanych na klasycznej pamięci RAM DDR4. Propozycje *(B)* i *(C)* dzielą pamięć operacyjną na dwie warstwy. W warstwie RAM, przechowywane są dane często używane lub niedawno użyte. Tak więc pamięć RAM odgrywa rolę bufora pomiędzy pamięcią podręczną procesora a pamięcią PCM. W zaproponowanej architekturze *(B)* użytkownik nie ma możliwości ręcznego buforowania danych. System operacyjny sam podejmuje decyzję, które dane będą znajdować się w buforze. W propozycji *(C)* użytkownik jest odpowiedzialny za buforowanie swoich danych, a system operacyjny zapisuje wszystkie dane do PCM, jako głównej pamięci operacyjnej. Inną zaletą

pamięci PCM jest jej trwałość. Komórki pamięci PCM mogą działać znacznie dłużej niż komórki pamięci flash. Zanim ulegną zdegradowaniu są w stanie wytrzymać około 100 milionów cykli. Dodatkowo komórki PCM też są bardziej stabilne od komórek flash, a informacja na nich zapisana nie ulega degradacji wraz z upływem czasu. Szacuje się, iż podczas pracy w normalnej temperaturze (85 stopni C) komórki przechowają informację przez około 300 lat [133].

Pamięć PCM oparta jest na nośniku krystalicznym. Do przechowywania danych wykorzystano proces zmiany stanu nośnika, który w temperaturze pokojowej może istnieć w postaci amorficznej (logiczne 0) lub krystalicznej (logiczne 1). Zmiana stanu następuje w wyniku podgrzania nośnika przez wyemitowaną wiązkę elektronów. Odczyt przeprowadzany jest także w wyniku działania strumienia elektronów, w oparciu o pomiar rezystancji nośnika, która jest inna dla różnych jego faz. Bardzo ważną cechą tego rodzaju pamięci jest możliwość przechowywania stanów pośrednich. Dzięki temu możemy w jednej komórce zapisać kilka bitów informacji.



Rysunek 3.6: Pamięć PCM

Tak jak jest to w przypadku pamięci RAM, pamięć PCM jest również bajtowo adresowalna. Jednak ze względu na jej budowę oraz lepszą współpracę z pamięcią podręczna procesora, została podzielona na strony o wielkości 64B. Rysunek 3.6 przedstawia uproszczoną budowę pojedynczej kości PCM. Pamięć zmiennofazowa, podobnie jak pamięć RAM podzielona jest na równolegle połączone ze sobą banki pamięci (ang. memory ranks). Zazwyczaj kość zbudowana jest z 4 lub 8 banków pamięci. Obecnie wykorzystuje się głównie 8 banków, ponieważ w czasie jednego odczytu możemy wczytać całą stronę pamięci PCM (64B), odczytując z każdego banku pojedyncze słowo maszynowe architektury 64 bitowej (8B), która jest obecnie najbardziej popularna. Jest to idealne wykorzystanie nie tylko pamięci PCM, ale także pamięci podręcznej procesora, ponieważ strona tej pamięci (ang. cache line) zazwyczaj posiada również wielkość 64B. Zatem podczas jednej operacji

odczytu lub zapisu pracujemy na całej linii pamięci podręcznej co prowadzi do maksymalizacji przepustowości i minimalizacji latencji pomiędzy pamięciami i procesorem. Ze względu na asymetrię pomiędzy czasem odczytu a zapisu oraz odmienną charakterystykę pamięci PCM od klasycznych dysków HDD jak i nowych dysków SSD, klasyczne struktury danych muszą być dostosowane do tego rodzaju pamięci. W [134] przedstawiono implementację drzewa B+, które kosztem większej liczby odczytów minimalizuje liczbę potrzebnych zapisów, zwiększając wydajność indeksu B+ o 30%.

3.6 Podsumowanie

Różne pamięci posiadają różne cechy. Unikalność wielu z nich powoduje zmianę klasycznego podejścia do problemu indeksowania i potrzebę dostosowania struktur danych pod charakterystykę pamięci. Stare dyski twarde HDD ze względu na ich małą przepustowość (około 150MB/s) oraz duża latencję (4.17ms) zostają zastępowane przez nowe rodzaje pamięci masowej jakimi są flash i PCM. Flash typu NAND wykorzystywana jest w dyskach SSD, które nie posiadają ruchomych części, co przekłada się na szybki czas dostępu do dowolnej komórki pamięci (0.36ms dla SATA, 0.005ms dla M2). Ich mała latencja i duża przepustowość oraz malejące koszty produkcji przyczyniły się do coraz szerszego wykorzystywania dysków SSD w serwerach jak i hurtowniach danych. Algorytmy działające na tych dyskach powinny maksymalizować użycie sekwencyjnego trybu kontrolera oraz minimalizować niepotrzebne usuwanie bloków, co jest nie tylko wolną operacją, ale także skracającą żywotność pamięci flash. Kolejnym rodzajem pamięci nieulotnej jest pamięć zmiennofazowa PCM. W przeciwieństwie do dysków HDD i SDD jest bajtowo adresowalna. Posiada bardzo dużą wytrzymałość oraz o wiele krótszy czas dostępu od dysków HDD czy SSD (350ns). Z tego powodu wykorzystywana jest nie tylko jako pamięć masowa, ale także jako pamięć operacyjna komputera w sytuacjach wymagających dużej pojemności pamięci głównej. Zarówno PCM jak i pamięć flash cechuje się asymetrią pomiędzy czasem odczytu i zapisu. Dlatego też algorytmy pracujące na tej pamięci powinny dążyć do minimalizacji niepotrzebnych zapisów.

Platforma testowa

W tym rozdziale omówimy platformę testową: symulator pamięci trwałej, podstawowy zestaw kwerend oparty na zestawach zaproponowanych w [135], [136], rozszerzony zestaw oparty o zbiór operacji testowych TPC-C [137] i TPC-H [138] oraz nowy sposób testowania indeksów częściowych zaproponowany w [139].

Celem platformy testowej jest ujednolicenie eksperymentów przeprowadzanych na indeksach. Każdy indeks powinien zostać przetestowany w tych samych deterministycznych warunkach, przy użyciu tego samego modelu pamięci oraz tych samych operacji. Ponieważ nowoczesne typy pamięci trwałej jak PCM Intel Optane i Intel Optane SSD 3D Xpoint są drogie oraz zawierają wiele technik optymalizacji, które są niedeterministyczne, do przeprowadzenia testów użyto nowatorskiego symulatora pamięci trwałej SIPS (ang. Storage and Index Performance Simulator) stworzonego na potrzeby tej pracy. Wszystkie eksperymenty zostały napisane w języku C++ (standard 17), skompilowane narzędziem g++ w wersji 9.4.0 z najwyższą możliwą optymalizacją (O3). Programy zostały uruchomione na systemie operacyjnym Ubuntu 20.04. Komputer, na którym przeprowadzono eksperymenty wyposażony jest w ośmiordzeniowy procesor intel i7-9700F, każdy rdzeń o taktowaniu maksymalnym 4700MHz. Pamięć podręczna procesora poziomu pierwszego (L1 cache) ma pojemność 2MB, pamięć podręczna współdzielona pomiędzy wszystkie rdzenie (L3 cache) ma pojemność 12MB. Pamięć operacyjna RAM o sumarycznej pojemności 32GB, składa się z dwóch kości 16GB połączonych ze sobą w trybie dual-channel pracujące z częstotliwością 2666MHz.

4.1 Symulator SIPS

Obecny postęp technologiczny sprawił, że przeprowadzenie rzetelnych eksperymentów jest bardzo trudne. Producenci dysków nie udostępniają pełnej specyfikacji sprzętu wraz z mechanizmami działającymi w tle na ich dyskach, rozmiarem bufora, modelem procesora i algorytmem buforującym przesyłane dane. Bardzo prawdopodobne jest, że algorytm będzie działał sprawniej na dyskach jednego producenta niż na dyskach drugiego, ponieważ implementacja indeksu będzie nieświadomie i pośrednio wykorzystywać wbudowaną optymalizację. Przykładowo, indeks może korzystać z dysku w taki sposób, że potrzebne dane będą znajdować się często w buforze wbudowanym, ponieważ bufor zarządza danymi w podobny sposób co wzorzec odczytów wykorzystywany przez indeks. Dobrym przykładem jest algorytm MRU buforowania danych, który przechowuje w buforze najczęściej wczytywane dane. W tym przypadku górne fragmenty indeksu B+ będą znajdować się w buforze i nie będzie potrzeby fizycznego odczytu z pamięci. Gdyby ten algorytm zmienić na LRU, który przetrzymuje ostatnio żądane fragmenty dysku, wówczas górny fragment drzewa B+ byłby poza buforem, ponieważ ścieżka z korzenia do kluczy nie jest zawsze taka sama, ale zmienia się w zależności od klucza. W takim przypadku dysk musiałby wykonać więcej operacji odczytu, co spowolniłoby prace na indeksie. Wi-

dać zatem, że sposób implementacji algorytmów wbudowanych w dysk jak i dodatkowe mechanizmy wpływające na efektywność pracy na pamięci zaburzają wyniki eksperymentów.

Obecnie każdy dysk SSD i PCM posiada własny procesor. Wykonuje on wiele operacji w tle nie informując o tym użytkownika. Dlatego wynik testu zależy od stanu dysku. Przykładowo, jeśli dysk SSD miałby wszystkie wolne strony skasowane, to zapis nowych danych mógłby się odbyć natychmiastowo. Jednak gdy proces *garbage collector* nie zdążyłby wyczyścić potrzebnych stron, wtedy operacja zapisu musiałaby poczekać aż proces skończy swoją pracę. Ponieważ nowoczesne dyski posiadają kilkuprocentowy bufor (np. użytkownik kupując dysk 1TB ma do dyspozycji tylko 920GB-950GB), to bardzo ciężko jest powtórzyć test i trafić na ten sam stan dysku co poprzednio. Kolejnym aspektem, który należy wziąć pod uwagę jest cena. Ze względu na wcześniej wymienione optymalizacje, należy przeprowadzić testy na kilku dyskach różnych producentów. Komputer wspierający pamięci PCM Intel Optane w chwili pisania tej pracy kosztuje ponad 30 tys. złotych, nowoczesny dysk SSD kosztuje ponad 2000 złotych. Ze względu na tak duży koszt jak i niedeterministyczne zachowanie realnych pamięci, do przeprowadzenie eksperymentów napisano symulator SIPS (ang. Storage and Index Performance Simulator).

Dyski SSD i pamięć flash zyskały mocno na popularności. Wiele symulatorów zostało stworzonych, aby móc testować zaproponowane algorytmy. Symulatory możemy podzielić na dwie główne kategorie: sprzętowe (ang. hardware simulators) i programowe (ang. software simulators). Symulatory sprzętowe [140], [141], [142], [143] działają na urządzeniach FPGA (ang. Field Programmable Gate Array), które próbują symulować kontroler dysków w czasie rzeczywistym. Oznacza to, że jeśli zapis na dysku powinien trwać 2 minuty, to urządzenie zgłosi koniec operacji dopiero po 2 minutach, chociaż symulacja zakończyła się w kilka sekund. Taki rodzaj urządzeń jest ciężko dostosować do swoich potrzeb, a konfiguracja parametrów dla kilku dysków jak i połączenie ich ze środowiskiem głównego komputera to również bardzo trudne zadanie. Dodatkowo sama symulacja zabiera tyle czasu co prawdziwa operacja, co nie jest efektywne podczas testowania wielu algorytmów indeksowania dla dużych baz danych na kilku rodzajach pamięci flash.

Programy symulujące dyski SSD mogą działać osobno jako aplikacja lub jako komponent w większym emulatorze całego sprzętu QEMU [144], [145]. FEMU [146], VSSIM [147] i SimpleSSD [148] to najpopularniejsze programy działające w środowisku emulatora QEMU. Mają one za zadanie symulować głównie funkcjonalność dysku, mechanizmy działające w środku kontrolera oraz zużycie samego dysku. Zatem te symulatory nie powinny być stosowane do testowania algorytmów pod kątem ich efektywności. Ostatnia kategoria to programy działające jako osobne aplikacje. Wystawiają one API dla użytkowników. Jeśli chcemy z nich korzystać, musimy dostosować implementacje algorytmów tak, aby zamiast prawdziwych operacji na systemie plików, używać operacji zdefiniowanych w symulatorze. Do najpopularniejszych z nich należą: MQSim [149], SSDModel [150], DiskSim [151], DiskSimExt [152], CPPSim [153], FlashSim [154], FlexSim [155], NandFlashSim [156], WiscSim [157] i SSDSim [158].

W [149] porównano najpopularniejsze symulatory dysków SSD względem czterech dysków o interfejsie SATA III. Najmniejszy błąd względny ma symulator MQSim: rzędu 10-20% w zależności od modelu dysku. Pozostałe modele mają błąd nawet 300% względem rzeczywistego czasu operacji. Różnice wprost wynikają z liczby komponentów, które potrafi symulować program: im jest ich więcej tym wynik będzie dokładniejszy. Dodatkowo

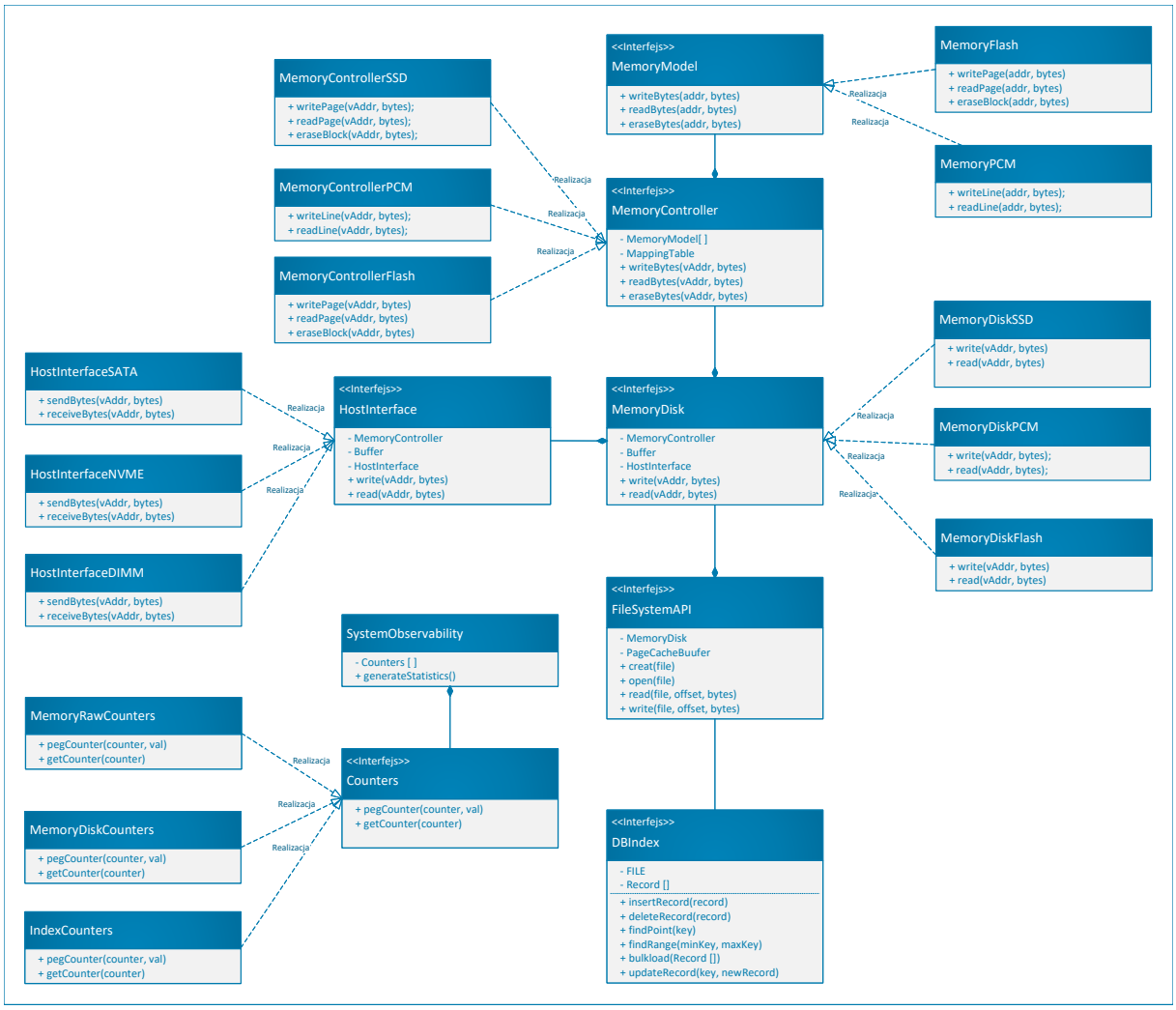
Symulator	Liczba linii kodu	Błąd względny			
		SSD A	SSD B	SSD C	SSD D
MQSim	13 tys.	8%	6%	18%	14%
SSDModel	1 tys.	91%	155%	196%	136%
FlashSim	8 tys.	99%	259%	310%	138%
SSDSim	5 tys.	70%	68%	74%	85%
WicSim	7 tys.	95%	277%	324%	135%
FEMU	7 tys.	-	-	-	-
VSSim	6 tys.	-	-	-	-
SimpleSSD	7 tys.	-	-	-	-

Tabela 4.1: Porównanie symulatorów SSD [149]

nowoczesne dyski SSD potrafią obsługiwać kilka zapytań jednocześnie. Dlatego też ważne jest, aby symulator potrafił zapewnić również taką funkcjonalność [159]. Ponieważ praca ta nie skupia się tylko na dyskach SSD, ale także na pamięci flash oraz PCM, użycie wyżej wymienionych symulatorów nie jest możliwe. Celem zaproponowanych eksperymentów jak i platformy testowej jest przeprowadzenie testów efektywności algorytmów indeksowania w jednolitym środowisku. Oznacza to, że do symulatora dysków SSD należałoby zintegrować symulator pojedynczej kości flash ([146]) jak i pamięci PCM ([160], [161], [162]).

Zamiast integracji wielu niekompatybilnych ze sobą narzędzi wybrano rozwiązanie napisania nowego symulatora. W celu ujednolicenia środowiska testowego dla wszystkich pamięci i indeksów, które zostaną przetestowane w tej pracy, zaimplementowano symulator SIPS. Symulator jest podobny do systemu Amber [163], który prócz peryferii dysku symuluje także warstwy systemu operacyjnego odpowiedzialnego za przesył danych pomiędzy dyskiem a pamięcią RAM oraz buforowanie danych za pomocą komponentu Page Cache. Rysunek 4.1 przedstawia schemat klas symulatora SIPS. Idąc od komponentów znajdujących się najniżej w hierarchii klas mamy *MemoryModel*, który odpowiedzialny jest za symulację niskopoziomowego układu pamięci. W tym komponencie nie mamy buforów, wirtualizacji pamięci i innych mechanizmów minimalizujących zużycie pamięci. Na kościach pamięci pracuje *MemoryController* i to on jest odpowiedzialny za wirtualizację pamięci oraz rozkład jej zużycia (ang. wear-leveling). Do tego celu jest używany algorytm FTL i jego odpowiednik dla pamięci PCM - PTL [164]. Odpowiada on także za podstawowe buforowanie i agregacje zapytań do pamięci. Kontroler zna charakterystykę pamięci, na której pracuje, wie że podzielona jest na segmenty, na których musi operować. Dzięki temu może zamienić kilka odczytów z pamięci na tej samej stronie w pojedynczy odczyt całej strony, a pozostałe operacje przeprowadzać na wbudowanym małym buforze danych. *MemoryDisk* to komponent symulujący cały dysk. To w nim znajduje się większość mechanizmów, które fizycznie umieszczone są na procesorze wbudowanym w dyski HDD, SDD czy PCM.

Każdy dysk ma swój własny interfejs z płytą główną, który również trzeba zaimplementować, ponieważ wpływa on na szybkość przesyłu danych. Dodatkowo nowoczesne dyski mają duże bufory RAM pozwalające ograniczać fizyczne dostępy do pamięci poprzez odpowiednie techniki buforowania np. LRU (ang. Last Recently Used), czy MRU (ang. Most Recently Used). Ponieważ symulator SIPS jest bardzo podobny do systemów Amber czy MQSim - jest samodzielnym programem, który nie współpracuje z żadnym emulato-



Rysunek 4.1: Schemat symulatora SIPS

rem lub systemem operacyjnym. Użytkownik implementując algorytm indeksowania musi użyć API systemu plików, które dostarczane jest przez symulator. Z tego względu zdecydowano, aby sam symulator rozbudować także o implementacje indeksów. Zatem SIPS to nie tylko symulator różnych pamięci, ale także symulator wielu indeksów baz danych, które na tych pamięciach pracują. Taki sposób implementacji ma wiele korzyści. Jedną z nich jest prostota implementacji takiego programu, gdyż nie trzeba integrować ze sobą wielu programów poprzez API. Drugą bardzo ważną zaletą jest możliwość implementacji komponentu *SystemObservability*, który zbiera informacje o interesujących nasz operacjach jak i kosztach tych operacji. Tabela 4.2 przedstawia najważniejsze statystyki zbierane przez symulator. W rzeczywistości jest ich o wiele więcej, gdyż zbierane są nie tylko całkowite czasy operacji, ale również ich liczba, średni czas wykonania oraz czas wykonania każdej kwerendy. Dzięki zintegrowaniu modelu pamięci, kontrolera, dysku oraz samego indeksu w jeden program, symulator dostarcza nam wszelkie potrzebne informacje do analizy algorytmów. Wiemy ile czasu nasz algorytm potrzebował na każda operację (dodawanie, usuwanie, wyszukiwanie) oraz znamy zużycie pamięci (ang. wearout) a także wpływ buforowania danych na to zużycie (różnice pomiędzy *MEMORY_COUNTER_VIRTUAL_WEAROUT* a *MEMORY_COUNTER_PHYSICAL_WEAROUT*).

Budując symulator pamięci flash i dysku SSD należy wziąć pod uwagę przynajmniej najważniejsze komponenty tych pamięci wpływające na efektywność ich pracy [165], [166]. W symulatorze SIPS najpierw zaimplementowano dokładnie pojedynczą kość pamięci flash wraz z algorytmami FTL: blokowym i stronicowym wbudowanymi w kontroler tej pamięci. Algorytm FTL można wybrać z puli dostępnych algorytmów lub go wyłączyć, aby zobaczyć wpływ jego działania na efektywność indeksów i zużycie pamięci. Tabela 4.3 przedstawia charakterystykę wybranych modeli pamięci flash.

Prawdziwe dyski SSD posiadają kilka kości flash. Tak więc implementacja klasy dysków SSD korzysta z klasy implementującej działanie kości Flash. Symulator SSD automatycznie buforuje operacje wykonywane na dysku za pomocą algorytmu MRU, dzięki czemu sam przełącza kontroler pamięci flash w tryb sekwencyjny lub punktowy. Ponieważ zależy nam na deterministycznym środowisku testowym, proces kasujący stare dane i czyszczący strony (ang. garbage collector) został zaimplementowany jako funkcja zamiast jako proces w tle. Oznacza to, że zawsze trzeba czekać aż proces skończy swoją pracę, jeśli zabrakło nowych stron do zapisu. Dodatkowo symulator posiada funkcję resetowania stanu dysku do ustawień fabrycznych, dzięki czemu każdy eksperyment może zostać przeprowadzony od tego samego punktu startowego. Tabela 4.4 przedstawia charakterystykę wybranych dysków SSD. Dwa dyski: Samsung 840 i Toshiba VX500 posiadają interfejs SATA, charakteryzują się podobnymi czasami odczytu i zapisu sekwencyjnego oraz dużo wolniejszymi operacjami losowymi. Dysk Intel DXP4511 posiada interfejs NVMe, który dzięki buforowaniu danych w kontrolerze, osiąga rewelacyjne prędkości odczytu jak i zapisu dla trybu sekwencyjnego przy jednocześnie bardzo wolnym zapisie losowym.

Część programu odpowiedzialna za symulację pamięci PCM była wzorowana na symulatorze zaproponowanym w [134]. Ponieważ PCM to pamięć bajtowo adresowalna używana często jako pamięć główna, zastosowano te same mechanizmy co w przypadku pamięci RAM. Pamięć podzielona jest na banki, których jest domyślnie 8. Operacje zapisu jak i odczytu wyrównane są do linii pamięci podręcznej procesora, która najczęściej ma wielkość 64B. Oznacza to, że zapis 1B, 8B jak i 64B zajmuje tyle samo czasu. Do rozkładu zużycia pamięci jak i jej wirtualizacji zastosowano podstawowy algorytm PTL opracowany w [164]. Działa podobnie jak stronicowy algorytm FTL. Dzieli pamięć na strony wielkości od 2KB do 4KB. Wielkość strony domyślnie ustawiona jest na rozmiar strony RAM, czyli 4KB. Ponieważ pamięć zmiennofazowa charakteryzuje się asymetrią pomiędzy czasem zapisu i odczytu, kontroler tej pamięci ma wbudowany mechanizm wykrywania faktycznych zmian i nadpisywania tylko tych bajtów, które różnią się od danych w kolejce do zapisu. Tabela 4.5 przedstawia parametry pamięci PCM wzorowane na [167], w którym sprawdzano szybkość zapisu i odczytu na pierwszej edycji pamięci Intel Optane DC w różnych trybach pracy.

Każdy z dysków posiada także interfejs z płytą główną (w naszym przypadku z system plików), który ze względu na prostotę rozwiązania został zaimplementowany jako sekwencyjny. Oznacza to, że symulator nie wspiera równoległych dostępów do dysku np. za pomocą interfejsu SATA. Takie uproszczenie zostało wprowadzone bez straty na rzetelności wyników, ponieważ celem jest testowanie efektywności indeksów na pojedynczych tabelach. Nie symulujemy silnika baz danych, który potrafi zrównoleglić zapytania do pamięci w celu ich przyspieszenia. Jako dodatkowy komponent zaimplementowano także algorytm buforujący Page Cache, który obecny jest w jądrze systemów operacyjnych.

Nazwa	Opis
INDEX_COUNTER_INSERT_TIME	Całkowity czas potrzebny na wstawienie rekordów do indeksu
INDEX_COUNTER_DELETE_TIME	Całkowity czas potrzebny na usunięcie rekordów z indeksu
INDEX_COUNTER_PSEARCH_TIME	Całkowity czas potrzebny na wyszukiwanie punktowe rekordów w indeksie
INDEX_COUNTER_RSEARCH_TIME	Całkowity czas potrzebny na wyszukiwanie zakresowe rekordów w indeksie
INDEX_COUNTER_TOTAL_TIME	Całkowity czas potrzebny na wszystkie operacje wykonywane na indeksie
MEMORY_COUNTER_READ_TIME	Całkowity czas potrzebny na wszystkie operacje odczytu danych wykonywane na modelu pamięci
MEMORY_COUNTER_WRITE_TIME	Całkowity czas potrzebny na wszystkie operacje zapisu danych wykonywane na modelu pamięci
MEMORY_COUNTER_OVERWRITE_TIME	Całkowity czas potrzebny na wszystkie operacje nadpisania danych wykonywane na modelu pamięci
MEMORY_COUNTER_TOTAL_TIME	Całkowity czas potrzebny na wszystkie operacje wykonywane na modelu pamięci
MEMORY_COUNTER_VIRTUAL_WEAROUT	Całkowita liczba bajtów zgłoszona do zmiany w warstwie kontrolera (bajty zmienione wirtualnie w buforze też są liczone)
MEMORY_COUNTER_PHYSICAL_WEAROUT	Całkowita liczba bajtów zgłoszona do zmiany w warstwie modelu pamięci (bajty zmienione wirtualnie w buforze są pomijane)

Tabela 4.2: Najważniejsze statystyki zbierane przez symulator SIPS

	Pojemność		Prędkość		
Model	Strony	Bloku	Odczytu	Zapisu	Kasowania
Samsung K9F1G08U0D	2KB	64KB	58MB/s	8MB/s	1MB/s
Micron MT29F32G08ABAAA	8KB	1MB	234MB/s	23MB/s	5MB/s
Micron MT29F32G08CBEDBL83A3WC1	4KB	512KB	81MB/s	4,5MB/s	1,1MB/s

Tabela 4.3: Wybrane modele pamięci flash

		Pojemność		Prędkość losowego		Prędkość sekwencyjnego	
Model	Interfejs	Strony	Bloku	Odczytu	Zapisu	Odczytu	Zapisu
Samsung 840	SATA	8KB	512KB	390MB/s	182MB/s	585MB/s	535MB/s
Toshiba VX500	SATA	4KB	256KB	379MB/s	267MB/s	568MB/s	525MB/s
Intel DCP4511	NVMe	4KB	256KB	1,2GB/s	240MB/s	2GB/s	1.47GB/s

Tabela 4.4: Wybrane modele dysków SSD

Algorytm ten można w każdej chwili włączyć. Domyślnie jest on wyłączony, ponieważ większość silników baz danych z niego nie korzysta.

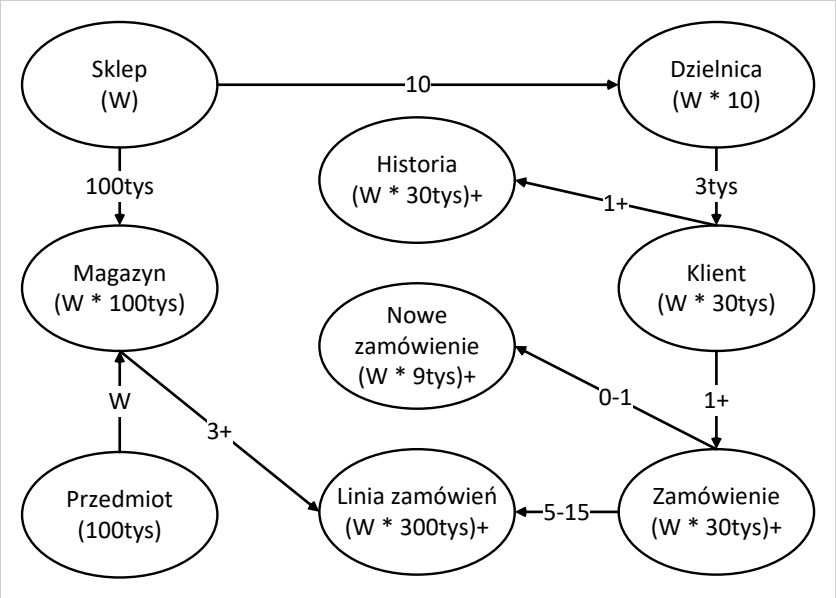
4.2 Zestawy kwerend

Testowe zestawy kwerend wraz ze schematami relacyjnych baz danych TPC-C [137] oraz TPC-H [138] zostały wprowadzone, aby przetestować algorytmy indeksowania w warunkach zbliżonych do rzeczywistych. W [168] porównano i scharakteryzowano oba zestawy kwerend. TPC-C jest klasycznym zestawem, działającym na tabelach sklepowych. Znajdziemy tam m.in. tabele klienta, zamówienie i listę towarów na magazynie. Schemat relacji, zaproponowaną liczbę relacji i liczbę rekordów dla każdej tabeli pokazuje rysunek 4.2. Zestaw ten mocno obciąża dysk [169] za pomocą losowych wyszukiwań po kluczu oraz wyszukiwań skanujących całą tabelę. TPC-H jest podobny do TPC-C, lecz skupia się głównie na wielowątkowych dostępach do bazy oraz na sprawdzaniu wydajności algorytmów łączących rekordy z kilku tabel (operacja join) [170]. Ponieważ algorytmy i struktury danych przedstawione w tej pracy skupiają się na indeksowaniu tabeli, zestaw TPC-C został wybrany jako główny schemat testowania indeksów. Dodatkowo w eksperymentach zrezygnowano z testowania relacji pomiędzy tabelami. Dostosowano wszystkie kwerendy tak, aby pracowały tylko na jednej tabeli. Oprócz tego zredukowano zapytania SQL do prostych operacji na strukturze danych takich jak: dodawanie nowego rekordu, usuwanie starego rekordu, wyszukiwanie kolumn po kluczu, skanowanie tabeli w celu

Model	Pojemność strony	Szybkość odczytu	Szybkość zapisu
Testowy Model PCM	64B	6,4GB/s	1,6GB/s

Tabela 4.5: Parametry testowego modelu pamięci PCM [167]

znalezienia wartości kolumny, która kluczem nie jest oraz aktualizacja wartości rekordu jak i klucza rekordu. Taka redukcja nie wpływa znacząco na rzetelność testów, ponieważ celem eksperymentów jest zmierzenie efektywności obsługi podstawowych operacji podanych wcześniej na różnych tabelach oraz modelach pamięci a nie efektywności silnika bazy danych w połączeniu z indeksem. Zatem operacje łączenia tabel w zapytaniu poprzez polecenia join, where itd. można pominąć.



Rysunek 4.2: Schemat bazy danych TPC-C

Tabela 4.6 przedstawia schemat bazy danych TPC-C, zawierający liczbę kolumn składających się na rekord oraz rozmiar rekordu, czyli sumę rozmiarów wszystkich kolumn. Rozmiar rekordu jest ważny podczas testowania struktur danych opartych o metodę zapisania całymi wierszami. Analogicznie liczba kolumn jest głównym czynnikiem wpływającym na szybkość indeksów kolumnowych. Ze względu na oba te czynniki do eksperymentów zostały wybrane trzy tabele o odmiennej charakterystyce. Tabela *Sklep* (4.8) uznawana jest za podstawową tabelę do testowania. Liczba kolumn 9 jest medianą pod względem liczby kolumn, a rozmiar rekordu 113B jest bliski i średniej (177B) i medianie (90B) rozmiaru wszystkich kolumn. Możemy zatem uznać tabelę *Sklep* jako tabelę reprezentacyjną bazy TPC-C. Kolejne wybrane tabele to *Nowe zamówienie* (4.9), która jest najmniejszą tabelą (20B) w bazie danych oraz tabela *Klient* (4.7), która jest największą tabelą w bazie TPC-C (719B).

4.2.1 Podstawowy zestaw kwerend

Operacje na indeksach dzielimy na dwie kategorie: punktowe oraz zakresowe. Operacje punktowe pracują tylko na pojedynczych danych. Do tych operacji należą: wstawianie pojedynczego rekordu, usuwanie jednego rekordu oraz wyszukiwanie punktowe (ang. point search). Na całym zakresie danych operują komendy wstawiania wielu rekordów (ang. bulkload) oraz wyszukiwania z zakresu kluczy (ang. range search). Indeks powinien wspierać wszystkie te operacje niezależnie od sposobu przechowywania danych (wierszowo lub kolumnowo). Pierwszy zestaw operacji - operacje punktowe są zazwyczaj używane w aplikacjach dostępnych dla szerokiego grona użytkowników. Przykładem jest strona internetowa sklepu. Za każdym razem gdy chcemy kupić jakiś produkt, system sklepu musi

sprawdzić dostępność jednego wybranego produktu. Taka operacja zrealizowana będzie za pomocą kwerendy wyszukiwania punktowego. Pozostając przy sklepie internetowym, właściciel będzie dodawał i usuwał pojedyncze produkty wykorzystując proste API swojego sklepu i bazy danych. Takich przykładów można podać wiele. Widzimy zatem, że operacje punktowe muszą znaleźć się w zestawie testowych kwerend, ponieważ ich czas obsługi wpłynie na opinie użytkowników wielu aplikacji. Tworząc zestaw testowy trzeba również wziąć pod uwagę sposób użycia bazy danych. Jedni użytkownicy będą częściej wpisywać i usuwać dane, inni odczytywać dane i na nich pracować a jeszcze inni zbalansują liczbę zapisów i odczytów. Z tego powodu powstały trzy różne podstawowe zestawy kwerend: ZP_{zapis}, ZP_{odczyt} i ZP_{balans}. W każdym z zestawów operacje dzielone są na X serii. Liczba serii ustalana jest przez użytkownika. Domyślnie jest ich 10. W każdej serii najpierw wykonywane są operacje dodawania rekordów, po nich następują wyszukiwania po kluczu, a na końcu operacje usuwania rekordów. Zapis do bazy został podzielony na wstawiania i usuwania w relacji 3:1.

1. ZP_{zapis} - 60% operacji wstawiania pojedynczego rekordu, 20% operacji wyszukiwania pojedynczego rekordu, z użyciem klucza, 20% operacji usuwania pojedynczego rekordu

2. ZP_{odczyt} - 15% operacji wstawiania pojedynczego rekordu, 80% operacji wyszukiwania pojedynczego rekordu, z użyciem klucza, 5% operacji usuwania pojedynczego rekordu

3. ZP_{balans} - 37.5% operacji wstawiania pojedynczego rekordu, 50% operacji wyszukiwania pojedynczego rekordu, z użyciem klucza, 12.5% operacji usuwania pojedynczego rekordu

Podstawowy zestaw kwerend sprawdza się dobrze na gotowych tabelach jak i tabelach pustych, nie zawierających żadnych rekordów. Operacje wyszukiwania pojedynczego rekordu jak i wstawiania jednego rekordu mają dla większości indeksów tę samą złożoność obliczeniową. Ich czas będzie różnił się głównie w zależności od charakterystyki pamięci, na której testowana jest baza danych. Przykładowo, na pamięci flash dodawanie rekordu będzie znacznie wolniejsze niż wyszukiwanie, ponieważ zapis wymaga kosztowej operacji usuwania całego bloku.

Tabela	Liczba kolumn	Rozmiar rekordu
Sklep	9	113B
Magazyn	17	322B
Przedmiot	5	90B
Dzielnica	11	125B
Klient	21	719B
Historia	8	76B
Nowe zamówienie	3	20B
Zamówienie	8	47B
Linia zamówień	10	86B

Tabela 4.6: Baza danych TPC-C

Nazwa kolumny	Rozmiar kolumny
C_ID	4B
C_D_ID	4B
C_W_ID	8B
C_FIRST	16B
C_MIDDLE	2B
C_LAST	16B
C_STREET_1	20B
C_STREET_2	20B
C_CITY	20B
C_STATE	2B
C_ZIP	9B
C_PHONE	16B
C_SINCE	16B
C_CREDIT	2B
C_CREDIT_LIM	16B
C_DISCOUNT	8B
C_BALANCE	16B
C_YTD_PAYMENT	16B
C_PAYMENT_CNT	4B
C_DELIVERY_CNT	4B
C_DATA	500B

Tabela 4.7: Klient TPC-C

Nazwa kolumny	Rozmiar kolumny
W_ID	8B
W_NAME	10B
W_STREET_1	20B
W_STREET_2	20B
W_CITY	20B
W_STATE	2B
W_ZIP	9B
W_TAX	8B
W_YTD	16B

Tabela 4.8: Sklep TPC-C

Nazwa kolumny	Rozmiar kolumny
NO_O_ID	8B
NO_D_ID	4B
NO_W_ID	8B

Tabela 4.9: Nowe Zamówienie TPC-C

4.2.2 Rozszerzony zestaw kwerend

Zestaw rozszerzony powstał na podstawie kwerend z TPC-C. Celem tego zestawu jest symulacja hurtowni danych, w której operacje są buforowane i wykonywane na wielu rekordach. W przeciwieństwie do zestawu podstawowego, tutaj używać będziemy wstawiania wielu danych jednocześnie (ang. bulkload), jeśli taka operacja jest wspierana przez indeks, a jeśli nie, to wstawianie będzie realizowane klasycznie, jeden po drugim. Wyszukiwanie całego zakresu kluczy jest sparametryzowane przez selektywność zapytania. Najczęściej używa się selektywności na poziomie od 1% do 5%. Domyślnie ten parametr ustawiony jest na 1%. Tak samo jak kwerendy podstawowe, rozszerzony zestaw podzielony jest na serie. W każdej serii najpierw wykonywane jest dodawanie rekordów, później wyszukiwanie, a na końcu usuwanie rekordów. Biorąc pod uwagę charakterystykę wielu kwerend TPC-C oraz ich redukcję do podstawowych operacji na indeksach tylko na jednej tabeli, przygotowano aż 4 różne zestawy kwerend.

1. ZR_A - 100 serii, każda seria zawiera operacje dodawania 5 rekordów, 10 wyszukiwań o podanej selektywności (domyślnie 1%) oraz usuwanie 5 rekordów,
2. ZR_B - 5 serii, każda seria zawiera operacje dodawania 100000 rekordów, 5 wyszukiwań o podanej selektywności (domyślnie 1%) oraz usuwanie 100000 rekordów,
3. ZR_C - 10 serii, każda seria zawiera operacje dodawania 10000000 rekordów, 20 wyszukiwań o podanej selektywności (domyślnie 1%) oraz usuwanie 1000000 rekordów,
4. ZR_D - 10 serii, każda seria zawiera operacje dodawania 1000000 rekordów, 10 wy-

szukiwań o podanej selektywności (domyślnie 1%) oraz usuwanie 10000 rekordów.

Rozszerzony zestaw kwerend sprawdza się o wiele lepiej na tabelach zawierających jakieś rekordy. Operacje wyszukiwania zależne są od selektywności. Zatem czas potrzebny na odczytanie danych jest zależny proporcjonalnie od wielkości bazy. W przeciwieństwie do podstawowego zestawu, zestaw rozszerzony osiągnie o wiele lepszy czas na pustej bazie, ponieważ wyszukiwania będą mniej kosztowne. Zalecana wielkość bazy danych to 10 milionów rekordów.

4.3 Testowanie indeksów częściowych

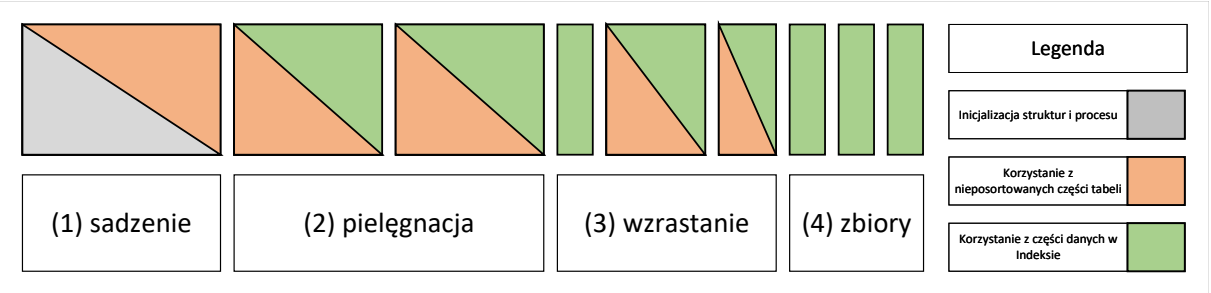
Ciężko jest porównać ze sobą algorytmy tworzące indeksy częściowe takie jak Indeks Cracking [56] czy też Adaptive Merging [13]. Sumaryczny czas może nie wystarczyć, aby w pełni dostrzec różnice i charakterystykę algorytmów. Oprócz czasu należy patrzeć na ilość pamięci potrzebnej na metadane, amortyzację kosztów tworzenia indeksu oraz czas potrzebny na ukończenie procesu. W [139] pokazano całkiem nowy sposób na analizę procesu tworzenia częściowego indeksu. Polega na obserwacji każdego zapytania przez pewien okres, dopóki indeks nie zostanie w pełni stworzony. Zatem te eksperymenty polegają na stworzeniu całego indeksu od podstaw za pomocą tylko kwerend wyszukujących, które dodają odczytane rekordy do indeksu. Na podstawie takiej obserwacji można dokładniej ocenić algorytm nie tylko biorąc pod uwagę czas tworzenia indeksu, ale także parametry ważne dla użytkownika końcowego np. maksymalny i średni czas obsługi zapytania w każdej fazie tworzenia indeksu. Autorzy wyznaczyli 4 takie fazy, które można wyróżnić w każdym algorytmie indeksowania.

1. **sadzenie (ang. planting)** - Pierwsza faza sadzenia polega na inicjalizacji całego procesu. Początkowo trzeba skopiować tabele wykonując przy tym pewne operacje potrzebne do rozpoczęcia procesu. Pierwsze kroki tego typu algorytmów zazwyczaj potrzebują więcej czasu niż prosty skan całej tabeli. Indeks zawiera na tyle mało elementów, że prawdopodobieństwo jego użycia jest niewielki. Dlatego musimy wykonać drogie skanowanie oraz dodać kilka operacji potrzebnych na powiększenie indeksu. Podsumowując, pierwsza faza trwa dopóki czas obsługi kwerendy jest większy niż zwykły, prosty skan tabeli.

2. **pielęgnacja (ang. nursing)** - Druga faza zaczyna się, gdy czas jest mniejszy niż prosty skan, ale nadal większy niż w przypadku użycia pełnego indeksowania. W tej fazie zaczynamy korzystać z posortowanych partii danych, jednocześnie obniżamy koszty dodatkowych operacji, ponieważ dodajemy coraz mniej danych do indeksu.

3. **wzrastanie (ang. growing)** - Trzecia faza zaczyna się, gdy natrafimy na zapytanie, które możemy obsłużyć w minimalnym czasie, czyli takim samym co pełny indeks. W tym okresie indeks jest na tyle duży, że prawdopodobieństwo jego przydatności jest wysokie. W prawie każdej kwerendzie częściowo z niego korzystamy, jednak prócz pojedynczych przypadków nie jesteśmy w stanie dorównać wyszukiwaniu w posortowanym zbiorze danych.

4. **zbiory (ang. harvesting)** - Ostatnia faza zbiorów zaczyna się, gdy indeks jest w pełni stworzony, a zatem nie mamy już czasowych narzutów na jego tworzenie. Czas na obsługę zapytania jest taki sam jak przy pełnym indeksie.



Rysunek 4.3: Fazy częściowego indeksowania

W [171] ustalono, że powyższa analiza powinna być przeprowadzona na różnym wzorcu zapytań. Zazwyczaj używa się wzorca losowego. Minimalny klucz zapytania losowany jest z rozkładu jednostajnego. A maksymalny klucz zapytania zależy od selektywności. Jest on ustalany tak, aby podczas kwerendy odpowiednia liczba danych została wczytana i zwrócona przez silnik baz danych. Zwykle selektywność ustawia się na 1%-5%. W tym wzorcu prawdopodobieństwo przydatności indeksu zależne jest od liczby danych już posortowanych. Wzorzec drugi - sekwencyjny jest podobny do ciągu przyczynowo skutkowego. Ma na celu symulację sytuacji, w której poprzedni wynik zapytania wpływa na kolejny, np. raporty miesięczne i korekty tych raportów. Implementując wzorzec sekwencyjny wybieramy minimalny klucz z tabeli wybranej do indeksowania. Następnie minimalny klucz żądany przez następne kwerendy jest przesuwany o połowę liczby danych wczytanych w poprzedniej kwerendzie. Liczba danych jest zależna od selektywności tak samo jak w przypadku wzorca losowego. Zatem w pierwszym zapytaniu wczytamy tylko dane z części nieposortowanej. W każdej kolejnej połowę danych z nieposortowanego sektora tabeli i połowę z indeksu. Oczywiście gdy indeks będzie gotowy, będziemy czytali już tylko z w pełni stworzonego indeksu. Trzecim wzorem jest wzorzec nowych kluczy. Każda kwerenda prosi o dane spoza indeksu. Zatem zaczniemy go wykorzystywać dopiero gdy zostanie w pełni stworzony. Teoretycznie jest to najgorszy przypadek dla algorytmów częściowego indeksowania, ponieważ nie korzystamy z indeksu podczas jego tworzenia. Dobrą praktyką jest przeprowadzenie testów dla wszystkich podanych wyżej wzorców, ponieważ różne zestawy kwerend testują różne części algorytmu. Przyjęło się jednak, że domyślnie używa się wzorca losowego, który najlepiej symuluje użycie bazy danych w systemie rozproszonym z wieloma użytkownikami, gdzie kwerendy od różnych użytkowników trafiające do systemu w tym samym czasie wyglądają na losowe.

Algorytmy indeksowania na pamięci flash

W tym rozdziale omówimy sposoby indeksowania danych, popularne algorytmy i struktury danych służące do indeksowania na pamięci flash oraz przedstawimy nowy algorytm zaproponowany w [1].

Pamięć flash została dokładnie omówiona w rozdziale **3**. Przypomnijmy sobie jednak główne cechy tej pamięci. Flash dzieli się na dwa rodzaje NOR i NAND. NOR charakteryzuje się bardzo szybkim odczytem, ze względu na połączenie równoległe, ale wolnym zapisem. Jego pojemność jest bardzo ograniczona. Zwykle pojemność kości NOR nie przekracza kilku MB. Jest głównie wykorzystywany do zapisywania kodu oprogramowania na urządzeniach wbudowanych. Do przechowywania danych używa się pamięci NAND. Charakteryzuje się szybszym zapisem i wolniejszym odczytem, ale co ważniejsze posiada znacznie większą pojemność sięgającą nawet kilkuset GB.

W tym rozdziale skupimy się wyłącznie na pamięci NAND. Pamiętajmy, że pamięć flash to pamięć blokowa. Podzielona jest na strony o wielkości 2KB-8KB i na bloki o wielkości 64KB - 512KB. Strona jest minimalną jednostką odczytu i zapisu, oznacza to, że nawet gdy chcemy wykonać operację WE/WY na 100B to musimy wykonać ją na całej stronie. Dodatkowo flash posiada pewne ograniczenia zmiany wartości bitu. Ze względu na architekturę pamięci nie można zmienić wartości bitu z „0” na „1”. Aby nadpisać stronę, musimy najpierw wykonać kosztowną operację kasowania danych (ang. erase) zmieniającą wartość całego bloku na ciąg samych „1”. Dopiero po operacji kasowania, możemy wpisać żądaną daną poprzez zmianę „1” na „0”. Proces ten powoduje ogromną asymetrię pomiędzy szybkością zapisu i odczytu danych [79]. Należy także pamiętać, że każda komórka pamięci posiada skończony cykl życia zapisu. Każda operacja zapisu jak i kasowania zmniejsza żywotność pamięci. Zatem algorytm pracujący na tej pamięci, powinien nie tylko minimalizować liczbę zapisów w celu skrócenia czasu wykonywania się operacji, ale także powinien minimalizować liczbę operacji kasowania aby maksymalnie wydłużyć okres życia pamięci flash.

5.1 Wykorzystywanie pamięci flash w urządzeniach wbudowanych

Obecnie komputery osobiste, serwery jak i hurtownie danych nie używają kości flash. W zamian za to przechowują dane na dyskach SDD, które są zbudowane z kości flash NAND. Jednak oprócz samych kości zawierają wiele dodatkowych elementów oraz specjalne oprogramowanie, które przyspiesza działanie samego dysku. W urządzeniach wbudowanych, które zazwyczaj są o wiele mniejsze od komputerów osobistych, nie ma miejsca

na pełne dyski SSD. Z tego względu w urządzeniach typu IoT (ang. Internet of Things) takich jak telefony komórkowe, inteligentne zegarki, pralki czy chociażby systemy pomiarowe do rowerów, używa się pojedynczych kości flash typu NAND do przechowywania danych oraz kości typu NOR do przechowywania oprogramowania.

Urządzenia wbudowane posiadają ograniczone moce obliczeniowe, które są dedykowane prawie w całości do wykonywania głównej funkcji urządzenia. Oznacza to, że takie urządzenia nie posiadają już wolnych zasobów obliczeniowych, aby w tle działała jakaś relacyjna baza danych. Zamiast kilku tabel i relacji pomiędzy nimi, dane przechowywane są w prostych strukturach danych jak drzewo B+ [19] lub drzewo LSM [172]. Drzewo LSM jest rzadko wybierane z powodu sporego zużycia pamięci RAM do przechowywania mapowania struktury wewnętrznej $SSTableMap$. Operacje na tych danych nie są wykonywane za pomocą języka SQL jak to jest w przypadku relacyjnych bazach danych posiadających silnik bazy i interpreter poleceń. Struktura danych wystawia użytkownikowi proste API, zazwyczaj składające się z prostych poleceń jak dodanie nowego elementu, wyszukanie elementu po kluczu jak i usunięcie elementu. Wykorzystując to proste API, aplikacja działająca na urządzeniu może przechowywać wszystkie potrzebne dane. Gdy jednak taka architektura nie wystarcza do zbudowania aplikacji, częstym rozwiązaniem jest budowa zewnętrznego serwera, bardzo wydajnego komputera z szybkimi dyskami SSD. Serwer ten wykorzystując proste API każdego z urządzeń odbiera dane i wstawia je do relacyjnej bazy danych. Dzięki temu użytkownik końcowy może pracować na zgromadzonych danych za pomocą języka SQL, ponieważ kwerenda zostanie odczytana na serwerze, zinterpretowana i wykonana na zgromadzonych wcześniej danych zapisanych jako relacyjna baza. Sama struktura danych, jak np. drzewo B+, nie jest wystarczająca do efektywnego przechowywania danych na pamięci flash. Pamiętamy z rozdziału **3**, że bez wirtualizacji pamięci kość straciłaby żywotność w ciągu kilku sekund. Z tego względu niezależnie czy urządzenie posiada jakiś system operacyjny np. Linux czy też nie, implementowany jest system wirtualizacji bloków [89] lub stron pamięci flash [90] w celu wydłużenia żywotności kości NAND. Do najpopularniejszy systemów wirtualizacji pamięci należą algorytm FTL [88], BFTL [173] i UBI [174].

5.2 Flash Aware Tree

W [1] zaproponowaliśmy nowatorską strukturę danych do przechowywania rekordów na pamięci flash typu NAND o nazwie FA-Tree. Głównymi cechami nowej struktury jest praca w trybie relacyjnych baz danych jako indeks zgrupowany, czyli działanie jako struktura danych przechowująca dowolne rekordy i wystawiająca proste API. Dodatkowo została zastosowana własna wirtualizacja pamięci jako efekt uboczny działania struktury, co pozwala na uruchomienie indeksu bez systemu wirtualizacji takiego jak na przykład system FTL. FA-Tree posiada szybkie wyszukiwanie, wynikające z drzewiastego ułożenia danych. Wyszukiwanie to wymaga $O(\log n)$ odczytów z pamięci flash. Dodatkowo w przeciwieństwie do popularnego drzewa B+, zaproponowany indeks minimalizuje liczbę zapisów dzięki leniwej reorganizacji danych. Eksperymenty z użyciem platformy testowej SIPS opisanej dokładnie w rozdziale **4** wykazały 30 krotną przewagę nowej struktury FA-Tree nad zwykłym drzewem B+ oraz około 35% przewagę nad drzewem LSM.

5.2.1 Struktura FA-Tree

Nasza nowa struktura FA-Tree składa się z kilku poziomów. Na samej górze, na poziomie 0 (L_0) mamy bardzo małe drzewo B+ o wielkości pojedynczego bloku kości flash. Ponieważ ten poziom jest bardzo mały, z łatwością zmieści się do pamięci RAM nawet na urządzeniu wbudowanym. Służy on do buforowania danych zanim trafią do dalszych poziomów (L_i, $i > 0$) zapisanych na kości flash. Aby uzyskać strukturę drzewa, liście w drzewie B+ zawierają wskaźniki do L_1, czyli pierwszego poziomu zapisanego na pamięci nieulotnej, dodatkowo każdy kolejny poziom zawiera zbiór wskaźników do odpowiednich stron pamięci poziomu niżej. Dzięki takiej strukturze możemy w łatwy sposób przejść po naszym drzewie od góry do dołu, aby wyszukać rekord o wskazanym kluczu. Każdy z poziomów posiada swoją maksymalną pojemność wyrażoną w liczbie bloków kości flash. Dzięki temu nasza struktura nie marnuje czasu na usuwanie częściowo zapisanego bloku. Każda operacja alokacji pamięci odbywa się poprzez alokację pełnego bloku. W tym momencie wykonujemy również mapowanie podobne do algorytmu FTL. Mapowanie jest niezależne od systemów takich jak FTL czy UBI. Oznacza to, że nasza struktura potrafi współpracować z tymi systemami jako kolejna warstwa oprogramowania oraz potrafi sama zapobiegać przedwczesnemu zniszczeniu bloku kości NAND. Aby zminimalizować koszty reorganizacji struktury, podobnie jak drzewa B+ czy LSM wprowadziliśmy współczynnik K. Każdy kolejny poziom, zawiera K razy więcej bloków ($|L_{i+1}| = |L_i| \cdot K$). Podczas eksperymentów zauważyliśmy, że współczynnik K powinno dobrać się odpowiednio do rodzaju kwerendy. Im większe K tym drzewo jest niższe, czyli koszt wyszukiwania jest mniejszy, ale koszt reorganizacji jest o wiele większy. Z tego względu albo powinno dobierać się ten współczynnik pod odpowiedni zestaw kwerend, albo powinno ustalać się neutralną wartość. Z przeprowadzonych eksperymentów wynika, że neutralne wartości K są z przedziału $< 4; 10 >$.

W naszej nowej strukturze wyróżniamy 4 rodzaje danych:

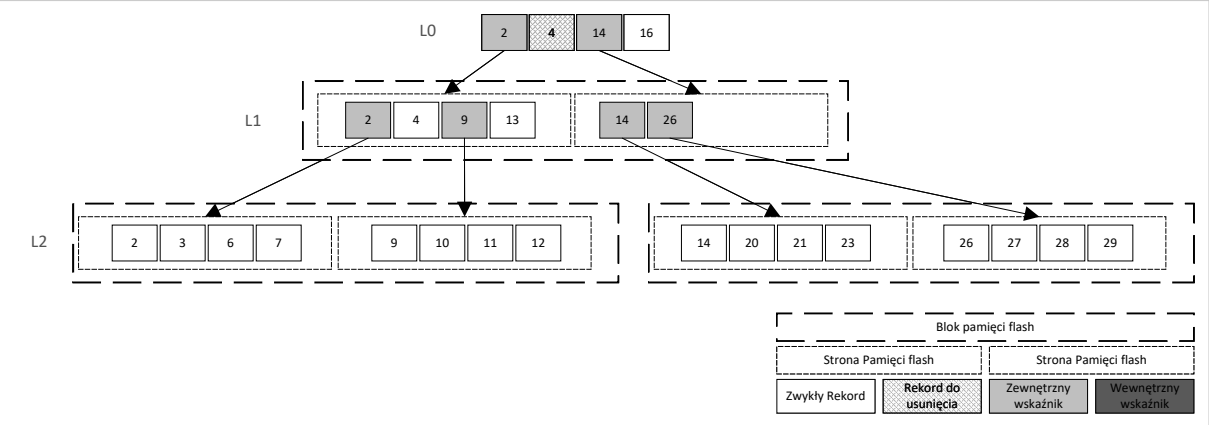
1. Normalna dana (ang. normal entry) - jest to zwykły rekord. Zawiera klucz, po którym rekord jest wyszukiwany i wstawiany oraz dowolne dane, które użytkownik chciał zapisać.

2. Usunięta dana (ang. deleted entry) - aby zminimalizować liczbę zapisów i operacji usuwania na pamięci flash, wprowadziliśmy leniwy mechanizm usuwania. Zamiast usuwać rekord natychmiastowo, co doprowadziłoby do reorganizacji drzewa, wstawiamy nowy rekord oznaczony jako d_{key} o takim samym kluczu jak rekord, który chcemy usunąć. Gdy rekordy podczas nieuniknionej migracji poziomów w dół spotkają się, wówczas są usuwane ze struktury. Operacja ta dokładnie opisana jest w kolejnych sekcjach tego rozdziału.

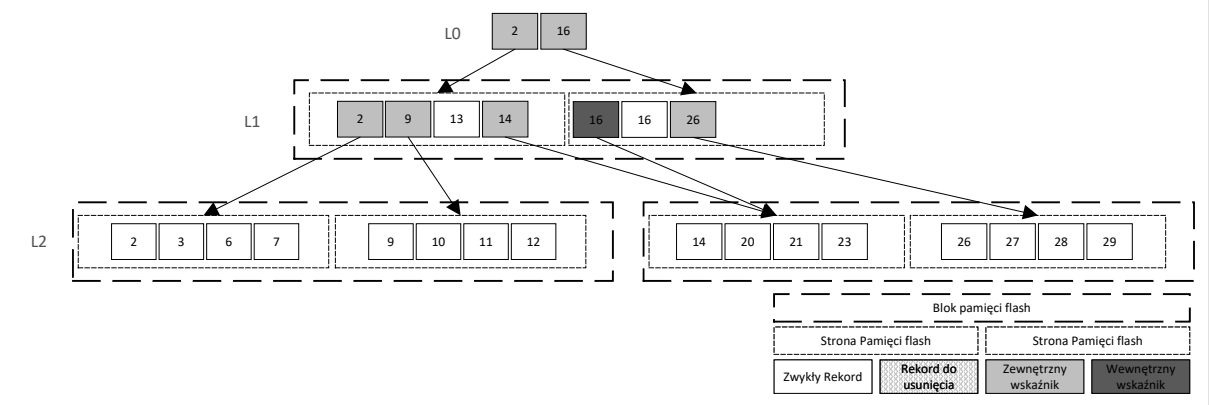
3. Zewnętrzny wskaźnik (ang. external fence) - ten element struktury zawiera dwie wartości. Identyfikator strony na którą wskazuje PID z poziomu niżej oraz wartość pierwszego klucza (czyli najmniejszego) strony na którą wskazuje. Podczas operacji scalania poziomów, dla każdej strony poziomu L_{i+1} tworzony jest nowy wskaźnik do tej strony i umieszczany jest w poziomie nad nim czyli w L_i. Dzięki temu struktura drzewa jest zachowana po każdej reorganizacji.

4. Wewnętrzny wskaźnik (ang. internal fence) - tak samo jak zewnętrzny wskaźnik zawiera 2 wartości: identyfikator strony na którą wskazuje PID oraz klucz. Tym razem nie jest to klucz strony, na którą wskazuje, ale klucz najmniejszego klucza

na poziomie, na którym znajduje się wewnętrzny wskaźnik. Pierwszym elementem każdej strony musi posiadać wskaźnik po reorganizacji poziomów. Jeśli strona nie zawiera zewnętrznego wskaźnika, wówczas tworzony jest wewnętrzny wskaźnik, który wskazuje na tę samą stronę co poprzedni wskaźnik danego poziomu.



Rysunek 5.1: Przykładowa struktura FA-Tree



Rysunek 5.2: FA-Tree, zaraz po scalaniu poziomów L_0 i L_1

Rysunek 5.1 obrazuje przykładowy układ struktury FA-Tree. Na samej górze mamy małe drzewo B+, które posiada maksymalnie rozmiar jednego bloku. W tym przykładzie jeden blok składa się z dwóch stron, a każda ze stron może pomieścić maksymalnie 4 rekordy. Poziom L_0 jest zapisany w RAM, posiada dwa wskaźniki do poziomu niżej, rekord do usunięcia z kluczem 4 oraz zwykły rekord z kluczem o wartości 16. Ponieważ na poziomach L_0 oraz L_1 zawsze pierwszym elementem strony jest zewnętrzny wskaźnik, struktura nie posiada ani jednego wewnętrznego wskaźnika. Jak widać, taki układ struktury pozwala nam na bardzo szybki odczyt rekordu o podanym kluczu. Załóżmy, że chcemy znaleźć rekord o kluczu 20. Zaczynając od poziomu L_0, zawsze szukamy wskaźnika (obojętnie czy wewnętrznego czy zewnętrznego) o wartości najbardziej zbliżonej do wartości naszego szukanego klucza, ale jednak mniejszej lub równej niż nasz klucz. Skoro szukamy wartości 20, to takim wskaźnikiem na poziomie L_0 będzie 14. Odczytujemy w takim razie drugą stronę poziomu L_1, dostajemy 2 wskaźniki 14 i 26. Tym razem również wybieramy 14 i trafiamy do strony zawierającej dane 14, 20, 21, 23. Teraz wystarczy już tylko odczytać rekord z kluczem 20 i zwrócić go użytkownikowi jako wynik zapytania.

Rysunek 5.2 przedstawia układ struktury zaraz po scaleniu ze sobą poziomów L_0 i L_1

z przykładu omawianego na rysunku 5.1. Ponieważ wcześniej na poziomie L_0 znajdował się rekord do usunięcia o wartości 4, który czekał na usunięcie prawdziwego rekordu na poziomie L_1, podczas scalania tych obydwu poziomów pominęliśmy oba rekordy i nie zapisaliśmy ich do nowych poziomów. Ponieważ poziom L_2 nie zmieniał się, wszystkie wskaźniki zewnętrzne ze starego poziomu L_1 pozostały bez zmian. Warto zwrócić uwagę, że rekord o kluczu 16 został przeniesiony z poziomu L_0 do L_1. Gdybyśmy go po prostu przepisali, stałby się pierwszym elementem drugiej strony nowego poziomu L_1. Ponieważ uniemożliwiłoby to późniejsze wyszukiwanie w drzewie, musieliśmy dodać wewnętrzny wskaźnik wskazujący na tę samą stronę do ostatni wskaźnik na poziomie L_1 czyli wskaźnik 14. Wartość wskaźnika wewnętrznego jest taka sama jak pierwszego normalnego rekordu na stronie czyli 16.

5.2.2 Procedury indeksu FA-Tree

Wstawianie

Aby zapisać nowy rekord do drzewa FA-Tree, najpierw rekord wstawiany jest do niewielkiego drzewa B+ w pamięci RAM. Dzięki temu nie wykonujemy kosztownych operacji zapisu na pamięci flash za każdym razem. Zamiast tego buforujemy wszystkie dane w szybkiej pamięci RAM. Gdy drzewo B+ jest w stanie pomieścić kolejnego rekordu, zrzucany jest do pamięci flash. Wykonujemy wtedy reorganizację poziomów L_0 oraz L_1. Ponieważ każdy z poziomów zawiera ograniczoną pojemność, to istnieje prawdopodobieństwo, że reorganizacja poziomów L_0 z L_1 przepełni poziom L_1. Wtedy rekurencyjnie scalamy poziomy L_1 z L_2 oraz wszystkie kolejne poziomy, gdy zajdzie tak potrzeba, dopóki poziom przyjmujący dane będzie miał wolne miejsce.

Wyszukiwanie

Struktura FA-Tree wspomaga wyszukiwanie punktowe podanego pojedynczego klucza oraz wyszukiwanie rekordów z podanego zakresu kluczy. Podczas wyszukiwania punktowego przechodzimy drzewo z góry na dół, dopóki nie napotkamy na interesujący nas rekord. Ponieważ każda strona z danego poziomu zawiera przynajmniej jeden wskaźnik na stronę poniżej, to możemy przejść przez drzewo wczytując tylko jedną stronę z każdego poziomu. Za każdym razem idziemy do strony o największym kluczu ze zbioru kluczy mniejszych niż nasz szukany. Musimy pamiętać, aby omijać rekordy zaznaczone jako *Usunięta dana*. Ponieważ nowy rekord do usunięcia wstawiony jest zawsze po oryginalnym rekordzie, będzie on na poziomie wyższym niż oryginalny rekord. Zatem gdy napotkamy na *Usunięta dana* o szukanym przez nas kluczu, możemy zakończyć wyszukiwanie i zgłosić użytkownikowi, że rekord o podanym kluczu został usunięty z drzewa. Bardzo podobnie przeprowadzamy wyszukiwanie z zakresu kluczy. Naszym kluczem wiodącym będzie klucz minimalny z zakresu. To na podstawie jego wartości będziemy przechodzić przez drzewo, dokładnie tak jak przy wyszukiwaniu punktowym. Dodatkowo na każdym poziomie będziemy wczytywać rekordy na prawo od odwiedzonego rekordu do momentu, aż nie napotkamy na rekord z kluczem większym niż maksymalny klucz szukanego zakresu. Z danymi do usunięcia postępujemy dokładnie tak samo jak przy wyszukiwaniu punktowym.

Usuwanie

Usuwanie rekordu o podanym kluczu nie następujące natychmiastowo jak to ma miejsce w drzewach B+. Zamiast tego wstawiamy nowy rekord z kluczem, który mamy usunąć i ustawiamy typ tego rekordu jako *Usunięta dana*. Gdy podczas reorganizacji poziomów natrafimy na oba rekordy: oryginalny oraz ten do usunięcia, wtedy usuwamy oba rekordy. Ta operacja realizowana jest jako brak zapisu poszczególnych rekordów ze starych zbiorów do nowego zbioru.

Reorganizacja drzewa

Reorganizacja drzewa przeprowadzana jest w przypadku, gdy któryś z poziomów nie ma wolnego miejsca na kolejne elementy. Wtedy uruchamiamy proces scalania dwóch poziomów: obecnego i kolejnego pod nim. Jeśli nie istnieje poziom niżej, wtedy jest on tworzony z odpowiednio dużą pojemnością (K razy większą od poprzedniego poziomu). Ponieważ każdy poziom jest posortowany, to scalanie poziomów L_i z L_{i+1} odbywa się tak samo jak scalanie posortowanych list. Tworzymy nowy poziom L'_{i+1}, zapisujemy tam za każdym razem minimum z dwóch scalanych poziomów i powtarzamy proces, dopóki oba poziomy nie zostaną scalone w jeden. Następnie usuwamy stare poziomy L_i i L_{i+1}, tworzymy kolejny poziom wyżej L'_i i wstawiamy do niego wskaźniki na strony z poziomu niższego L'_{i+1}. Podczas reorganizacji pamiętamy, że jeśli poziom L_i będzie zawierał rekord o tym samym kluczu co rekord na poziomie L_{i+1}, ale z odpowiednim znacznikiem do usunięcia, wtedy pomijamy oba rekordy i żadnego z nich nie wpisujemy do nowego poziomu L'_{i+1}. Możliwe, że podczas tego procesu poziom L'_{i+1} również nie będzie miał wystarczająco miejsca aby pomieścić wszystkie dane. Wtedy wykonujemy scalanie poziomów L'_{i+1} z L_{i+2}. Proces ten powtarzamy rekurencyjnie do momentu uzyskania poziomu mieszczącego wszystkie rekordy.

5.2.3 Opis algorytmów

Algorytm scalania poziomów

Algorytm scalania poziomów 5.1 jest najważniejszym algorytmem całej struktury. To wprowadzenie leniwego usuwania jak i buforowania danych pozwoliło na minimalizację liczby reorganizacji struktury oraz zapisów na kości flash. W momencie gdy dany poziom jest pełny, wykonuje scalanie z poziomem niżej. Linijki 1-4 są odpowiedzialne za przygotowanie struktury do algorytmu. Pamiętajmy, że każdy poziom składa się z określonej liczby bloków, a nie stron tak jak ma to miejsce w przypadku drzew B+ czy LSM. Dzięki temu podczas tworzenia poziomów możemy użyć wbudowanego systemy wirtualizacji blokowej pamięci flash. Dodatkowo podczas usuwania starych poziomów (linie 36 - 37) w pełni wykorzystujemy operację kasowania, ponieważ nie usuwamy zbędnych stron, które nie należały do naszego poziomu, a które trzeba byłoby skopiować w inne miejsce. Cały proces jest bardzo podobny do scalania posortowanych list. Dlatego algorytm działa dopóki oba poziomy nie będą puste (linie 8-35). Ponieważ niższy poziom L_{i+1} będzie usuwany i nadpisany nowym poziomem L'_{i+1}, to wszystkie wskaźniki w poziomie L_i staną się nieaktualne. Dlatego w linii 9 pomijamy wszystkie wskaźniki, zarówno zewnętrzne jak i wewnętrzne. Poziom L_{i+2} zostanie bez zmian (chyba że rekurencyjnie wywołamy algorytm z linii 35), a więc nie możemy usunąć wskaźników zewnętrznych z poziomu L_{i+1}. Gdybyśmy to zrobili, stracilibyśmy bezpowrotnie dostęp do poziomu L_{i+2}. Z kolei wskaź-

niki wewnętrzne na poziomie L_{i+1} nie są nam potrzebne. Zostały stworzone tylko dlatego, aby spełnić warunek struktury mówiący o tym, że pierwszy rekord na stronie musi być wskaźnikiem. Ponieważ zmieni się układ poziomu L_{i+1}, który zostanie przepisany do L'_{i+1}, to również zmienią się wskaźniki wewnętrzne. Tak jak wspominaliśmy, usuwanie elementów nie odbywa się natychmiastowo. Zamiast tego wstawiany jest rekord o tym samym kluczu odpowiednio oznaczony jako rekord do usunięcia. Gdy podczas scalania poziomów (linie 12-13) napotkamy na parę takich rekordów, wówczas usuwamy je pomijając oba rekordy w algorytmie. Oznacza to, że żaden z nich nie zostanie wpisany do nowego poziomu L'_{i+1}. Linie 15-22 są odpowiedzialne za wybór minimalnego klucza z obu poziomów oraz przesunięcie wskaźnika na kolejny rekord. W przypadku poziomu L_{i+1}, musimy także zapamiętać ostatni wskaźnik zewnętrzny. Będzie on potrzebny do stworzenia nowego wskaźnika wewnętrznego na poziomie L'_{i+1} (linie 29-32). Przed wstawieniem nowego rekordu do poziomu L'_{i+1} (linia 33), musimy upewnić się, czy nasza struktura jest prawidłowa. Zatem jeśli zamierzamy wstawić rekord do nowej, pustej strony, musimy najpierw dodać wskaźnik do tej strony do poziomu L'_i aby umożliwić dostanie się do poziomu L'_{i+1}.

5.2.4 Analiza kosztu wstawiania rekordu

W tej sekcji zajmiemy się analizą algorytmu wstawiania nowego rekordu do drzewa FA oraz porównamy je do złożoności obliczeniowej dodawania rekordu do drzewa B+. Nie zajmiemy się wyszukiwaniem, ponieważ samo wyszukiwanie realizowane jest w obu przypadkach w podobny sposób. Wynika to z architektury obu struktur. Każde drzewo wymaga odczytu tylko jednej strony na każdy poziom. Zatem w obu przypadkach koszt wyszukiwania jest logarytmiczny. Tabela 5.1 przedstawia złożoność asymptotyczną wyszukiwania i wstawiania elementów dla struktur drzewa FA oraz drzewa B+. Jak widać wszystkie operacje są logarytmiczne.

Twierdzenie 5.1 *Niech T oznacza amortyzowany czas wstawiania elementu do drzewa FA wynosi, wtedy:*

$$T = \mathrm{O}\left(\frac{K}{P_{cap} - K} \cdot \log n\right),$$

gdzie
K - współczynnik pojemności 2 poziomów ($K = \frac{|L_{i+1}|}{|L_i|}$),
$P_{cap} = \left\lceil \frac{P_{size}}{R_{size}} \right\rceil$ - liczba rekordów możliwych do zapisania na stronie

Dowód. Zanim przejdziemy do pełnej analizy złożoności obliczeniowej wstawiania rekordów do struktur FA-Tree, ustalmy pewne oznaczenia

- n - liczba wstawianych danych
- R_{size} - rozmiar rekordu wyrażony w bajtach
- P_{size} - rozmiar strony wyrażony w bajtach
- L_i^j - liczba danych na poziomie i zaraz po skończeniu scalania j
- W_{cost} - koszt zapisu pojedynczej strony
- R_{cost} - koszt odczytu pojedynczej strony
- FA_{height} - liczba poziomów drzewa FA
- m_i - liczba wywołań algorytmu scalania po n danych dla poziomu i

Pseudokod 5.1: faLevelMerge(L_i, L_{i+1})

```
1  if L_{i+1} nie istnieje then
2      Stwórz pusty poziom L_{i+1} z pojemnością L_i.capacity · K
3  Stwórz pusty poziom L'_i z pojemnością L_i.capacity
4  Stwórz pusty poziom L'_{i+1} z pojemnością L_{i+1}.capacity
5
6  Niech r_i będzie pierwszym rekordem z poziomu L_i
7  Niech r_{i+1} będzie pierwszym rekordem z poziomu L_{i+1}
8  while L_i AND L_{i+1} są niepuste do
       // Poziom L_{i+1} będzie tworzony na nowo, możemy pominąć wszystkie wskaźniki
9      Pomiń wszystkie zewnętrzne i wewnętrzne wskaźniki w L_i
       // Poziom L_{i+2} zostaje, zatem zewnętrzne wskaźniki również muszą pozostać
10     Pomiń wszystkie wewnętrzne wskaźniki w L_{i+1}
11
       // Usuń rekordy do usunięcia
12     while r_i.type = RECORD_TO_DELETE AND r_{i+1}.type =
        RECORD_NORMAL AND r_i.key = r_{i+1}.key do
13         Pomiń oba rekordy r_i oraz r_{i+1}
14
       // Weź minimum z obu poziomów
15     if r_i.key < r_{i+1}.key then
16         recordToInsert := r_i
17         Niech r_i będzie następnym rekordem z poziomu L_i
18     else
19         recordToInsert := r_{i+1}
20         if r_{i+1}.type = RECORD_EXTERNAL_FENCE then
21             lastFence := r_{i+1}
22         Niech r_{i+1} będzie następnym rekordem z poziomu L_{i+1}
23
24     if Obecna strona P poziomu L'_{i+1} jest pusta then
           // Każda nowa strona musi być wskazywana przez jakiś wskaźnik wyżej
25         external.pid := P
26         external.key := recordToInsert.key
27         Wpisz external do L'_i
28
           // Pierwszy rekord na stronie, musi być wskaźnikiem
29         if recordToInsert.type <> RECORD_EXTERNAL_FENCE then
30             internal.pid := lastFence.pid
31             internal.key := recordToInsert.key
32             Wpisz internal do L'_{i+1}
33     Wpisz recordToInsert do poziomu L'_{i+1}
34     if L'_{i+1} nie ma już wolnego miejsca then
35         faLevelMerge(L'_{i+1}, L_{i+2})
36 Zamień stary poziom L_i na nowy poziom L'_i
37 Zamień stary poziom L_{i+1} na nowy poziom L'_{i+1}
```

	Koszt wyszukiwania	**Koszt wstawiania**	
Struktura danych	**Odczyt**	**Odczyt**	**Zapis**
FA-Tree	$\mathrm{O}(\log n)$	$\mathrm{O}\left(\frac{K}{P_{cap}-K} \cdot \log n\right)$	$\mathrm{O}\left(\frac{K}{P_{cap}-K} \cdot \log n\right)$
B+-Tree	$\mathrm{O}(\log n)$	$\mathrm{O}(\log n)$	$\mathrm{O}(\log n)$

Tabela 5.1: Koszt operacji WE/WY

- N_i^{write} - sumaryczna liczba danych do zapisania podczas m_i scaleń dla poziomu i
- N_i^{read} - sumaryczna liczba danych do wczytania podczas m_i scaleń dla poziomu i
- t_{insert} - czas potrzebny na dodanie n element i przeprowadzenie m_i scaleń dla każdego poziomu i

Zacznijmy od rzeczy prostych. Wiemy że wskaźników wewnętrznych jest maksymalnie tyle ile stron na danym poziomie.

$$|internal_i| = \left\lceil \frac{|L_i|}{P_{cap}} \right\rceil$$

Analogicznie, wskaźników zewnętrznych jest dokładnie tyle ile stron na niższym poziomie.

$$|external_i| = \left\lceil \frac{|L_{i+1}|}{P_{cap}} \right\rceil$$

Wiemy, że zaraz po wykonaniu algorytmu scalania nowy poziom wyższy ma tylko wskaźniki zewnętrzne, stąd:

$$|L_{i-1}^j| = |external_{i-1}| \qquad j > 0$$
$$|L_{i-1}^j| = \left\lceil \frac{|L_i|}{P_{cap}} \right\rceil \qquad j > 0$$

Spróbujmy oszacować koszt zapisu dla poziomu i. Scalanie przepisuje dane z obu poziomów do nowych poziomów. Zatem, koszt scalania to sumaryczny koszt zapisów danych dla obu poziomów zaraz po zakończeniu scalania.

$$N_i^{write} = \sum_{j=1}^{m_i} (|L_{i-1}|^j + |L_{i-1}^j|)$$

Wiemy, że poziom L_{i-1} zawiera tylko wskaźniki zewnętrzne. Na podstawie ich liczby możemy oszacować ile było danych poniżej.

$$|L_i^j| = |external_{i-1}| \cdot P_{cap}$$

Zatem mamy:

$$N_i^{write} = \sum_{j=1}^{m_i}(|external_{i-1}| \cdot P_{cap} + |L_{i-1}^j|)$$

$$N_i^{write} = \sum_{j=1}^{m_i}(|L_{i-1}^j| \cdot P_{cap} + |L_{i-1}^j|)$$

$$N_i^{write} = \sum_{j=1}^{m_i}((1 + P_{cap}) \cdot |L_{i-1}^j|)$$

$$N_i^{write} = (1 + P_{cap}) \cdot \sum_{j=1}^{m_i}|L_{i-1}^j| \tag{5.1}$$

Załóżmy teraz, że wszystkie rekordy n zostały scalone do najniższego poziomu. Musiały zatem zostać skopiowane do poziomu L_{i-1}, a później do L_i gdy poziom L_{i-1} był pełen.

$$n = \sum_{j=1}^{m_i}(|L_{i-1}| - |external_{i-1}^j|)$$

$$n = \sum_{j=1}^{m_i}(|L_{i-1}| - |L_{i-1}^j|)$$

$$n = m_i \cdot |L_{i-1}| - \sum_{j=1}^{m_i}|L_{i-1}^j|$$

$$n = m_i \cdot |L_{i-1}| - \sum_{j=1}^{m_i}\left\lceil \frac{|L_i^j|}{P_{cap}} \right\rceil$$

$$\sum_{j=1}^{m_i}|L_{i-1}^j| = (m_i \cdot |L_{i-1}| - n) \tag{5.2}$$

Łącząc ze sobą 5.1 z 5.2 dostajemy:

$$N_i^{write} = (1 + P_{cap}) \cdot (m_i \cdot |L_{i-1}| - n) \tag{5.3}$$

Spróbujmy przeprowadzić podobne rozumowanie dla liczby odczytów podczas scalania obu poziomów. Aby przeprowadzić scalanie musimy odczytać oba poziomu, mają one tyle danych ile wpisaliśmy po zakończeniu poprzedniego scalania, stąd:

$$N_i^{read} = \sum_{j=1}^{m_i}(|L_{i-1}^{j-1}| + |L_i^{j-1}|)$$

Spróbujmy teraz wyrazić N_i^{read} za pomocą N_i^{write}. Skorzystajmy z faktu, że możemy założyć, iż liczba danych na poziomie i po scaleniu j jest równa sumie 2 poziomów utworzonych po poprzednim scaleniu

$$|L_{i-1}^{j}| = |L_{i-1}^{j-1}| + |L_{i}^{j-1}|$$

Dzięki temu, możemy łatwo wyrazić N_i^{read} za pomocą N_i^{write}.

$$N_i^{write} = \sum_{j=1}^{m_i}(|L_{i-1}^{j}| + |L_i^{j}|)$$

$$N_i^{write} = \sum_{j=1}^{m_i}(|L_{i-1}^{j-1}| + |L_i^{j-1}| + |L_i^{j}|) \tag{5.4}$$

$$N_i^{read} = N_i^{write} - \sum_{j=1}^{m_i}|L_{i-1}^{j}|$$

$$N_i^{read} = N_i^{write} - \frac{n}{P_{cap}} \tag{5.5}$$

Jak widać odczytujemy mniej danych, ponieważ nie musimy odczytywać nowych wskaźników zewnętrznych na poziomie L_{i-1}^{j}.

Możemy teraz oszacować czas potrzebny na scalanie wszystkich poziomów. Zanim to zrobimy, warto znać górne ograniczenie na liczbę scaleń. Wiemy, że scaleń było tyle ile stosunek wszystkich wstawianych danych n do liczby rekordów, jakie może pomieścić dany poziom. Biorąc pod uwagę, że nie wszystkie rekordy to dane użytkownika dostajemy:

$$m_i < \frac{n}{|L_{i-1}| - \frac{|L_{i-1}|}{P_{cap}} - -\frac{|L_i|}{P_{cap}}}$$

$$m_i < \frac{P_{cap}}{P_{cap} - K - 1} \cdot \frac{n}{|L_{i-1}|}$$

Teraz w końcu jesteśmy gotowi aby oszacować czas potrzebny na wstawianie wszystkich elementów n oraz wszystkich reorganizacji poziomów. Użyjemy do tego czasy amortyzowanego:

$$t_{insert} = \frac{1}{n} \cdot \sum_{i=1}^{FA_{height}-1}(\frac{R_{size} \cdot N_i^{write}}{W_{cost}} + \frac{R_{size} \cdot N_i^{read}}{R_{cost}})$$

$$t_{insert} < \frac{W_{cost} + R_{cost}}{W_{cost} \cdot R_{cost}} \cdot P_{size} \sum_{i=1}^{FA_{height}-1}(\frac{m_i \cdot |L_{i-1}|}{n} - 1)$$

$$t_{insert} < \frac{W_{cost} + R_{cost}}{W_{cost} \cdot R_{cost}} \cdot P_{size} \sum_{i=1}^{FA_{height}-1}(\frac{P_{cap}}{P_{cap} - K - 1} - 1) \tag{5.6}$$

Wiemy, że wysokość trzeba FA_{height} jest logarytmiczna, uzależniona od współczynnika K oraz wielkości drzewa B+ w RAM czyli poziomu L_0. Wynosi więc ona:

	Pojemność		Prędkość		
Model	Strony	Bloku	Odczytu	Zapisu	Kasowania
Samsung K9F1G08U0D	2KB	64KB	58MB/s	8MB/s	1MB/s
Micron MT29F32G08ABAAA	8KB	1MB	234MB/s	23MB/s	5MB/s
Micron MT29F32G08CBEDBL83A3WC1	4KB	512KB	81MB/s	4,5MB/s	1,1MB/s

Tabela 5.2: Wybrane modele pamięci flash

$$FA_{height} = \left\lceil \frac{\log_K n}{|L_0|} \right\rceil$$

Możemy zatem przepisać nierówność 5.6, używając powyższej nierówności. Dostajemy:

$$t_{insert} < \frac{W_{cost} + R_{cost}}{W_{cost} \cdot R_{cost}} \cdot \frac{K+1}{P_{cap} - K - 1} \cdot 1) \cdot P_{size} \cdot \left\lceil \frac{\log_K n}{|L_0|} \right\rceil$$

$$t_{insert} = \mathrm{O}\left(\frac{K}{P_{cap} - K} \cdot \log n \right) \tag{5.7}$$

♦

5.2.5 Eksperymenty

W tej części rozdziału przedstawimy i przeanalizujemy eksperymenty wykonane za pomocą symulatora SIPS, który został dokładnie opisany w rozdziale 4. Do eksperymentów wybrano 3 kości flash typu NAND o odmiennej charakterystyce. Tabela 5.2 zawiera szczegółowe parametry wybranych modeli. Samsung K9F1G08U0D charakteryzuje się bardzo niską pojemnością zarówno strony (2KB) jak i bloku (64KB) oraz stosunkowo małą prędkością odczytu (58MB/s). Micron MT29F32G08CBEDBL83A3WC1 posiada większą pojemność strony i bloku, odpowiednio 4KB i 512KB. Jednak szybkość usuwania bloku jest porównywalna do kości Samsunga. Warto zwrócić uwagę, że pomimo szybkiego odczytu na poziomie 81MB/s, ten model pamięci charakteryzuje się bardzo wolnym zapisem równym tylko 4,5MB/s. Tak duża asymetria pomiędzy szybkościami zapisu i odczytu bardzo często występuje w kościach flash. Do eksperymentów wybrano także kość Micron MT29F32G08ABAAA o bardzo dobrych parametrach. Bardzo duża prędkość odczytu i zapisu to nie jedyne zalety tego modelu. Pomimo tak dużego bloku o wielkości aż 1MB, prędkość kasowania jest aż 5 razy większa od pozostałych modeli w tym zestawieniu i wynosi 5MB/s. Eksperymenty zostały przeprowadzone na dwóch tabelach popularnego zestawu kwerend TPC-C [137], który został dokładnie opisany w rozdziale 4.

Podstawowy zestaw kwerend

Podstawowy zestaw kwerend imituje proste wykorzystanie bazy danych podobne do wykorzystania zwykłych struktur danych. Takie wzorce kwerend są bardzo często wykorzystywane przez użytkowników urządzeń wbudowanych, ponieważ indeks na takim

Nazwa kolumny	Rozmiar kolumny
W_ID	8B
W_NAME	10B
W_STREET_1	20B
W_STREET_2	20B
W_CITY	20B
W_STATE	2B
W_ZIP	9B
W_TAX	8B
W_YTD	16B

Tabela 5.3: Sklep TPC-C

Nazwa kolumny	Rozmiar kolumny
NO_O_ID	8B
NO_D_ID	4B
NO_W_ID	8B

Tabela 5.4: Nowe Zamówienie TPC-C

urządzaniu działa jako właśnie pojedyncza instancja struktury danych wystawiająca prosty interfejs dla użytkownika i nie ma tam silnika bazy danych. W zestawach wyróżniamy trzy operacje: wstawianie, wyszukiwanie i usuwanie. Każda operacja przeprowadzana jest na jednym rekordzie. Oznacza to, że wyszukiwanie nie posiada selektywności jako parametr. Jest to wyszukiwanie punktowe, które dla podanego klucza zwraca rekord zapisany w indeksie lub komunikat błędu, gdy rekord nie został znaleziony. Podstawowy zestaw kwerend posiada trzy wzorce o odmiennej charakterystyce:

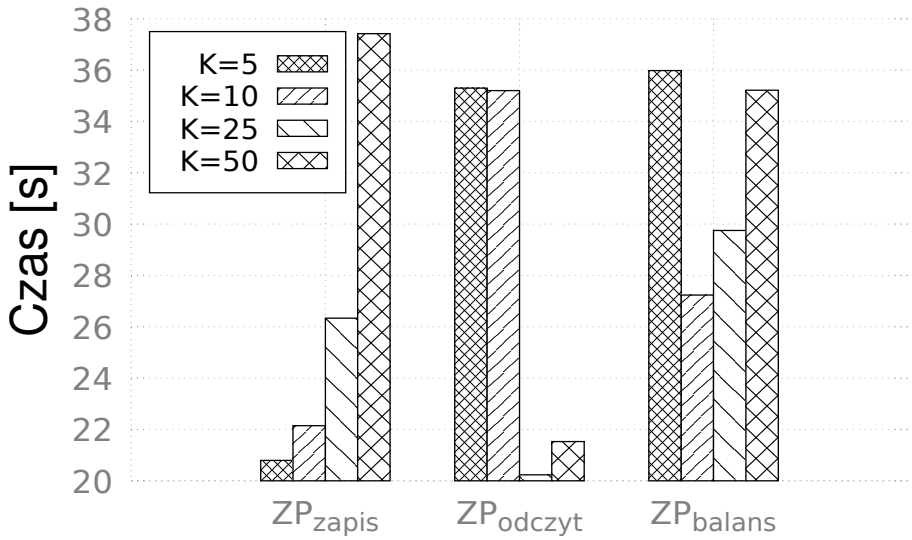
1. ZP_{zapis} - 60% operacji wstawiania pojedynczego rekordu, 20% operacji wyszukiwania pojedynczego rekordu, z użyciem klucza, 20% operacji usuwania pojedynczego rekordu

2. ZP_{odczyt} - 15% operacji wstawiania pojedynczego rekordu, 80% operacji wyszukiwania pojedynczego rekordu, z użyciem klucza, 5% operacji usuwania pojedynczego rekordu

3. ZP_{balans} - 37.5% operacji wstawiania pojedynczego rekordu, 50% operacji wyszukiwania pojedynczego rekordu, z użyciem klucza, 12.5% operacji usuwania pojedynczego rekordu

W tej serii eksperymentów wybraliśmy tabelę Sklep (5.3) o wielkości rekordu równym 113B. Indeksy, które wybraliśmy to drzewo B+, drzewo LSM oraz zaproponowany nowy indeks FA. Warto jednak pamiętać, że najpopularniejszym indeksem jest drzewo B+. Z kolei struktura LSM, ze względu na potrzebę przechowywania danych w pamięci operacyjnej RAM, nie jest zbyt często wykorzystywana w urządzeniach wbudowanych. Drzewo FA używało bufora w pamięci operacyjnej RAM do przechowywania poziomu L_0 o wielkości równej pojemności bloku flash. Indeks LSM używał bufora o tej samej wielkości również do tymczasowego przechowywania danych zanim one trafią do poziomu L_1. Dodatkowo używał bufora o wielkości 1MB na $SSTableMap$. Zestawy kwerend zostały wykonane z użyciem 100 tys. operacji. Tak więc zestaw ZP_{zapis} wykona 60 tys. zapisów, następnie wykona 20 tys. punktowych wyszukiwań pojedynczego rekordu, a na końcu 20 tys. razy usunie losowy rekord z tabeli. Początkowy stan tabeli to tabela pusta.

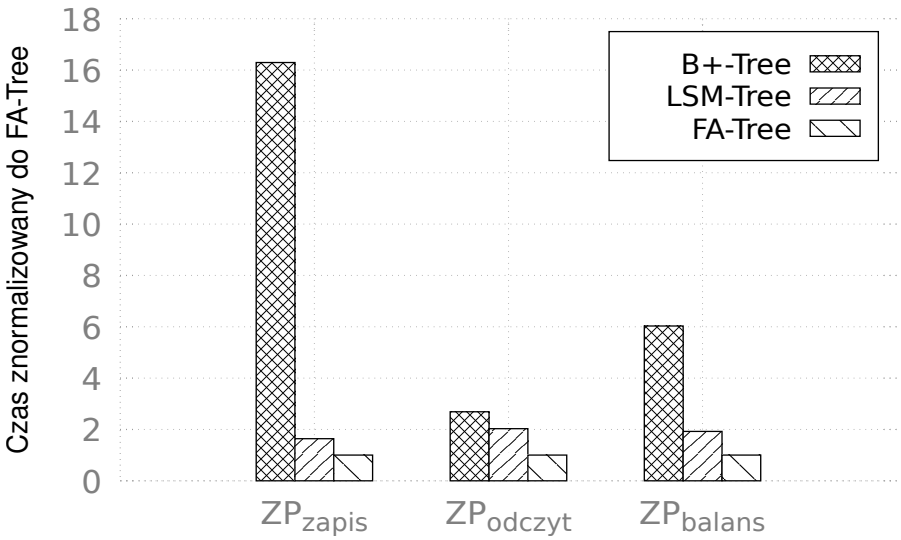
Zanim przejdziemy do porównania indeksu FA z indeksami B+ oraz LSM, przyjrzymy się jaki wpływ na szybkość wykonywania operacji ma główny parametr całego systemu FA czyli parametr K. Parametr ten określa współczynnik pomiędzy następującymi po sobie poziomami. Jeśli parametr K jest równy 5, to każdy kolejny poziom posiada



Rysunek 5.3: Czas wykonania 100tys. operacji dla różnych parametrów drzewa FA
Flash: Samsung K9F1G08U0D
Tabela: Sklep (113B)

5 razy większą pojemność od poprzedniego. Konfiguracja systemu bezpośrednio wpływa więc bezpośrednio na szybkość wykonywania operacji. Z rysunku 5.4 możemy wywnioskować, że im większe K tym wolniej wstawiamy kolejne elementy do naszego indeksu, ale samo wyszukiwanie jest szybsze. Dzieje się tak, ponieważ K bezpośrednio wpływa na wysokość drzewa. Im wyższe drzewo tym więcej poziomów musimy przeszukać aby znaleźć rekord dla odpowiedniego klucza. Stąd gdy K jest mniejsze, to wysokość jest większa, a to wpływa negatywnie na czas wyszukiwania. Podczas wstawiania rekordów mamy odwrotną sytuację. Gdy drzewo jest niskie, czyli posiada kilka dużych poziomów, pojedyncza reorganizacja trwa dłużej. Zauważmy, że K nie wpływa na rozmiar bufora (poziomu L_0). Zatem jeśli poziom L_1 jest duży, to za każdym scaleniem L_0 z L_1 musimy przepisać ogromną liczbę rekordów a to wydłuża czas wstawiania. Dodatkowo warto zauważyć, że parametr K wpływa na topologię drzewa. Podczas eksperymentów mamy do wykonania stałą liczbę operacji. Może się więc zdarzyć tak, że dla jednego ułożenia drzewa przeprowadzimy więcej reorganizacji niż dla innego ułożenia. Stąd w ZP_{balans} trend rosnący zaczyna się dopiero od $K = 10$. Ponieważ we wszystkich trzech modelach pamięci, pomimo odmiennej charakterystyki, otrzymaliśmy te same wyniki (ten sam trend), to w pracy zamieściliśmy tylko wyniki dla Samsung K9F1G08U0D, które opisane są na rysunku 5.4. Oznacza to, że wielkość strony, bloku jak i czas zapisu i odczytu nie wpływa na zmianę zachowania obserwowaną przy zmianie parametru K. Dzięki temu indeks można konfigurować do przewidywanego zestawu kwerend a nie do modelu pamięci. Gdy w zestawie przeważają zapisy, warto ustawić K na bardzo małe, zaś gdy przeważają odczyty, K powinno być dość duże (np. 25). Ponieważ pamięć flash posiada bardzo wolny zapis, to powinniśmy skupić się na optymalizacji tej operacji. Dlatego w kolejnych eksperymentach ustawiliśmy parametr K na wartość 5.

Tabele 5.5, 5.6, 5.7 przedstawiają dokładny czas jaki uzyskały indeksy podczas wykonywania zestawu kwerend na odpowiednio kościach Samsung K9F1G08U0D, Micron MT29F32G08CBEDBL83A3WC1, Micron MT29F32G08ABAAA. Możemy zaobserwować, że parametry kości flash, przekładają się na czas wykonywania kwerend, ale nie



Rysunek 5.4: Znormalizowany czas dla 100tys. operacji
Flash: Samsung K9F1G08U0D
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZP_{zapis}	338,86s	34,09s	20,79s
ZP_{odczyt}	94,93s	71,7s	35,29s
ZP_{balans}	216,9s	68,23s	35,98s

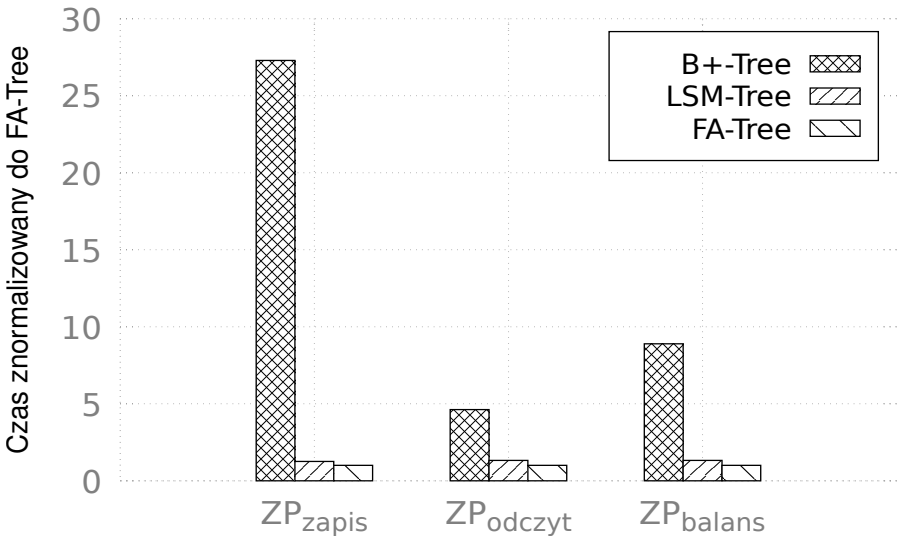
Tabela 5.5: Czas wykonania 100tys. operacji
Flash: Samsung K9F1G08U0D
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZP_{zapis}	644,26s	29,61s	23,6s
ZP_{odczyt}	171,19s	49,01s	37,01s
ZP_{balans}	409,31s	61,03s	46,03s

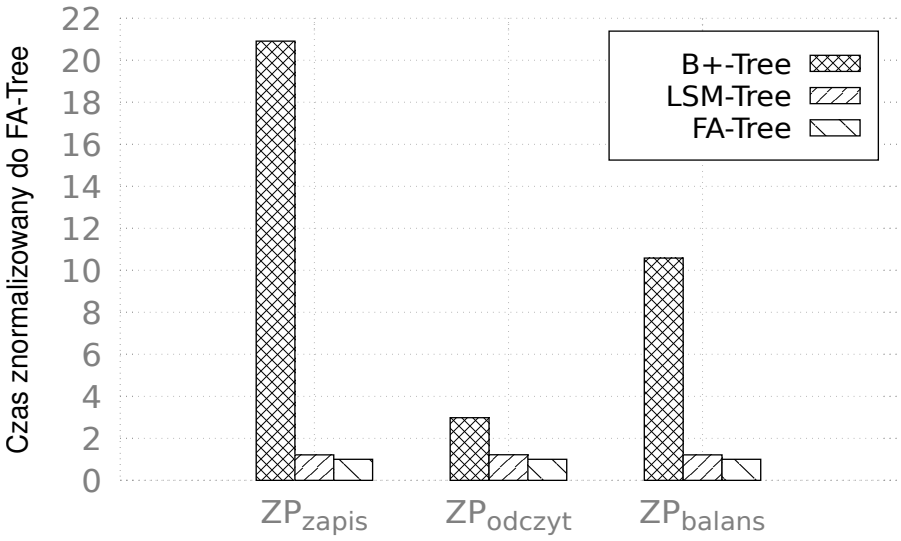
Tabela 5.6: Czas wykonania 100tys. operacji
Flash: Micron MT29F32G08CBEDBL83A3WC1
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZP_{zapis}	271,06s	15,76s	12,96s
ZP_{odczyt}	75,54s	30,9s	25,31s
ZP_{balans}	173,39s	19,88s	16,38s

Tabela 5.7: Czas wykonania 100tys. operacji
Flash: Micron MT29F32G08ABAAA
Tabela: Sklep (113B)



Rysunek 5.5: Znormalizowany czas dla 100tys. operacji
Flash: Micron MT29F32G08CBEDBL83A3WC1
Tabela: Sklep (113B)



Rysunek 5.6: Znormalizowany czas dla 100tys. operacji
Flash: Micron MT29F32G08ABAAA
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
$\boldsymbol{ZP_{zapis}}$	301MB	59MB	59MB
$\boldsymbol{ZP_{odczyt}}$	75MB	12MB	12MB
$\boldsymbol{ZP_{balans}}$	188MB	32MB	32MB

Tabela 5.8: Zużycie pamięci po 100tys. operacji
Flash: Samsung K9F1G08U0D
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
$\boldsymbol{ZP_{zapis}}$	588MB	28MB	28MB
$\boldsymbol{ZP_{odczyt}}$	147MB	4MB	4MB
$\boldsymbol{ZP_{balans}}$	367MB	15MB	15MB

Tabela 5.9: Zużycie pamięci po 100tys. operacji
Flash: Micron MT29F32G08CBEDBL83A3WC1
Tabela: Sklep (113B)

zmieniają trendu. Zawsze najwolniejszy był indeks B+, przed nim pracę kończył LSM, a zawsze najszybszy okazywał się indeks FA. Rysunki 5.4, 5.5, 5.6 przedstawiają wyniki tych samych eksperymentów co tabele, jednak zamiast wartości wyrażonej w sekundach, zawierają znormalizowany czas do czasu uzyskanego przez indeks FA (indeks FA ma zawsze wartość 1). Dzięki temu możemy łatwo stwierdzić jak nowe drzewo FA wypadło na tle pozostałych struktur danych.

Analizując tę serię eksperymentów możemy wywnioskować, że drzewo B+ nie jest przystosowane do pamięci blokowej o asymetrycznym koszcie zapisu i odczytu. W zestawach ZP_{zapis} i ZP_{balans} czas wykonania zapytań jest najwolniejszy. Wynika to z faktu, że drzewo B+ posiada dobry czas wyszukiwania danych, jednak czas zapisu nowych rekordów jest bardzo długi. Za każdym razem, gdy nowy rekord musi zostać wstawiony do węzła w drzewie, musimy przeorganizować węzeł, a czasami także wyższe poziomy. Taka reorganizacja odbywa się za pomocą kasowania bloków i ponownego nadpisania danych na stronie. Z tego względu wstawianie pojedynczych rekordów nie jest dobrym podejściem gdy korzystamy z tego rodzaju pamięci. Na kości flash Micron MT29F32G08CBEDBL83A3WC1, która posiada najwolniejszy czas zapisu z naszego zestawienia na poziomie 4,5MB/s drzewo B+ uzyskało 28 krotnie gorszy czas od indeksu FA. Podobną sytuację mamy przy użyciu najszybszej kości Micron MT29F32G08ABAAA. W tym przypadku drzewo B+ wykonało zestaw kwerend ZP_{zapis} 17 razy wolniej niż drzewo FA. Struktura LSM jest dostosowana do blokowych pamięci takich jak dyski HDD, kości flash i dyski SSD. Mimo to, wewnętrzna topologia tego indeksu powoduje, że jest on zoptymalizowany pod kątem wstawiania, usuwania rekordów jak i wyszukiwania wielu rekordów jednocześnie (ang. range search). W tej serii eksperymentów do wyszukiwania wykorzystujemy polecenie pojedynczego szukania rekordu (ang. point search). Jest ono zaprojektowane efektywnie na indeksie B+ oraz FA, jednak nie na indeksie LSM. Z tego powodu mimo efektywnego wstawiania rekordów, indeks LSM wykonuje zestawy kwerend wolniej od struktury FA na każdym modelu flash. Przykładowo na kości Micron MT29F32G08CBEDBL83A3WC1, zestaw ZP_{zapis} wykonał o 25% wolniej niż FA, zestaw ZA_{balans} 32% wolniej, a zestaw ZP_{odczyt} aż 35% wolniej.

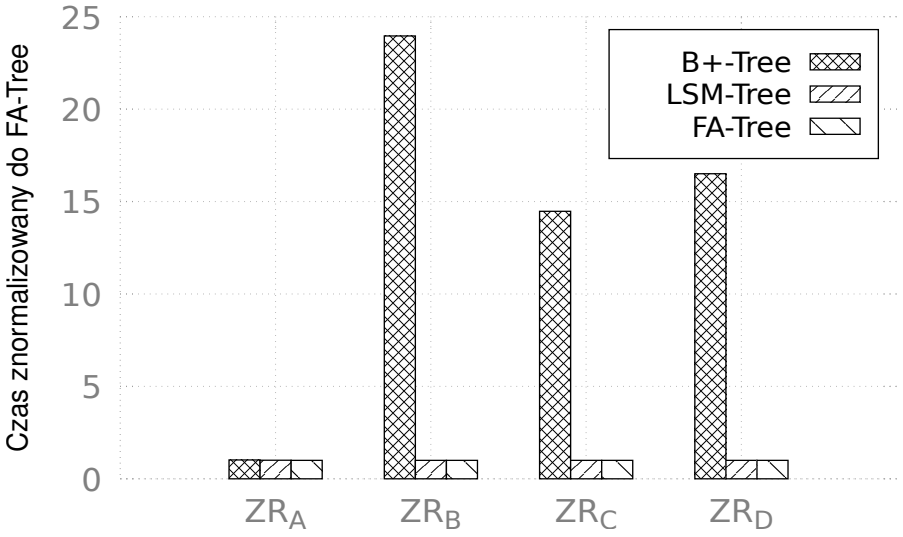
Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZP_{zapis}	1161MB	23MB	23MB
ZP_{odczyt}	290MB	2MB	2MB
ZP_{balans}	726MB	13MB	13MB

Tabela 5.10: Zużycie pamięci po 100tys. operacji
Flash: Micron MT29F32G08ABAAA
Tabela: Sklep (113B)

Oprócz samego czasu wykonania kwerend musimy także zwrócić uwagę na wykorzystanie pamięci. Sposób w jaki algorytm pracuje na pamięci przekłada się na żywotność kości flash. Im indeks przeprowadza mniej zapisów i kasowań, tym wykorzystanie pamięci (ang. wearout) jest mniejsze, a żywotność pamięci jest większa. Symulator SIPS dostarcza nam przybliżone dane na temat wykorzystania kości flash. Tabele 5.8, 5.9, 5.10 przedstawiają przybliżoną liczbę MB jaką indeksy zapisały lub nadpisały na modelach flash: Samsung K9F1G08U0D, Micron MT29F32G08CBEDBL83A3WC1, Micron MT29F32G08ABAAA. Zauważmy, że drzewo B+ nadpisuje nawet 123 razy więcej pamięci niż drzewa LSM i FA. Wynika to z algorytmu wstawiania. Drzewo B+ dodaje rekordy pojedynczo, za każdym razem reorganizując cały węzeł drzewa. Zatem za każdym razem trzeba przepisać minimalnie jedną stronę na nowo. Drzewa LSM i FA korzystają z małego bufora i dzięki algorytmowi scalania poziomów, dodają wiele rekordów jednocześnie minimalizując liczbę zużytej pamięci. Z tabel możemy jasno wywnioskować, że podobne topologie i algorytmy dodawania i usuwania rekordów w indeksach LSM i FA uzyskują bardzo zbliżone wyniki, nie różniące się więcej niż 1%. Dodatkowo warto zauważyć, że im większe bloki i strony, tym większą przewagę uzyskuje drzewo FA nad B+. Na modelu Micron MT29F32G08ABAAA, który posiada stronę o rozmiarze 8KB B+ nadpisuje 123 razy więcej pamięci niż FA. Jednak na modelu Samsung K9F1G08U0D o rozmiarze strony równej tylko 2KB, indeks B+ nadpisał tylko 6 razy więcej bajtów od indeksu FA. Wynika to z faktu, że indeksy dostosowują się do modelów pamięci. Drzewo B+ posiada rozmiar węzła taki sam jak rozmiar strony lub bloku (w zależności od ustawień, w eksperymentach był to rozmiar strony). Zatem im większa strona tym więcej bajtów należy przepisać podczas wstawiania rekordu. Sytuacja w indeksach FA i LSM jest odwrotna. Ich topologia wymusza aby bufor (poziom L_0) był minimalnie wielkości bloku kości flash. Zatem im większy blok, tym większy bufor, a to przekłada się na mniejszą liczbę reorganizacji poziomów, stąd na modelu Samsung K9F1G08U0D drzewo FA nadpisało 59MB (ZP_{zapis}), a na modelu Micron MT29F32G08ABAAA tylko 23MB (ZP_{zapis}).

Rozszerzony zestaw kwerend

Zestaw rozszerzony powstał na podstawie kwerend z TPC-C. Celem tego zestawu jest symulacja hurtowni danych, w której operacje są buforowane i wykonywane na wielu rekordach. W przeciwieństwie do zestawu podstawowego, tutaj używać będziemy wstawiania wielu danych jednocześnie (ang. bulkload), jeśli taka operacja jest wspierana przez indeks, a jeśli nie, to wstawianie będzie realizowane klasycznie, jeden po drugim. Indeksy FA jak i LSM pośrednio wspierają operację bulkload. Indeksy te mają wbudowany bufor, zatem operacja dodawania wielu rekordów jak i pojedynczego przeprowadzana jest dokładnie w ten sam sposób. Jest to ogromna zaleta tych struktur danych. Niestety drzewo B+ nie wspiera tej operacji, gdy struktura nie jest pusta. Zatem dla indeksu B+ będziemy



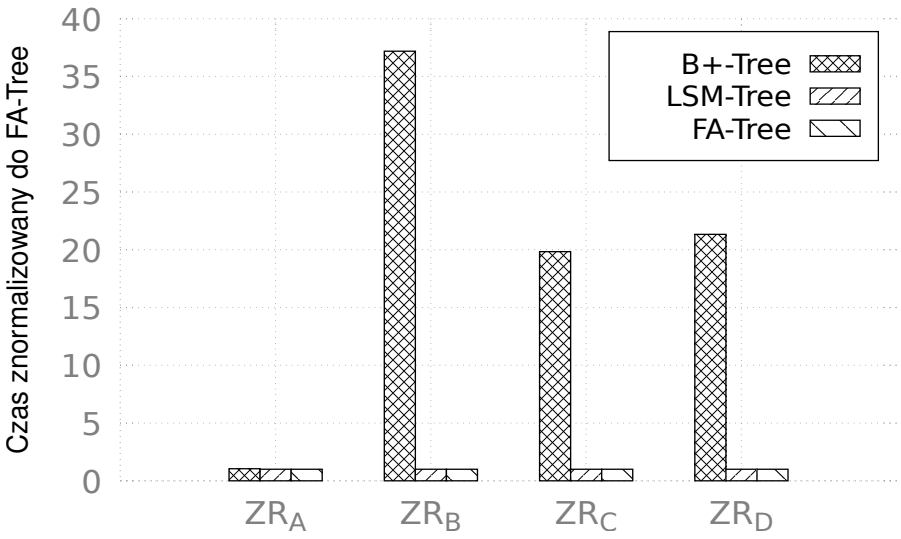
Rysunek 5.7: Znormalizowany czas
Flash: Samsung K9F1G08U0D
Tabela: Sklep (113B)

używać normalnego wstawiania. W tym zestawie kwerend używamy wyszukiwania całego zakresu rekordów zamiast punktowego wyszukiwania. Każde wyszukiwanie ma ustawioną selektywność na poziomie 1% całej tabeli. Tak więc zakres kluczy jest tak dobrany, aby wynikiem zapytania w przybliżeniu był 1% wszystkich rekordów. Tak jak w przypadku kwerend podstawowych, rozszerzony zestaw podzielony jest na serie. W każdej serii najpierw wykonywane jest dodawanie rekordów, później wyszukiwanie, a na końcu usuwanie rekordów. Biorąc pod uwagę charakterystykę wielu kwerend TPC-C oraz ich redukcję do podstawowych operacji na indeksach tylko na jednej tabeli, przygotowano aż 4 różne zestawy kwerend, które dokładnie opisano w rozdziale 4.

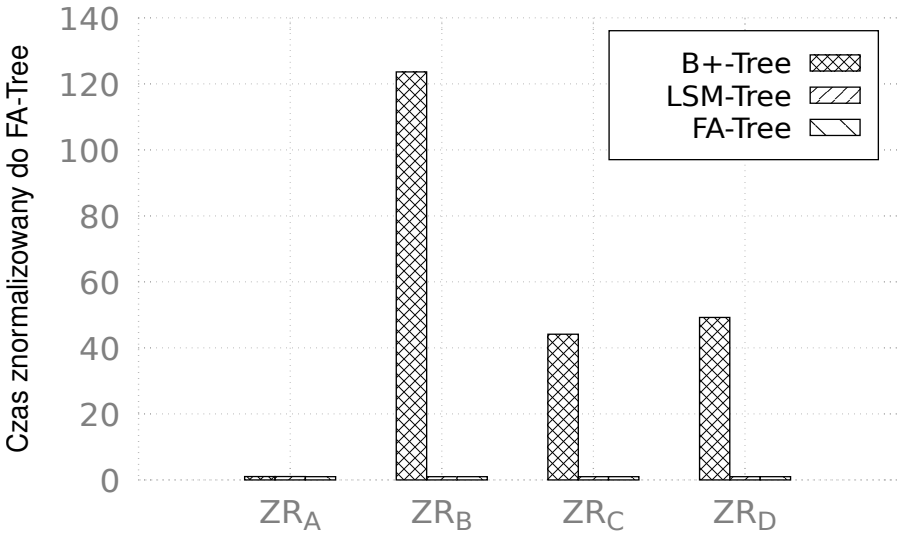
1. ZR_A - 100 serii, każda seria zawiera operacje dodawania 5 rekordów, 10 wyszukiwań o podanej selektywności 1% oraz usuwanie 5 rekordów,
2. ZR_B - 5 serii, każda seria zawiera operacje dodawania 100000 rekordów, 5 wyszukiwań o podanej selektywności 1% oraz usuwanie 100000 rekordów,
3. ZR_C - 10 serii, każda seria zawiera operacje dodawania 10000000 rekordów, 20 wyszukiwań o podanej selektywności 1% oraz usuwanie 1000000 rekordów,
4. ZR_D - 10 serii, każda seria zawiera operacje dodawania 1000000 rekordów, 10 wyszukiwań o podanej selektywności 1% oraz usuwanie 10000 rekordów.

Rozszerzony zestaw kwerend sprawdza się o wiele lepiej na tabelach zawierających już rekordy. Dlatego też początkowo indeksy będą zawierać 10mln rekordów.

Tabele 5.11, 5.12, 5.13 przedstawiają dokładny czas jaki uzyskały indeksy podczas wykonywania zestawu rozszerzonego kwerend na odpowiednio kościach Samsung K9F1G08U0D, Micron MT29F32G08CBEDBL83A3WC1, Micron MT29F32G08ABAAA. Łatwo możemy zaobserwować, że na każdym modelu pamięci, na wszystkich zestawach kwerend czas wykonania na indeksie LSM oraz FA jest bardzo podobny i różni się mniej niż 1%. Wynika to z kilku rzeczy. Po pierwsze, czas wstawiania i usuwania kolejnych rekordów w obu indeksach jest prawie taki sam, co wynika z podobieństwa topologii obu struk-



Rysunek 5.8: Znormalizowany czas
Flash: Micron MT29F32G08CBEDBL83A3WC1
Tabela: Sklep (113B)



Rysunek 5.9: Znormalizowany czas
Flash: Micron MT29F32G08ABAAA

Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZR_A	198s	195s	194s
ZR_B	1h	153s	153s
ZR_C	140h	9,7h	9,7h
ZR_D	13,4h	0,8h	0,8h

Tabela 5.11: Czas wykonania
Flash: Samsung K9F1G08U0D
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZR_A	145s	140s	139s
ZR_B	2h	186s	186s
ZR_C	268h	13,5h	13,5h
ZR_D	25,5h	1,2h	1,2h

Tabela 5.12: Czas wykonania
Flash: Micron MT29F32G08CBEDBL83A3WC1
Tabela: Sklep (113B)

Zestaw	B+-Tree	LSM-Tree	FA-Tree
ZR_A	51s	49s	49s
ZR_B	0,8h	23s	23s
ZR_C	112h	2,5h	2,5h
ZR_D	10,8h	0,2h	0,2h

Tabela 5.13: Czas wykonania
Flash: Micron MT29F32G08ABAAA
Tabela: Sklep (113B)

tur. Duże różnice w czasie wykonania możemy zaobserwować tylko podczas wyszukiwania punktowego, które przetestowaliśmy w zestawie podstawowym. Tutaj mamy wyszukiwania z całego zakresu. Ponieważ wszystkie trzy wybrane indeksy posiadają wsparcie dla tego wyszukiwania, czas na znalezienie pierwszego klucza jest w tej operacji marginalny. Najwięcej czasu zajmuje odczyt danych ułożonych obok siebie. Ponieważ we wszystkich wybranych przez nas indeksach algorytm postępuje tak samo, to uzyskuje ten sam czas. Zatem indeksy LSM oraz FA uzyskują te same czasy podczas operacji wstawiania, usuwania i wyszukiwania z zakresu. Z kolei drzewo FA ma dużą przewagę nad LSM podczas wyszukiwania punktowego. Dzięki wsparciu wyszukiwania z zakresu, mamy podobne czasy w zestawie ZR_A na wszystkich indeksach B+, LSM, FA. Potwierdza to naszą hipotezę, że drzewo B+ ma bardzo dobry czas wyszukiwania i punktowego i całego zakresu, ale, z drugiej strony, bardzo długi czas wstawiania i usuwania na kościach flash. Widać to bardzo dobrze porównując ze sobą czasy innych zestawów kwerend. Rysunki 5.7, 5.8, 5.9 przedstawiają wyniki tych samych eksperymentów co tabele, jednak zamiast absolutnej wartości wyrażonej w sekundach, zawierają znormalizowany czas do czasu uzyskanego przez indeks FA (indeks FA ma zawsze wartość 1). Z nich możemy wywnioskować, że im bardziej operacje dodawania nowych i usuwania starych rekordów przeważają w zestawie nad wyszukiwaniami, tym drzewo FA osiąga lepszy wynik na tle indeksu B+. Przykładowo zestaw ZR_B na kości Micron MT29F32G08ABAAA wykonywał się przez 0,8h na indeksie B+, zaś przy użyciu drzewa FA zaledwie 23s, a więc 123 razy szybciej. Dodatkowo możemy zauważyć, że im większy rozmiar strony i bloku pamięci flash, tym gorzej radzi sobie drzewo B+ w stosunku do drzewa FA. Wspominany wcześniej zestaw ZR_B, uruchomiony na Samsungu K9F1G08U0D (strona 2KB) wykonywał się 24 razy dłużej na indeksie B+ niż na indeksie FA, podczas gdy na modelach Micron MT29F32G08CBEDBL83A3WC1 (strona 4KB) i Micron MT29F32G08ABAAA (strona 8KB) zestaw ten wykonywał się odpowiednio 37 i 123 razy dłużej.

Oprócz wpływu wielkości strony na czas wykonywania się kwerend, sprawdziliśmy wpływ wielkości rekordów. Do tego eksperymentu wybraliśmy model Samsung, ponieważ

Zestaw	Nowe zamówienie (20B)	Sklep (113B)	Klient (719B)
ZR_A	1,08:1	1,02:1	1,01:1
ZR_B	190:1	24:1	8:1
ZR_C	95,3:1	14,5:1	6,2:1
ZR_D	105,9:1	16,5:1	7,7:1

Tabela 5.14: Stosunek czasu wykonania zestawów B+Tree do FA-Tree
Flash Samsung K9F1G08U0D

jest to najpopularniejszy model pośród wybranych przez nas kości flash. Tabela 5.14 pokazuje stosunek czasu wykonania się kwerend na drzewie B+ do drzewa FA. Jak widzimy, im mniejszy rekord, tym indeks B+ razi sobie gorzej w stosunku do indeksu FA. Wynika to z tego, że każdy zapis do węzła, niezależnie od jego wielkości, pochłania tyle samo czasu ze względu na reorganizację węzła oraz z faktu, że należy zawsze zapisać całą stronę. Ponieważ drzewo FA buforuje rekordy oraz operuje na całych blokach, to ani wielkość bloku, ani wielkość rekordu nie wpływa negatywnie na czas wykonania się poszczególnych operacji. To kolejna zaleta jaką warto zauważyć.

Podsumowując, indeks FA łączy ze sobą zalety drzewa B+ oraz indeksu LSM. Posiada on efektywne wyszukiwanie punktowe jak i rekordów z całego zakresu, tak jak drzewo B+. Dodatkowo przystosowany jest do pamięci blokowych uzyskując czasy dodawania i usuwania rekordów na poziomie indeksu LSM. Oprócz tego, indeks FA używa tylko jednego małego buforu do przechowywania danych w poziomie L_0. Bufor ten znacząco przyspiesza wykonanie kwerend, gdy jego wielkość jest równa rozmiarowi bloku pamięci flash. W przeciwieństwie do struktury LSM, nie używa on dodatkowych buforów, zatem zużycie pamięci RAM przez indeks FA jest minimalne. Warto również dodać, że indeks FA ma wbudowane mapowanie logiczne bloków, czyli posiada wbudowany algorytm FTL-blokowy. Zatem wykorzystanie indeksu FA na urządzeniach wbudowanych jest bardzo proste. Nie wymaga on skomplikowanego systemu operacyjnego czy kilku warstw oprogramowania na poziomie sterowników. Do działania wystarcza mu podstawowy sterownik pracujący z modelami flash. Szybkość wykonywania zapytań z prostego interfejsu jest nawet 28 razy szybsza niż czas uzyskany przez standardową strukturę danych B+. Dodatkowo samo zużycie pamięci jest 21 razy mniejsze, co umożliwia dłuższą eksploatację tej pamięci. Ten aspekt w systemach wbudowanych jest bardzo istotny, ponieważ wymiana takich urządzeń, często jest kłopotliwa (np. wymiana całej infrastruktury sieci LTE). Teoretyczna analiza algorytmów jak i praktyczne eksperymenty przeprowadzone na symulatorze SIPS jednoznacznie pokazały, że warto używać nowej struktury FA na urządzeniach wbudowanych zamiast drzewa B+ czy struktury LSM.

Algorytmy indeksowania na dyskach SSD

W tym rozdziale omówimy sposoby indeksowania danych, popularne algorytmy i struktury danych służące do indeksowania na dyskach SSD oraz przedstawimy nowe algorytmy indeksowania: indeksowanie wierszowe za pomocą struktury FALSM-Tree [2], indeksowanie kolumnowe na indeksie CF-Tree [3] oraz indeksowanie częściowe realizowane za pomocą systemu LAM [4]. Ogólna koncepcja każdego z rodzajów indeksowania została przedstawiona w rozdziale **2**. W tym rozdziale skupimy się na dostosowaniu algorytmów indeksujących do sposobu działania dysków SSD.

Charakterystyka dysków SSD została dokładnie omówiona w rozdziale **3**. Przypomnijmy sobie jednak główne cechy tej pamięci. Dyski SSD (ang. Solid State Disk) posiadają kompletnie odmienną charakterystykę od tradycyjnych dysków twardych (HDD) [100], [101]. Nie zawierają żadnych ruchomych części, a więc nie trzeba czekać aż głowica dysku przesunie się na odpowiednie miejsce tak jak w przypadku dysków HDD. Dzięki temu czas odczytu i zapisu jest znacznie szybszy. Zyskały na popularności głownie ze względu na: szybko rosnącą pojemność (obecnie kilku terabajtów), spadek kosztów produkcji, dobrą odporność na czynniki fizyczne, mały pobór mocy oraz dużą szybkość odczytu i zapisu. Dyski te składają się z kilku lub nawet kilkunastu kości flash typu NAND. Zatem tak samo jak w przypadku tych pamięci, na dyskach SSD wykonujemy operację odczytu i zapisu tylko na pełnych stronach, pamiętając, że nadpisanie danych odbywa się za pomocą długo trwającego procesu usuwania danych z bloków pamięci. W przeciwieństwie do zwykłych kości flash, dyski SSD wykonują tę pracę za nas, nie musimy martwić się o usuwanie bloków pamięci i o przepisywanie danych na inne strony. Ze względu na wykorzystanie wielu układów pamięci flash, aby w pełni wykorzystać potencjał pamięci, dysk SSD mapuje strony tak, aby móc jednocześnie odczytać lub zapisać dane na wszystkich kościach równolegle. Gdy użytkownik, poda większy zakres danych, to kontroler pamięci przestawia się w tryb sekwencyjny i pracuje znacznie szybciej niż normalnie. Niestety gdy użytkownik nie zna pełnego zakresu pamięci, dysk SSD może pracować tylko na pojedynczej kości flash. Wtedy taki tryb nazywamy losowym i jest on znacznie wolniejszy od trybu sekwencyjnego. Ze względu na skomplikowaną charakterystykę dysków SSD oraz niską przepustowość danych w trybie losowym, algorytmy indeksujące powinny maksymalizować użycie trybu sekwencyjnego, co znacząco zmniejsza czas obsługi kwerend.

6.1 Indeksowanie wierszowe

Indeksy wierszowe to najstarsze i najczęściej używane struktury danych w bazach danych. Zaletą tego typu indeksów jest prostota implementacji (o wiele łatwiej stworzyć indeks wierszowy niż kolumnowy) oraz szybkość utrzymania porządku na takim indek-

sie. Cały rekord zapisany jest w ciągłym obszarze pamięci. Dodawanie nowego rekordu, usuwanie starego jak i modyfikacje wykonują pojedyncze operacje na dysku. Z drugiej strony taki rodzaj indeksowania posiada kilka wad. Po pierwsze, wierszowe ułożenie rekordów nie sprzyja kompresji. Dodatkowo, zazwyczaj tabela składa się z kilku lub nawet kilkunastu atrybutów, a większość kwerend nie potrzebuje ich wszystkich na raz. Mimo to, ze względu na ułożenie w pamięci, musimy wczytać cały rekord, niezależnie od zapytania, co niepotrzebnie wydłuża czas trwania kwerendy. Najpopularniejszą strukturą danych wykorzystywaną do indeksowania wierszowego jest drzewo B+ [19]. Niestety, ze względu na odmienną charakterystykę dysków SSD od pamięci operacyjnej komputera RAM i dysków HDD, struktura ta nie jest dobrym wyborem. Drzewo B+ osiąga bardzo kiepską wydajność na dyskach SSD oraz na pamięci flash [68]. Mogliśmy to zaobserwować w poprzednim rozdziale **5** poświęconym indeksowaniu na pamięci flash. Drzewo B+ nie tylko osiągało gorszy czas obsługi kwerend (nawet 137 razy wolniejszy niż drzewo FA), ale także przeprowadzało o wiele więcej operacji zapisu i usuwania, co powodowało 23 krotne skrócenie żywotności pamięci względem drzewa FA.

Z tych powodów wiele nowych struktur zostało zaproponowanych. Do najważniejszych należą struktury: FD [25], LA [26] oraz LSM [172]. Drzewo FD podzielone jest na poziomy, a każdy z poziomów jest zbiorem stron dysku SSD. Pamięć alokowana jest od razu na cały poziom, zatem jest to ciągły obszar pamięci. Aby dodać rekord do drzewa FD, najpierw wstawiany jest on do niewielkiego bufora, następnie gdy bufor nie może pomieścić więcej danych, rekordy przepisywane są do pierwszego poziomu. Gdy pierwszy poziom osiągnął maksymalną liczbę rekordów, dane przepisywane są do kolejnego poziomu. Główną zaletą tego indeksu jest wykorzystanie sekwencyjnego trybu kontrolera dysku SSD. Każdy poziom to ciągły obszar pamięci. Dodatkowo, gdy przepisujemy dane, to czynimy to na pełnych poziomach, co oznacza, że dokładnie znamy obszar pamięci jaki chcemy wczytać i zapisać ponownie. Dzięki temu dysk SSD może przełączyć się na tryb sekwencyjny.

Indeks LA wprowadził mechanizm leniwej reorganizacji struktury drzewa. Zamiast przeprowadzać wszystkie operacje natychmiastowo, zapisywane są one do dziennika zdarzeń. Same rekordy lądują w buforze i oczekują na dodanie ich do struktury. W sytuacji gdy dziennik posiada maksymalną liczbę wpisów, to zaczyna się procedura aplikowania operacji do struktury. Takie podejście ma wiele zalet. Po pierwsze agregacja operacji może powodować niwelowanie niektórych z nich. Przykładowo jeśli mamy wstawić rekord i usunąć go, nie musimy nic robić. Gdy mamy wstawić rekord, a następnie znaleźć go za pomocą klucza, możemy od razu zwrócić rekord użytkownikowi z bufora. Dodatkowo, jeśli wstawiamy kilka rekordów do tego samego węzła w drzewie, możemy to zrobić za pomocą pojedynczego zapisu na dysku SSD. Mechanizm ten niestety posiada również kilka wad. Największa z nich to mała efektywność w zróżnicowanych zestawach kwerend. Jeśli mamy wstawiać rekord, wyszukać elementy o innym kluczu i usunąć dane o jeszcze innym kluczu, wówczas musimy wykonać różne operacje na odległych od siebie miejscach w drzewie, a więc agregacja tych operacji nic nie daje. Ponieważ dalej mamy strukturę podobną do drzewa B+, to dysk SSD głównie pracuje w trybie losowym (punktowym). Dodatkowo taki mechanizm wymaga dużo pracy podczas implementacji, aby był w pełni zabezpieczony na braki w dostawie prądu i inne niespodziewane wyłączenia systemu.

Struktura LSM jest bardzo podobna do drzewa FD. Również indeks podzielony jest na poziomy, z tym że poziom nie składa się ze zbioru stron dysku SSD (nie posiada wskaźników na każdą stronę). Poziom to po prostu ciągły obszar pamięci o ustalonej długości.

Pomiędzy poziomami nie ma żadnych wskaźników, a więc nie da się przejść przez indeks LSM tak jak przez klasyczne drzewo. Zamiast tego podzielono poziom na pewne podzbiory (ang. SSTable) w taki sposób, że każdy taki podzbiór (węzeł) jest posortowany. Jego własności takie jak miejsce w pamięci, minimalna i maksymalna wartość klucza są zapisane w buforze indeksu LSM i często w pamięci operacyjnej komputera RAM, aby mieć do tych danych szybki dostęp. Wstawianie jest praktycznie takie same jak w drzewie FD. Najpierw wstawiamy rekord do bufora, a gdy bufor nie może pomieścić więcej danych, przepisujemy dane do poziomu niższego i tak dalej, dopóki nie napotkamy na poziom, który ma miejsce na wszystkie nowe rekordy. Jedyną różnicą jest brak tworzenia wskaźników i zapisywania ich w wyższym poziomie. Zamiast tego tworzymy wpis z własnościami dla każdego nowego podzbioru danych o ustalonej wielkości i zapisujemy je do bufora metadanych. Ponieważ struktura LSM nie jest klasycznym drzewem, nie da się znaleźć klucza przechodząc od góry na dół, ponieważ nie mamy wskaźników na poziomy niżej. Zamiast tego przeszukujemy własności zapisanych węzłów i wybieramy tylko te, które mogą zawierać szukane klucze. Ponieważ i podczas wstawiania i podczas wyszukiwania znamy z góry fragmenty pamięci dysku SSD, które chcemy odczytać i zapisać, to używając indeksu LSM możemy w pełni wykorzystać tryb sekwencyjny kontrolera pamięci. Z tego powodu struktura LSM jest często używana nie tylko jako indeks w relacyjnych bazach danych, ale także jako indeks systemu plików, czy zwykła struktura danych przechowująca dane jako para klucz i wartość.

6.2 Struktura LSM

Obecnie bazy danych to często rozproszone systemy. Jeden serwer zbiera dane z kilku lub nawet kilkuset urządzeń, które wysyłają prośby o wykonanie kwerend. Chociaż pojedynczy użytkownik bazy, który korzysta z urządzenia podłączonego do serwera dodaje tylko kilka rekordów jednocześnie, to sumarycznie serwer dostaje w ciągu sekundy tysiące nowych rekordów do dodania. Z tego powodu coraz częściej spotyka się różnego rodzaju techniki buforowania danych [175], [176] oraz techniki, które dodatkowo maja na celu minimalizację zapisów na dysków SSD [177]. Dzięki takiemu buforowaniu, możemy agregować rekordy, które musimy dodać do indeksu w jedną kwerendę dodającą wszystkie nowe dane jednocześnie. Do tego służy operacja dodawania zbiorczego (ang. bulk loading). Wiele algorytmów zostało zaproponowanych aby zoptymalizować operację dodawania zbiorczego na indeksach, które wspierają taką operację [178], [179], [180]. Jednak nie wszystkie indeksy wspierają tę procedurę, np. drzewo B+ wspiera dodawanie zbiorcze tylko w sytuacji gdy drzewo jest puste. Indeksy takie jak FD, FA czy LSM wspierają dodawanie zbiorcze tylko w niewielkim stopniu. Jest to ich naturalny sposób dodawania rekordów, ponieważ każdy z wymienionych indeksów posiada niewielki bufor danych. Dopiero gdy bufor jest pełny to dane dodawane są do niższych poziomów. Takie zbiorcze wstawianie rekordów nie jest idealne. Gdy liczba danych jest ogromna, to wstawianie elementów od góry, nawet z buforowaniem nie jest dobrym pomysłem. W [181] zaproponowano nowy algorytm dodawania zbiorczego do indeksu LSM. Jednak zamiast na optymalizacji zapisów, skupiono się na równomiernym rozłożeniu danych na wszystkie poziomy, tak aby imitować normalne pojedyncze dodawanie do indeksu.

Ze względu na bardzo szerokie zastosowanie indeksu LSM, wiele optymalizacji tej struktury zostało zaproponowanych w ostatnich latach. Przegląd najważniejszych z nich można znaleźć w [182]. W [183] zaproponowano nową technikę reorganizacji drzewa. Za-

miast scalać ze sobą kolejne poziomy, scala się poziom z poziomem niższych posiadającym najmniej danych. W [184] podzielono rekordy na zbiory kluczy i zbiory wartości. Oba zbiory zapisane są w osobnych obszarach pamięci. Taki układ przyspiesza wyszukiwanie w drzewie, zwiększa efektywność kompresji, ale wydłuża czas wstawiania kolejnych rekordów i dodatkowo wymaga algorytmu łączenia klucza w odpowiedni rekord. Algorytm ten ze względu na asymetrię kosztów odczytywania i zapisania danych nie sprawdza się dobrze na dyskach SSD. W [185], [186] zaproponowano techniki optymalizacji indeksu LSM pod kątem dysków SSD i pamięci flash. Głównie skupiono się na alokacji pamięci, która minimalizuje liczbę użytych bloków oraz na kompresji danych wewnątrz bloku, tak aby móc zapisać więcej danych i zminimalizować liczbę operacji odczytu i zapisu. Wiele innych technik optymalizacji indeksu LSM zostało zaproponowanych pod kątem SSD. W [187] stworzono mechanizm dostosowywania się wielkości poziomów do przyszłych kwerend na podstawie analizy poprzednich, dzięki czemu koszt scalania poziomów był niższy niż w przypadku klasycznego LSM. W [188] zrezygnowano z sortowania SSTable, czyli podzbiorów danych, z których składa się poziom tego indeksu. Optymalizacja ta nie ma wpływu na liczbę odczytów i zapisów podczas scalania poziomów. Ma niestety negatywny wpływ na wyszukiwanie zakresowe. Zamiast odczytać tylko potrzebne dane, musimy wczytać cały węzeł, który składa się często z kilku lub nawet kilkunastu stron. Tę optymalizację zastosowano do urządzeń wbudowanych o małym rozmiarze bufora, dzięki czemu sortowanie zewnętrzne podczas dodawania działa szybciej, kosztem wolniejszego wyszukiwania. Techniki tej nie powinno się stosować na serwerach, które posiadają dużą moc obliczeniową i spore zasoby pamięci operacyjnej RAM.

6.3 Flash Aware LSM-Tree

Ze względu na szerokie zastosowanie indeksu LSM oraz brak odpowiedniego algorytmu potrafiącego dostosować operacje dodawania zbiorczego do dysków SSD, w [2] zaproponowaliśmy nową wersję indeksu LSM nazwaną FA-LSM (ang. Flash Aware LSM-Tree). Struktura ta wprowadza nowy algorytm dodawania całego zbioru rekordów oraz zmienia układ poziomów. Poziom składa się z kilku ciągłych obszarów pamięci. Nie jest on jednak posortowany. Umożliwia to dodanie rekordów do każdego poziomu, bez zbędnego wywoływania algorytmu scalania. Oznacza to, że nie musimy dodawać nowych rekordów z góry na dół, tak jak to ma miejsce w przypadku klasycznego LSM. Możemy zamiast tego dodać zbiór rekordów do dowolnego poziomu. Eksperymenty na zbiorze kwerend rozszerzonych z użyciem symulatora SIPS wykazały 5 krotnie lepszy czas względem zwykłego LSM oraz 6 krotnie mniejsze wykorzystanie pamięci flash na dysku SSD co przekłada się na dłuższe życie komórek pamięci jak i całego dysku.

6.3.1 Struktura FALSM

Nasza nowa struktura FALSM tak samo jak zwykłe LSM składa się z kilku poziomów. Poziom L_0 jest to bufor nowych danych, który znajduje się w szybkiej pamięci operacyjnej RAM. Jego wielkość ustalana jest indywidualnie dla każdego systemu. Zazwyczaj ma on wielkość 2MB-16MB. Każdy kolejny poziom L_i, $i > 0$ zapisany już jest na dysku SSD. Wielkość poszczególnych poziomów jest zależna od parametru k, który zazwyczaj wynosi $4 - 10$. Oznacza to, że jeśli L_0 ma wielkość 2MB, k jest równe 4, to L_1 ma pojemność 8MB, L_2 32MB itd. FALSM również nie ma wskaźników na poziomy niżej. Każdy poziom składa się z węzłów (SSTable), czyli ciągłych obszarów pamięci. Każdy taki podzbiór

rekordów ma pewien początkowy adres, wielkość, wartość minimalnego i maksymalnego klucza zapisanego w tym zbiorze rekordów. Te atrybuty nazywamy własnościami węzła, a zapisane są one w dodatkowym buforze, który posiada wszystkie niezbędne metadane. Tym co różni strukturę FALSM od LSM jest wprowadzenie dwóch rodzajów węzłów:

- Normalny węzeł (ang. Normal SSTable) - część poziomu, która tak jak w przypadku zwykłego LSM jest w pełni posortowana. Każdy taki węzeł posiada odpowiedni wpis o swoich własnościach w buforze. Na danym poziomie są zakresy kluczy, stworzone z atrybutów minimalnej i maksymalnej wartości klucza, które dla każdego węzła są zbiorami rozłącznymi,
- Dopisany węzeł (ang. Overflow SSTable) - część poziomu, która została dopisana podczas dodawania zbiorczego. Sam węzeł jest posortowany. Jednak ze względu na to, że część poziomu została dopisana w inny sposób niż przy reorganizacji drzewa, występowanie dopisanego węzła oznacza, że poziom nie jest w pełni posortowany. Każdy dopisany węzeł również posiada wpis w buforze, który jest specjalnie oznaczony, aby wskazać, że poziom nie jest już w pełni posortowany. Podczas reorganizacji poziomów sortujemy oba poziomy w nowy, w pełni posortowany poziom. Zatem podczas reorganizacji indeksu dopisane węzły zostaną przekształcone w normalne węzły, a zakres kluczy może mieć części wspólne z innymi węzłami normalnymi jak i dopisanymi.

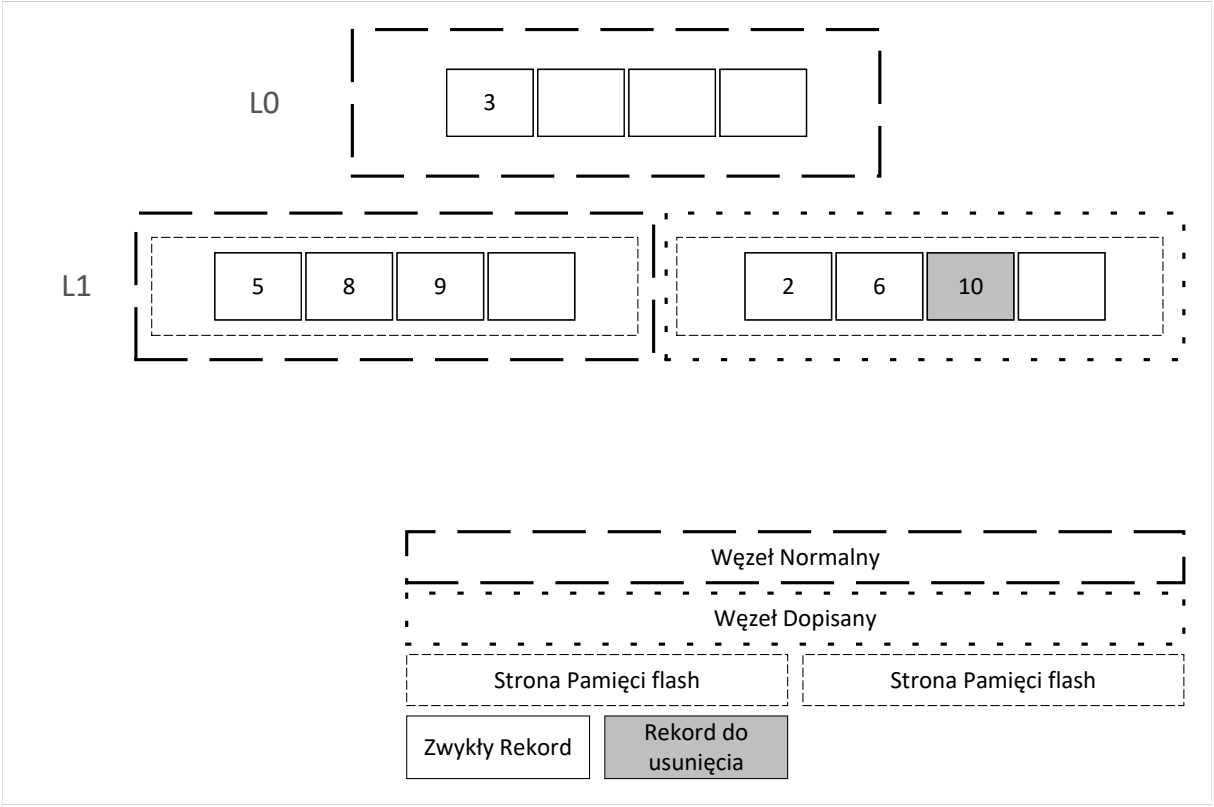
Każdy z węzłów posiada określoną wielkość, która zwykle równa jest rozmiarowi bufora (L_0). Maksymalna pojemność normalnego węzła i dopisanego węzła jest taka sama. Podczas dodawania zbiorczego możliwe jest zatem stworzenie kilku dopisanych węzłów. Ponieważ wprowadzenie dopisanego węzła sprawia, że poziom nie jest posortowany (tylko węzły są posortowane wewnętrznie), to wyszukiwanie również musi ulec zmianie. Normalnie gdy szukamy klucza, wiemy że może on występować tylko w jednym węźle na danym poziomie. W strukturze FALSM klucz może występować w jednym normalnym węźle i w każdym dopisanym węźle danego poziomu. Oczywiście za wybór węzłów odpowiada bufor posiadający atrybuty węzłów, a podczas wyszukiwania wybierzemy tylko te, które mogą zawierać szukany klucz, na podstawie minimalnego i maksymalnego klucza danego węzła.

Wprowadzenie dopisywania nowych węzłów podczas dodawania zbiorczego drastycznie zmniejszyło liczbę zapisów na dysku SSD. Przykładowo, załóżmy że poziom L_0 może pomieścić 10 rekordów, zaś poziom L_1 40 rekordów. Podczas dodawania zbiorczego 10 rekordów klasyczne LSM najpierw wstawiłoby 10 rekordów jeden po drugim do bufora. Następnie bufor zostałby przepisany do L_1, przy czym koszt takiego przepisania to zapis 10 rekordów na dysk SSD. Potem ponownie wstawiamy 10 nowych danych pojedynczo do bufora i ponownie scalamy L_0 z L_1. Koszt tego scalania to odczyt 10 rekordów i zapis 20 rekordów. Zatem wstawianie 20 danych kosztowało nas łącznie 10 odczytanych rekordów i zapis aż 30 rekordów. Struktura FALSM działa inaczej. Wybiera poziom, który jest w stanie zmieścić wszystkie rekordy, a jeśli takiego poziomu nie ma, tworzy nowy. W naszym przykładzie poziom L_1 posiada odpowiednią pojemność dla 20 rekordów. Zatem wpisujemy od razu do L_1 poziomu wszystkie 20 danych. Koszt takiego wstawiania to po prostu zapis 20 rekordów, co jest znaczącą poprawą w stosunku do klasycznego algorytmu wstawiania do indeksu LSM. Im więcej poziom może posiadać dopisanych węzłów, tym dłużej będzie trwać wyszukiwanie. Z drugiej jednak strony szybciej będziemy w stanie dodawać kolejne elementy. Ponieważ liczba dopisanych węzłów wpływa na szybkość wykonywania kwerend, wprowadziliśmy parametr T (6.1) określający jak dużo nowych danych można

dopisać za jednym razem do obecnego poziomu. Przykładowo jeśli $T = 2$, to możemy za pomocą jednego dodawania zbiorczego dopisać rekordy o rozmiarze 50% pojemności poziomu. W tej procedurze nie bierzemy pod uwagę obecnego stanu poziomu, ale jego maksymalną pojemność. Oznacza, to że wybrany przez algorytm poziom może nie posiadać już miejsca na nowe dane. Wtedy najpierw scalamy wybrany poziom z niższym poziomem a następnie wpisujemy nowe rekordy do wybranego poziomu.

$$liczbaWpisywanychDanych \leqslant \frac{maksymalnaLiczbaDanych}{T} \tag{6.1}$$

Wiedząc jak będzie wyglądał zestaw kwerend można odpowiednio skonfigurować strukturę FALSM za pomocą parametrów k i T, tak aby działała jak najlepiej. Zazwyczaj parametr k ustawiany jest na wartości pomiędzy 4 a 10, zaś parametr T pomiędzy 2 a 5.



Rysunek 6.1: Przykładowa struktura FA-LSM

Rysunek 6.1 przedstawia przykładowy układ struktury FALSM. Zakładamy, że $k = 2$, $T = 2$ oraz, że strona pamięci flash dysku SSD może pomieścić maksymalnie 4 rekordy. Na samej górze mamy poziom L_0, który nie jest zapisany na dysku, ale jest przetrzymywany w szybkiej pamięci RAM. Zawiera tylko jeden rekord o kluczu równym 3. Jak widać z poziomu L_0 nie ma żadnego wskaźnika na poziom L_1, a więc nie da się przejść po drzewie z góry na dół. Poziom L_1 składa się z dwóch węzłów. Pierwszy z nich po lewej jest to węzeł normalny, czyli taki który powstał poprzez scalenie poziomu L_0 z poprzednią wersją poziomu L_1. Zawiera on wartości 5, 8 i 9. Drugi węzeł to węzeł dopisany, stworzony został przez algorytm dodawania zbiorczego. Zawiera on trzy rekordy, z czego dwa są zwykłymi rekordami, a rekord z kluczem o wartości 10 to rekord do usunięcia. Aby zminimalizować

liczbę zapisów, drzewa takie jak FA, FD czy właśnie LSM i FALSM zamiast usuwać dane natychmiastowo, dodają nowy rekord o wartości klucza, który należy usunąć. Dzięki temu usuwanie odraczane jest w czasie. Zwróćmy uwagę, że mimo iż klucze się nie powtarzają oraz rekordy są posortowane wewnątrz węzłów, to już zakres kluczy powstały z atrybutów węzłów nie jest unikatowy (rozłączny). Zakres węzła normalnego to (5;9), a zakres węzła dopisanego to (2;10). Zatem jeśli będziemy chcieli wyszukiwać klucz o wartości 6, musimy przeszukać obydwa węzły, ponieważ mogą zawierać rekord o wskazanym kluczu.

6.3.2 Procedury indeksu FALSM

Wstawianie

Dodawanie nowego rekordu odbywa się dokładnie tak samo jak z wykorzystaniem klasycznego indeksu LSM. Najpierw rekord wstawiany jest do bufora w pamięci RAM. Dzięki temu nie wykonujemy kosztownych operacji zapisu na dysku za każdym razem. Gdy bufor nie będzie w stanie pomieścić kolejnego rekordu, zrzucany jest na dysk. Wykonujemy wtedy reorganizację poziomów L_0 oraz L_1. Ponieważ każdy z poziomów zawiera ograniczoną pojemność, to istnieje prawdopodobieństwo, że reorganizacja poziomów L_0 z L_1 przepełni poziom L_1. W takim przypadku rekurencyjnie scalamy poziomy L_1 z L_2 oraz wszystkie kolejne poziomy, gdy zajdzie tak potrzeba, dopóki poziom przyjmujący dane będzie miał wolne miejsce.

Wyszukiwanie

Ze względu na obecność dopisanych węzłów wyszukiwanie wykonujemy inaczej niż na strukturze LSM. Pamiętajmy, że dla każdego węzła, czy to normalnego czy to dopisanego mamy wpis w buforze metadanych zawierający jego właściwości. Zatem gdy szukamy klucza, musimy wczytać wszystkie węzły, w których taki klucz może się znajdować. Gdy poziom nie zawiera żadnego dopisanego węzła, wiemy że tylko jeden węzeł normalny może zawierać szukany klucz. Gdy poziom posiada węzły dopisane, należy odczytać wszystkie węzły na poziomie, które według atrybutów mogą zawierać szukany klucz. Należy również pamiętać, że jeśli napotkamy rekord do usunięcia, należy zakończyć procedurę wyszukiwania i jako wynik zgłosić użytkownikowi brak szukanego rekordu.

Usuwanie

Usuwanie rekordu o podanym kluczu wykonujemy dokładnie tak samo jak na indeksie LSM. Samo usuwanie nie jest wykonywane natychmiastowo jak to ma miejsce w drzewach B+. Zamiast tego wstawiamy nowy rekord z kluczem, który mamy usunąć i ustawiamy typ tego rekordu jako *Usunięta dana*. Gdy podczas reorganizacji poziomów natrafimy na oba rekordy: oryginalny oraz ten do usunięcia, wtedy usuwamy oba rekordy. Ta operacja realizowana jest jako brak zapisu poszczególnych rekordów ze starych zbiorów do nowego zbioru.

Scalanie poziomów

Reorganizacja drzewa przeprowadzana jest w przypadku, gdy któryś z poziomów nie ma wolnego miejsca na kolejne elementy. Wtedy uruchamiamy proces scalania dwóch poziomów: obecnego i kolejnego pod nim. Jeśli nie istnieje poziom niżej, wtedy jest on tworzony z odpowiednio dużą pojemnością (k razy większą od poprzedniego poziomu).

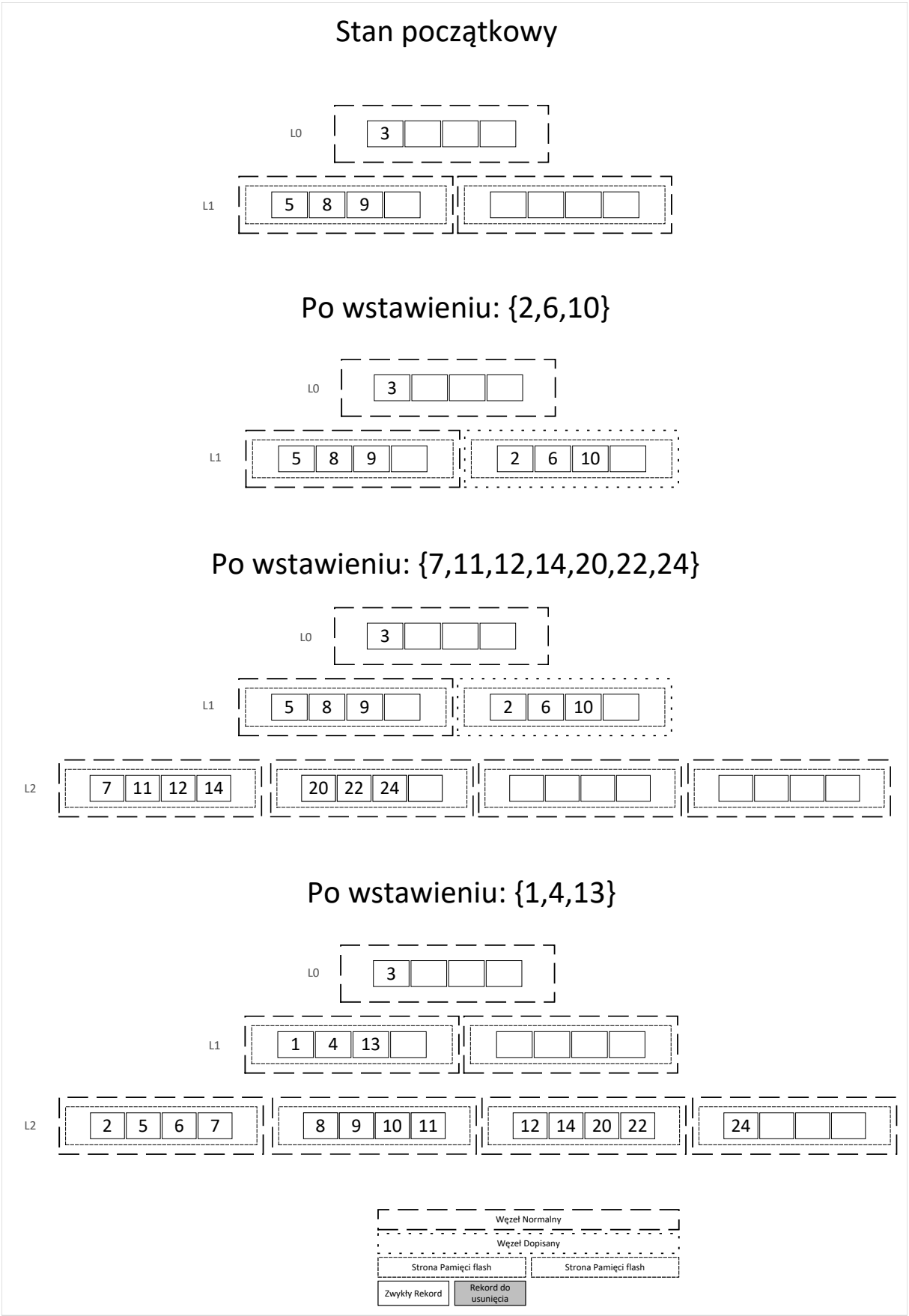
W klasycznym drzewie LSM oba poziomy są posortowane, a zatem scalanie wykonujemy algorytmem łączenia dwóch posortowanych list. W przypadku struktury FALSM, mamy podobną sytuację z tą różnicą, że poziom może zawierać wiele posortowanych zbiorów. Każdy z takich zbiorów (węzłów) możemy traktować jako posortowaną listę. Zatem aby scalić oba poziomy wystarczy użyć algorytmu łączenia wielu list. Realizowane jest to za pomocą kolejki priorytetowej, czyli struktury kopca. Początkowo do kopca dodajemy minimalny klucz z każdego węzła z obu poziomów. Za każdym razem wyciągamy z kopca rekord o minimalnym kluczu i dodajemy kolejny rekord z węzła. Dzięki temu w wyniku scalania otrzymamy nowy poziom w pełni posortowany. Oznacza to, że węzły dopisane połączą się z innymi węzłami dopisanymi jak i normalnymi tworząc poziom zawierający tylko normalne węzły. Podczas reorganizacji pamiętamy, że jeśli poziom L_i będzie zawierał rekord o tym samym kluczu co rekord na poziomie L_{i+1} ale z odpowiednim znacznikiem do usunięcia, wtedy pomijamy oba rekordy i żadnego z nich nie wpisujemy do nowego poziomu L'_{i+1}. Możliwe, że podczas tego procesu poziom L'_{i+1} również nie będzie miał wystarczająco miejsca aby pomieścić wszystkie dane. Wtedy wykonujemy scalanie poziomów L'_{i+1} z L_{i+2}. Proces ten powtarzamy rekurencyjnie do momentu uzyskania poziomu mieszczącego wszystkie rekordy.

Dodawanie zbiorcze

Wprowadzenie algorytmu dodawania zbiorczego jest główną cechą nowego indeksu FALSM. Aby dodać wiele rekordów jednocześnie zamiast wstawiać je do bufora i czekać na reorganizację struktury, indeks FALSM dodaje nowe dane od razu do poziomu, który ma odpowiednią pojemność. Początkowo szukamy poziomu, który spełnia warunek 6.1. Wybieramy poziom o najmniejszym indeksie z tych, które spełniają podany warunek. Jeśli wybrany poziom pomieści wszystkie nowe rekordy, to tworzymy nowe węzły dopisane i dodajemy je do poziomu. Tym samym poziom przestaje być posortowany. Zauważmy, że bierzemy pod uwagę liczbę danych do dodania oraz maksymalną pojemność poziomów, a nie ich obecny rozmiar. Zatem wybrany poziom, może nie posiadać już miejsca na nowe rekordy. W tym wypadku najpierw przeprowadzamy scalanie wybranego poziomu z poziomem następnym, a dopiero wtedy dodajemy nowe dane do wybranego poziomu. Ponieważ w tej sytuacji nowy poziom nie zawiera żadnych węzłów, to dopisane węzły stają się węzłami normalnymi, a więc wszystkie rekordy są posortowane. Możemy też mieć sytuację, w której żaden z istniejących poziomów nie spełnia warunku 6.1. Oznacza to, że liczba danych do wstawienia jest na tyle duża, że nie zmieści się w istniejącym drzewie FALSM. Wtedy tworzymy nowe poziomy wypełniając je w całości aż do momentu wyczerpania nowych danych.

Rysunek 6.2 przedstawia przykładowy proces dodawania zbiorczego rekordów do struktury FALSM. Na potrzeby przykładu zakładamy, że $k = 2$, $T = 2$ oraz że węzeł składa się z jednej strony pamięci flash, a strona dysku może pomieścić 4 rekordy. Początkowo mamy jeden rekord w buforze (L_0) o wartości 3. Pozostałe rekordy mieszczą się w jednym normalnym węźle na poziomie L_1. Gdy wykonujemy wstawianie zbiorcze dodatkowych trzech elementów ($\{2, 6, 10\}$), to szukamy poziomu, który ma minimalnie pojemność 6 (ponieważ $T \cdot 3 = 6$). Takim poziomem jest L_1. Zatem tworzymy węzeł dopisany i dodajemy go do poziomu L_1. Od tego momentu każde wyszukiwanie na tym poziomie musi dostosować się do faktu, iż poziom nie jest w pełni posortowany. Kolejną kwerendą jest dodanie zbiorcze aż 7 nowych rekordów ($\{7, 11, 12, 14, 20, 22, 24\}$).

[h]



Rysunek 6.2: Przykład dodawania zbiorczego na strukturze FALSM

Zatem musimy znaleźć poziom o pojemności przynajmniej 14. Niestety struktura FALSM

nie posiada takiego poziomu. Zatem tworzymy nowy, który będzie miał pojemność 16 i wpisujemy nowe rekordy do tego poziomu. Warto zaznaczyć, że wpisujemy nowe elementy do normalnych węzłów, ponieważ poziom był pusty. Pozostałe poziomy są bez zmian, ponieważ dodawanie elementów odbyło się bezpośrednio na poziomie L_2. Ostatnią kwerendą w tym przykładzie jest dodanie trzech kolejnych rekordów ($\{1, 4, 13\}$). Szukamy więc poziomu z pojemnością przynajmniej 6. Takim poziomem jest L_1. Jednak obecnie nie jest w stanie pomieścić aż tylu danych. Zatem zanim dodamy do niego nowe elementy, musimy przeprowadzić reorganizacją struktury. Scalamy ze sobą L_1 z L_2. W wyniku tego powstaje nowy poziom L'_2, który zawiera wszystkie rekordy ze starych poziomów L_1 i L_2. Podczas tego scalania odbyła się całkowita reorganizacja poziomu L_2, w L'_2. Nowy poziom, zawsze jest posortowany i składa się tylko z normalnych węzłów, które zawierają rekordy z poziomów L_1 oraz L_2. Oznacza to, że podczas reorganizacji przekształciliśmy wszystkie dopisane węzły z obu poziomów w węzły normalne dzięki ponownemu sortowaniu rekordów. Ostatnim etapem reorganizacji jest stworzenie nowego pustego poziomu L'_1. Ponieważ jego pojemność maksymalna jest dokładnie taka sama jak starego poziomu L_1, wiemy że nie musimy ponownie szukać poziomu dla nowych rekordów. Wstawiamy zatem elementy $\{1, 4, 13\}$ do nowego poziomu L'_1 jako węzeł normalny, ponieważ podczas wstawiania poziom był pusty.

6.3.3 Algorytm dodawanie zbiorczego

Pseudokod 6.1: falsmBulkloading(Entry *entriesToInsert*)

```
1  Niech height będzie wysokością drzewa FA-LSM
2  Niech numEntries będzie rozmiarem zbioru wejściowego danych entriesToInsert
3  Niech maxEntry[i] będzie maksymalną liczbą danych jaką może pomieścić
     poziom i
4  Niech maxEntryInBuffer będzie maksymalną liczbą danych jaką może
     pomieścić bufor (L_0)
5
   // Jeśli bulkload się nie opłaca, dodaj dane do bufora
6  if numEntries < maxEntryInBuffer then
7      addToBuffer(entriesToInsert)
8      return
9
   // Znajdź odpowiednio duży poziom, uwzględniając parametr T
10 lvl := 1
11 while lvl <= height do
12     if numEntries < ⌈maxEntry[lvl] / T⌉ then
13         addToLevel(entriesToInsert, lvl)
14         return
15     lvl := lvl + 1
16
   // Poziom nie został znaleziony, musimy stworzyć nowe poziomy i wpisać tam dane
17 pos := 0
18 while numEntries > 0 do
19     createLevel(lvl)
20     n := min(maxEntry[lvl], numEntries)
21
       // Dodaj obecny podzbiór danych entriesToInsert do poziomu, wypełniając go w całości
22     Entry entriesToLoad[ ] := truncateEntrySet(entriesToInsert, pos, n)
23     addToLevel(entriesToLoad, lvl)
24
25     numEntries := numEntries - n
26     pos := pos + n
27     lvl:=lvl + 1
```

W tej sekcji dokładnie omówimy nowy algorytm dodawanie zbiorczego do indeksu FALSM. Ze względu na dość duży poziom skomplikowania, pseudokod został podzielony na 2 części: funkcję *falsmBulkloading* 6.1, która jest główną funkcją interfejsu struktury oraz funkcję *addToLevel* 6.2, która jest funkcją pomocniczą odpowiedzialną za dodanie nowych elementów do wybranego poziomu.

Zacznijmy od omówienia funkcji głównej. Funkcja *falsmBulkloading* przyjmuję zbiór nowych rekordów jako argument wejściowy, w zależności od jego rozmiaru, możemy podjąć decyzję o wykonaniu klasycznego dodawania rekordów do bufora. Linie 6-8 są za to odpowiedzialne. Gdy liczba danych jest mniejsza niż rozmiar bufora, to oznacza, że jest

Pseudokod 6.2: addToLevel(Entry *entriesToInsert*, int lvl)

```
1  Niech numEntries będzie rozmiarem zbioru wejściowego danych entriesToInsert
2  Niech maxEntry[i] będzie maksymalną liczbą danych jaką może pomieścić
     poziom i
3  Niech numEntry[i] będzie obecną liczbą danych na poziomie i
4
     // Poziom nie pomieści więcej danych, musimy wykonać scalanie poziomów
5  if numEntry[lvl] + numEntriesToInsert > maxEntry[lvl] then
6      Entry entriesToMerge[]:= getEntries(lvl)
         // Poziom niżej istnieje, wczytaj zatem dane z poziomu lvl + 1
7      if exists lvl + 1 then
8          Entry entries[]:= getEntries(lvl + 1)
9          entriesToMerge:= entriesToMerge ∪ entries
10         removeEntries(lvl + 1)
11         removeLevel(lvl + 1)
12
           // Dane zostały wczytane, nawet jeśli poziom istniał, został on usunięty
13     createLevel(lvl + 1)
14     Entry sortedEntries[]:= sortEntries(entriesToMerge)
15
           // Wykonaj rekurencyjnie dodanie danych do pustego poziomu lvl + 1
16     addToLevel(sortedEntries, lvl + 1)
17
     // Jeśli poziom jest pusty, możemy stworzyć normalne węzły, w przeciwnym wypadku
        tworzymy węzły dopisane
18 if isEmpty(lvl) then
19     NormalSSTable tab[] := createNormalSSTables(lvl)
20     addToSSTables(tab, entriesToInsert)
21 else
22     OverflowSSTable tab[] := createOverflowSSTables(lvl)
23     addToSSTables(tab, entriesToInsert)
```

ich tak mało, że nie powinniśmy tworzyć dla tych danych węzła dopisanego, który zostanie dodany do jakiegoś poziomu. Spowodowałoby to zbyt dużą liczbę takich węzłów, zatem spowolniłoby to znacząco algorytm wyszukiwania danych za pomocą klucza. Gdy danych jest więcej niż rozmiar bufora, musimy znaleźć odpowiedni poziom, który nie tylko posiada odpowiednio dużą pojemność, ale także spełnia warunek 6.1 określony przez parametr T (linie 10-15). Gdy taki poziom znajdziemy (linie 12-14), to możemy wykonać funkcję *addToLevel* i zakończyć algorytm dodawania zbiorczego. Istnieje jednak sytuacja, w której indeks FALSM nie posiada odpowiedniego poziomu. Wówczas taki poziom (lub wiele poziomów) należy stworzyć (linia 19), dodać go do struktury FALSM, a następnie wpisać do nich maksymalną liczbę nowych rekordów (linie 22 i 23).

Funkcja *addToLevel* służy do dodawania elementów do wybranego poziomu. Jako argumenty wejściowe przyjmuje właśnie zbiór rekordów do dodania oraz numer poziomu, do którego należy dodać ten zbiór. Możliwe, że dodane wartości nie zmieszczą się w wybra-

nych poziomie (linie 5-11). Wtedy musimy wykonać scalanie poziomów, a więc wczytać wszystkie elementy poziomu lvl oraz poziomu $lvl + 1$, jeśli taki istnieje (7-11). Gdy w buforze znajdują się wszystkie elementy, tworzymy nowy poziom (linia 13), sortujemy otrzymane rekordy (linia 14), a następnie dodajemy je do nowego poziomu za pomocą tej samej funkcji (rekurencyjnie wołamy funkcję *addToLevel* w linii 16). Następnie, niezależnie od tego, czy musieliśmy przeprowadzić reorganizację czy nie, możemy zapisać zbiór nowych rekordów do wybranego poziomu. Jeśli wybrany poziom jest pusty, możemy stworzyć normalne węzły i dodać je do poziomu (18-20). W przeciwnym przypadku, musimy stworzyć węzły dopisane i zapisać je na końcu obecnego poziomu (linie 21-23).

6.3.4 Eksperymenty

		Pojemność		Prędkość losowego		Prędkość sekwencyjnego	
Model	Interfejs	Strony	Bloku	Odczytu	Zapisu	Odczytu	Zapisu
Samsung 840	SATA	8KB	512KB	390MB/s	182MB/s	585MB/s	535MB/s
Toshiba VX500	SATA	4KB	256KB	379MB/s	267MB/s	568MB/s	525MB/s
Intel DCP4511	NVMe	4KB	256KB	1,2GB/s	240MB/s	2GB/s	1.47GB/s

Tabela 6.1: Wybrane modele dysków SSD

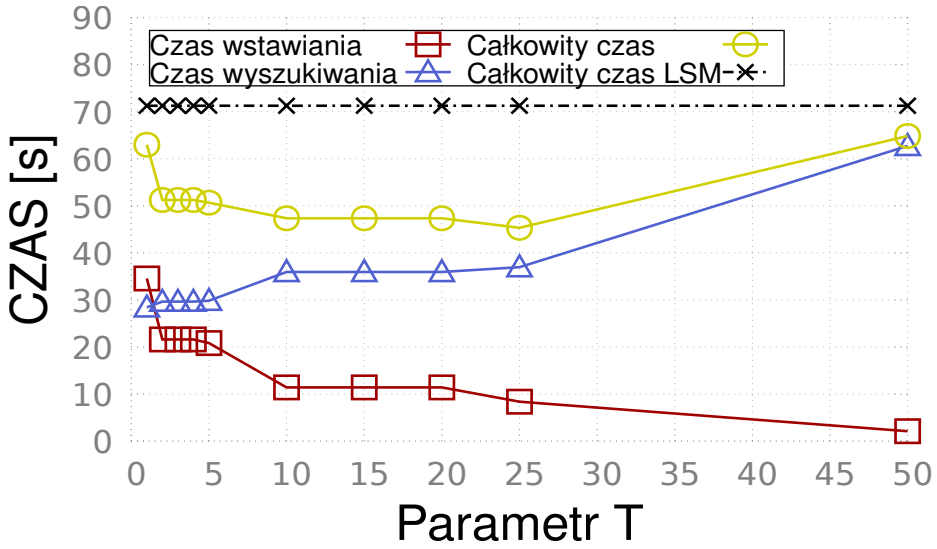
W tej części rozdziału przedstawimy i przeanalizujemy eksperymenty wykonane za pomocą symulatora SIPS, który został dokładnie opisany w rozdziale 4. Do eksperymentów wybrano trzy dyski SSD o odmiennej charakterystyce. Tabela 6.1 zawiera szczegółowe parametry wybranych modeli. Samsung 840 charakteryzuje się bardzo dużą pojemnością strony oraz bloku, odpowiednio 8KB i 512KB. Standardowo pojemność strony to 4KB i taką właśnie posiadają modele Toshiba i Intel. Wybrany model Samsunga posiada bardzo szybki odczyt i zapis sekwencyjny, zbliżony do maksymalnej teoretycznej prędkości interfejsu SATA 3 (750MB/s), która ze względu na dodatkowe dane i bity kontrolne wynosi w praktyce 590MB/s. Mimo bardzo dobrej prędkości sekwencyjnej, to szybkość zapisu losowego jest bardzo wolna na poziomie 182MB/s. Kolejnym wybranym dyskiem jest model Toshiba VX500. Jest to standardowy model dysku SSD z interfejsem SATA 3. Prędkości sekwencyjne zbliżone są do modelu Samsunga, jednak Toshiba posiada o wiele większą prędkość losowego zapisu (267MB/s). Najlepszym dyskiem w tym zestawieniu jest Intel DCP 4511. Ten model charakteryzuje się ogromną prędkością sekwencyjnych operacji (ponad 1GB/s) oraz bardzo szybkim odczytem losowym (1,2GB/s). Największą wadą tego modelu jest bardzo wolny zapis losowy na poziomie tylko 240MB/s. Większość eksperymentów została przeprowadzona na modelu Samsung 840 ponieważ jest to obecnie najpopularniejszy dysk SSD z interfejsem SATA 3. Eksperymenty zostały przeprowadzone na trzech charakterystycznych tabelach (Sklep, Nowe zamówienie i Klient) popularnego zestawu kwerend TPC-C [137], który został dokładnie opisany w rozdziale 4.

Parametr T

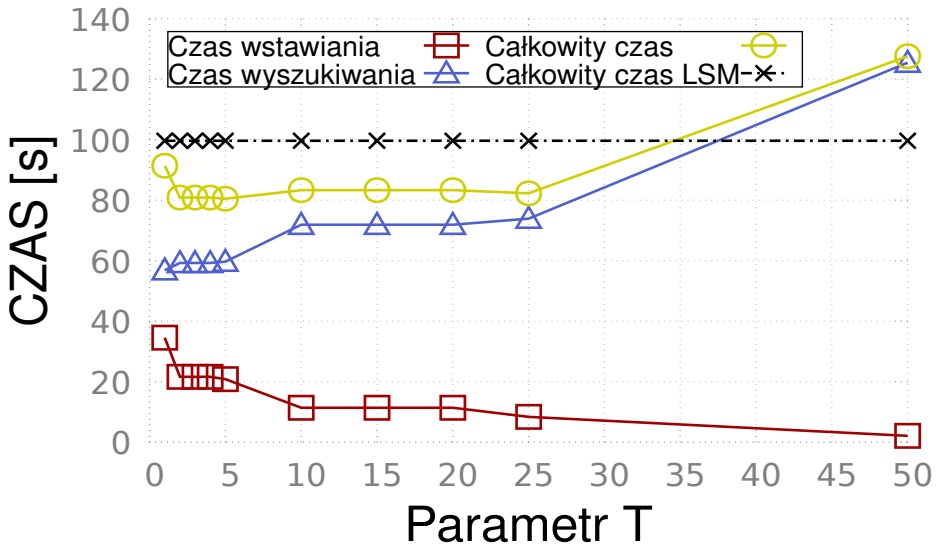
Najważniejszym parametrem indeksu FALSM jest parametr T, który jest kluczowym czynnikiem wyboru poziomu podczas procedury dodawania zbiorczego nowych elementów do indeksu. W tej części eksperymentów zajmiemy się dokładnie obserwacją oraz analizą

wpływu parametru T na szybkość wykonywania się kwerend. Symulator SIPS nie tylko mierzy sumaryczny czas wykonania kwerendy, ale także potrafi zmierzyć dokładnie ile czasu indeks poświęcił na poszczególne operacje takie jak wyszukiwanie rekordów czy ich dodawanie. Skorzystamy z tego, aby lepiej zrozumieć wpływ T na indeks FALSM. Nasza nowa struktura głównie różni się od klasycznego LSM wsparciem operacji dodawania zbiorczego. Samo dodawanie pojedynczych rekordów wykonywane jest dokładnie tak samo na obu indeksach. Z tego powodu zrezygnowaliśmy z przeprowadzania eksperymentów z użyciem podstawowego zestawu kwerend. W zamian za to, na potrzeby naszych obserwacji, stworzyliśmy cztery nowe zestawy, w których używamy tylko dodawania zbiorczego i wyszukiwania zakresu kluczy o selektywności 1%.

- T_A - Wstawiamy 10mln rekordów za pomocą kolejnych zbiorów o losowym rozmiarze z przedziału 50tys. do 100tys. Po każdej procedurze wstawiania przeprowadzamy **20** wyszukiwań elementów z losowego zakresu o selektywności 1%. Zestaw kwerend jest wywoływany dopóki wszystkie 10mln rekordów nie zostanie dodane do indeksu.
- T_B - Wstawiamy 10mln rekordów za pomocą kolejnych zbiorów o losowym rozmiarze z przedziału 50tys. do 100tys. Po każdej procedurze wstawiania przeprowadzamy **40** wyszukiwań elementów z losowego zakresu o selektywności 1%. Zestaw kwerend jest wywoływany dopóki wszystkie 10mln rekordów nie zostanie dodane do indeksu.
- T_C - Wstawiamy 10mln rekordów za pomocą kolejnych zbiorów o losowym rozmiarze z przedziału 50tys. do 100tys. Po każdej procedurze wstawiania przeprowadzamy **100** wyszukiwań elementów z losowego zakresu o selektywności 1%. Zestaw kwerend jest wywoływany dopóki wszystkie 10mln rekordów nie zostanie dodane do indeksu.
- T_D - Wstawiamy 10mln rekordów za pomocą kolejnych zbiorów o losowym rozmiarze z przedziału 50tys. do 100tys. Po każdej procedurze wstawiania przeprowadzamy **250** wyszukiwań elementów z losowego zakresu o selektywności 1%. Zestaw kwerend jest wywoływany dopóki wszystkie 10mln rekordów nie zostanie dodane do indeksu.

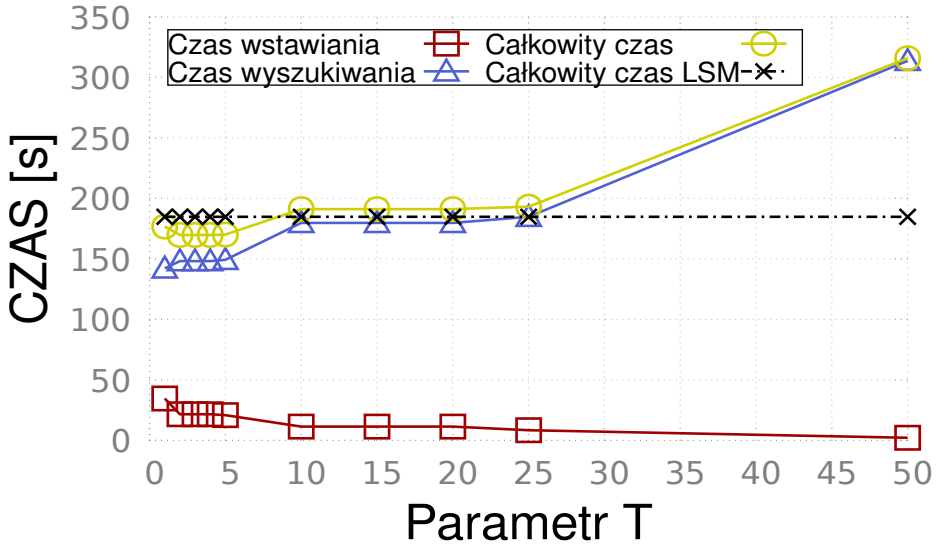


Rysunek 6.3: Czas wykonania zestawu kwerend T_A
Dysk: Samsung 840
Tabela: Sklep (113B)

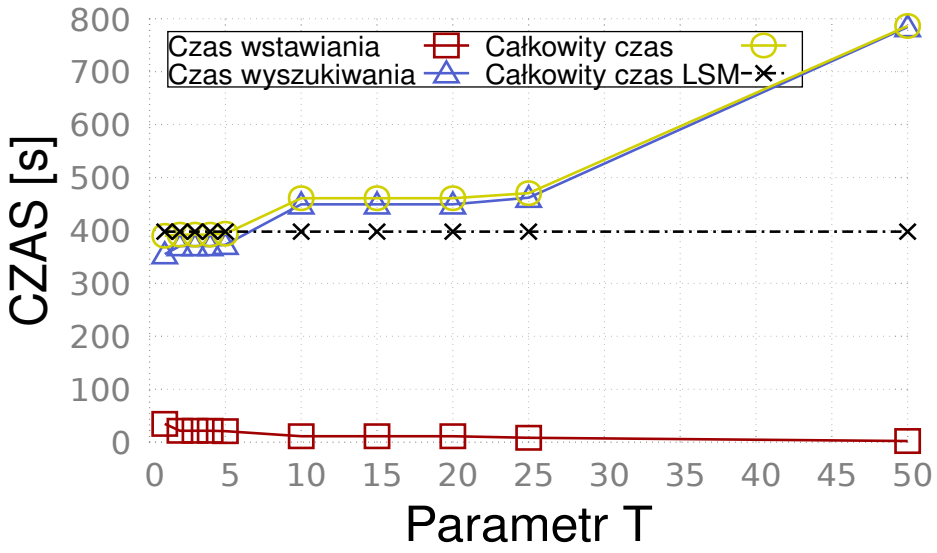


Rysunek 6.4: Czas wykonania zestawu kwerend T_B
Dysk: Samsung 840
Tabela: Sklep (113B)

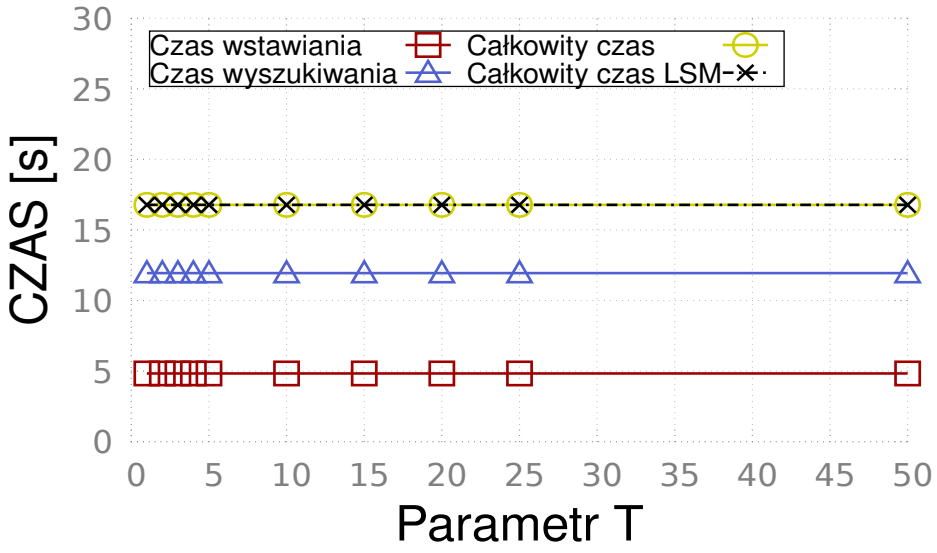
Rysunki: 6.3, 6.4, 6.5, 6.6 przedstawiają wyniki uzyskane na dysku Samsung 840 dla zestawów odpowiednio T_A, T_B, T_C, T_D przeprowadzonych na tabeli TPC-C Sklep o rozmiarze 113B.



Rysunek 6.5: Czas wykonania zestawu kwerend T_C
Dysk: Samsung 840
Tabela: Sklep (113B)



Rysunek 6.6: Czas wykonania zestawu kwerend T_D
Dysk: Samsung 840
Tabela: Sklep (113B)

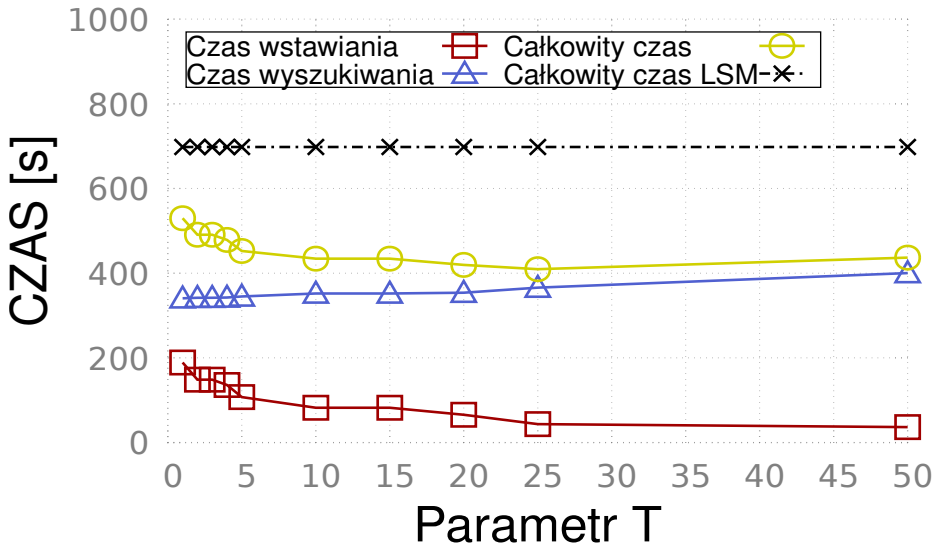


Rysunek 6.7: Czas wykonania zestawu kwerend T_B
Dysk: Samsung 840
Tabela: Nowe zamówienie (20B)

Pierwszą obserwacją jest malejący czas dodawania rekordów oraz rosnący czas wyszukiwania wraz z rosnącym parametrem T. Wynika to ze zmian topologi drzewa FALSM. Im większe T, tym więcej poziomów posiada drzewo, a nowe zbiory danych często lądują do niższych poziomów. To z kolei wpływa na minimalizację liczby reorganizacji oraz zmniejszenie kosztów samej reorganizacji, ponieważ poziomy zawierają mniej danych. Niestety wraz ze wzrostem poziomów rośnie liczba węzłów jakie musimy wczytać podczas wyszukiwania, a to przekłada się na zwiększony koszt tej operacji.

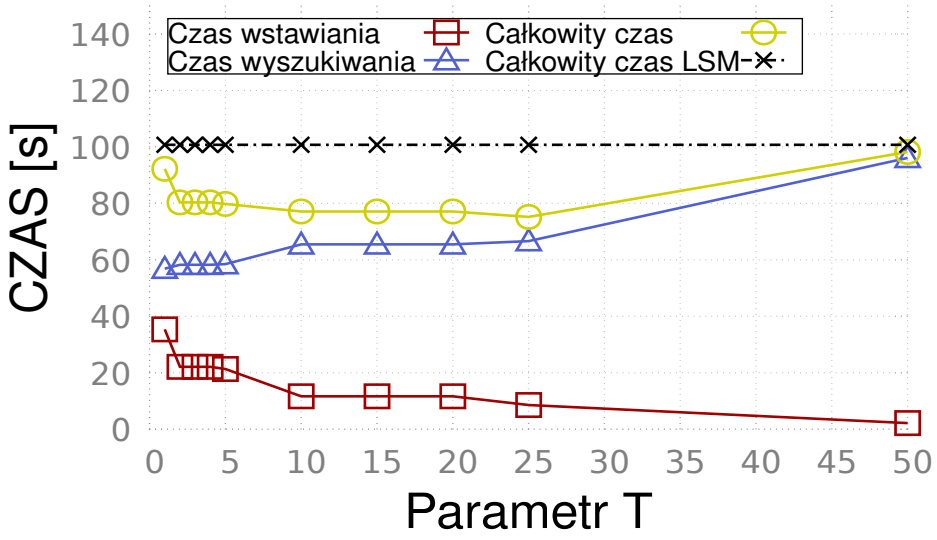
Drugą obserwacją jest wpływ na całkowity czas wykonywania się zestawu kwerend. Gdy liczba wyszukiwań jest większa niż 20 (czyli zestawy T_B, T_C, T_D), to dla dużych wartości T struktura FALSM wykonuje kwerendy o wiele dłużej niż zwykła struktura LSM. Zbyt duża liczba poziomów zmniejszyła czas dodawania elementów, jednak drastycznie wydłużyła czas wyszukiwania, co w efekcie końcowym przełożyło się na gorszą efektywność od indeksu LSM.

Trzecią i najważniejszą obserwacją jest wybór najlepszego T. W przypadku zestawu T_A, najlepszy wynik FALSM osiągnął dla $T = 25$. Jednak już dla pozostałych zestawów przy takiej wartości T, indeks okazał się gorszy od zwykłego LSM. Gdy liczba wyszukiwań jest dość duża, najlepiej ustawić T na wartości 3, 4 lub 5. Przy takich wartościach FALSM zawsze był bardziej efektywny od struktury LSM.

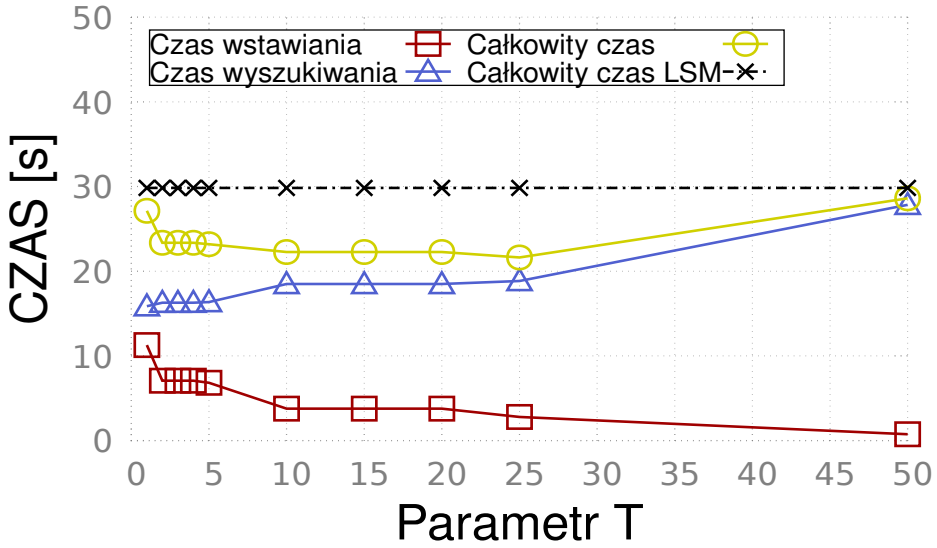


Rysunek 6.8: Czas wykonania zestawu kwerend T_B
Dysk: Samsung 840
Tabela: Klient (719B)

W kolejnej serii eksperymentów sprawdziliśmy wpływ rozmiaru tabeli na wybór najlepszej wartości T. Eksperymenty zostały przeprowadzone na tabelach: Nowe Zamówienie (rysunek 6.7), Sklep (rysunek 6.4) oraz Klient (rysunek 6.8) przy użyciu dysku Samsung 840. Parametr T nie miał żadnego wpływu na czas wykonania się operacji dla małej tabeli Nowe zamówienie o rozmiarze 20B. Wynika to z faktu, że większość danych trafiła do bufora i nie została przepisana na docelowy poziom (pamięć dysku SSD). Ponieważ bufor dla obu indeksów: FALSM i LSM ustawiony jest na tę samą wielkość 4MB, to czas wykonania się kwerend dla tych indeksów był dokładnie taki sam. Analizując wyniki z eksperymentu przeprowadzonego na dużej tabeli Klient o rozmiarze 719B, możemy stwierdzić, że im większy rozmiar rekordu tym wybór parametru T jest ważniejszy. Wynika to z faktu, iż im większy rekord tym więcej czasu kosztuje jego przepisanie podczas reorganizacji struktury. Zatem dobry wybór poziomu docelowego potrafi drastycznie obniżyć czas wstawiania danych. W tym przypadku najlepszy całkowity czas został osiągnięty dla $T = 25$. Wynik ten był o 8% lepszy od najlepszego poprzednio wybranego $T = 5$.



Rysunek 6.9: Czas wykonania zestawu kwerend T_B
Dysk: Toshiba VX500
Tabela: Sklep (113B)



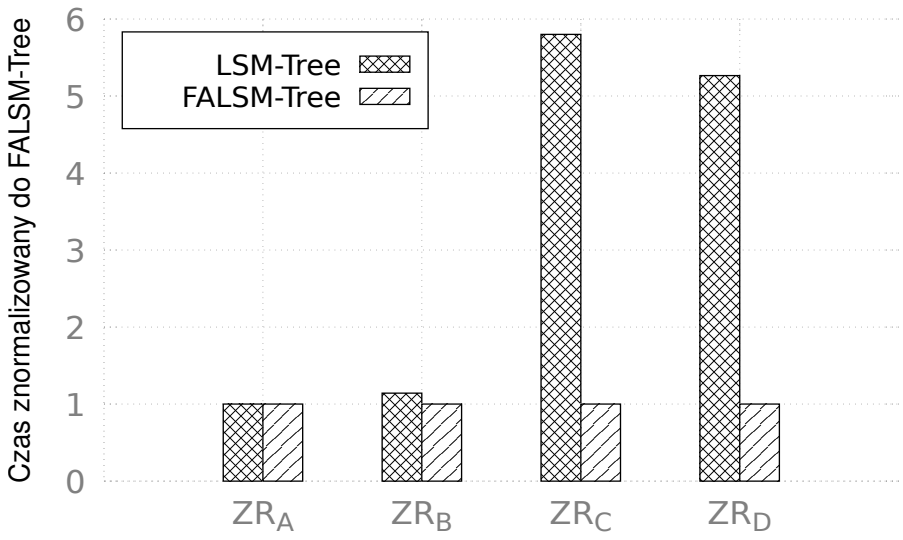
Rysunek 6.10: Czas wykonania zestawu kwerend T_B
Dysk: Intel DCP4511
Tabela: Sklep (113B)

W trzeciej serii eksperymentów zbadaliśmy wpływ charakterystyki dysków na efektywność struktury FALSM przy różnych parametrach T. Ponownie wybraliśmy standardową tabelę TPC-C Sklep. Jednak tym razem zamieściliśmy wyniki tylko dla zestawu T_B, ponieważ pozostałe zestawy uzyskały ten sam trend. Rysunki: 6.4, 6.9, 6.10 przedstawiają wyniki odpowiednio dla dysków Samsung, Toshiba i Intel. Porównując ze sobą wszystkie trzy dyski, widzimy że szybkość odczytu jak i zapisu nie ma wpływu na wybór najlepszego parametru T. Okazało się, że głównie to rozmiar strony i bloku ma wpływ na osiągi FALSM. Rozmiar strony jak i parametr T ma wpływ na topologię drzewa: im

mniejsza strona, tym więcej ich jest w jednym węźle. To oznacza, że musimy zapisać lub odczytać więcej stron podczas obsługi pojedynczego węzła. Z obserwacji wynika, że na dyskach o mniejszych stronach optymalne T znajduje się w okolicach wartości 20-25. Jednak warto zaznaczyć, że czas całkowity wykonania się kwerend nie odbiega więcej niż 3% od poprzedniej optymalnej wartości 5. Możemy zatem uznać, że $T = 5$ jest wartością, która sprawdza się bardzo dobrze dla dowolnego dysku jak i dowolnego rozmiaru tabeli. Użyjemy tego faktu w kolejnych eksperymentach, w których ustawimy wartość T właśnie na 5.

Rozszerzony zestaw kwerend

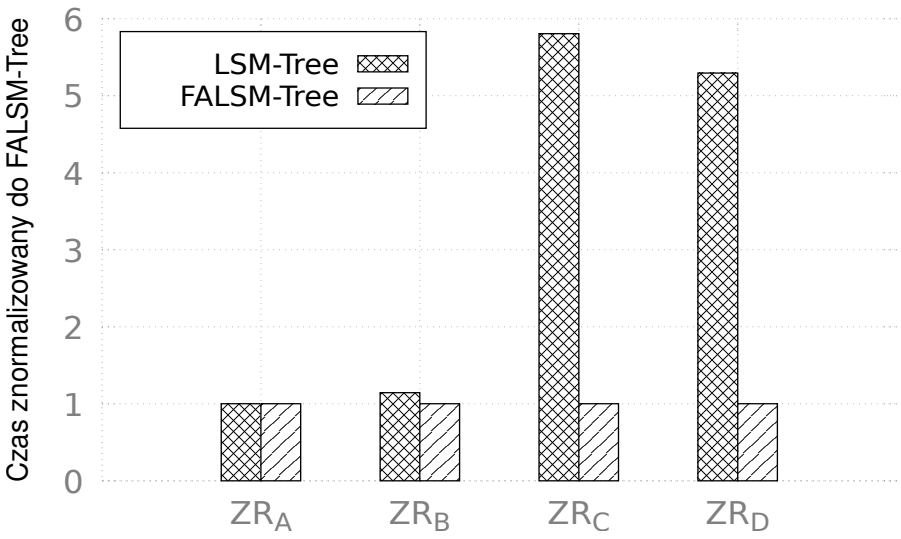
Tak jak wspominaliśmy wcześniej zestaw rozszerzony powstał na podstawie kwerend z TPC-C. Celem tego zestawu jest symulacja hurtowni danych, w której operacje są buforowane i wykonywane na wielu rekordach. Oznacza to, że każde dodawanie rekordów użyje dodawania zbiorczego na strukturze FALSM oraz zwykłego dodawania do bufora na strukturze LSM.



Rysunek 6.11: Znormalizowany czas
Dysk: Samsung 840
Tabela: Sklep (113B)

W tym zestawie kwerend używamy wyszukiwania całego zakresu rekordów zamiast punktowego wyszukiwania. Każde wyszukiwanie ma ustawioną selektywność na poziomie 1% całej tabeli. W każdej serii zestawu kwerend najpierw wykonywane jest dodawanie rekordów, później wyszukiwanie, a na końcu usuwanie rekordów. Przygotowano cztery różne zestawy kwerend, które dokładnie opisano w rozdziale **4**.

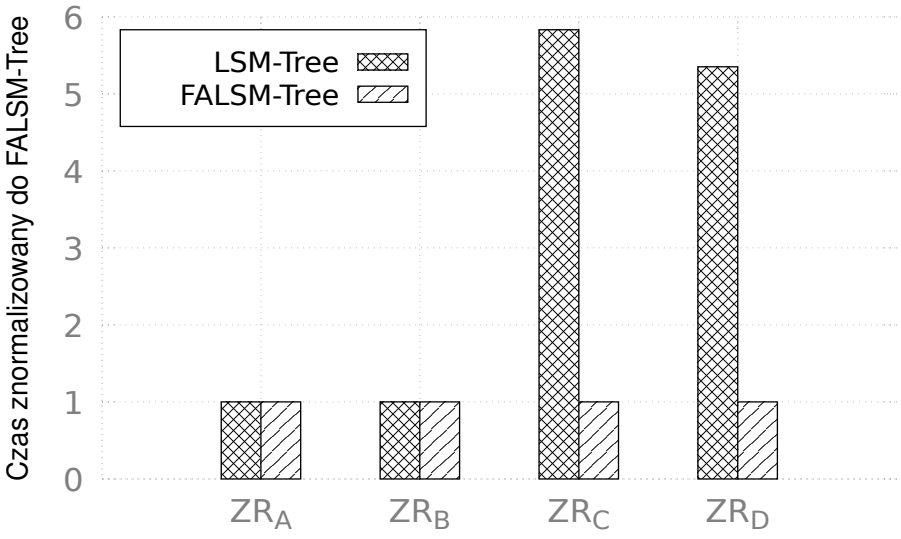
1. ZR_A - 100 serii, każda seria zawiera operacje dodawania 5 rekordów, 10 wyszukiwań o selektywności 1% oraz usuwanie 5 rekordów,
2. ZR_B - 5 serii, każda seria zawiera operacje dodawania 100000 rekordów, 5 wyszukiwań o selektywności 1% oraz usuwanie 100000 rekordów,
3. ZR_C - 10 serii, każda seria zawiera operacje dodawania 10000000 rekordów, 20 wyszukiwań o selektywności 1% oraz usuwanie 1000000 rekordów,



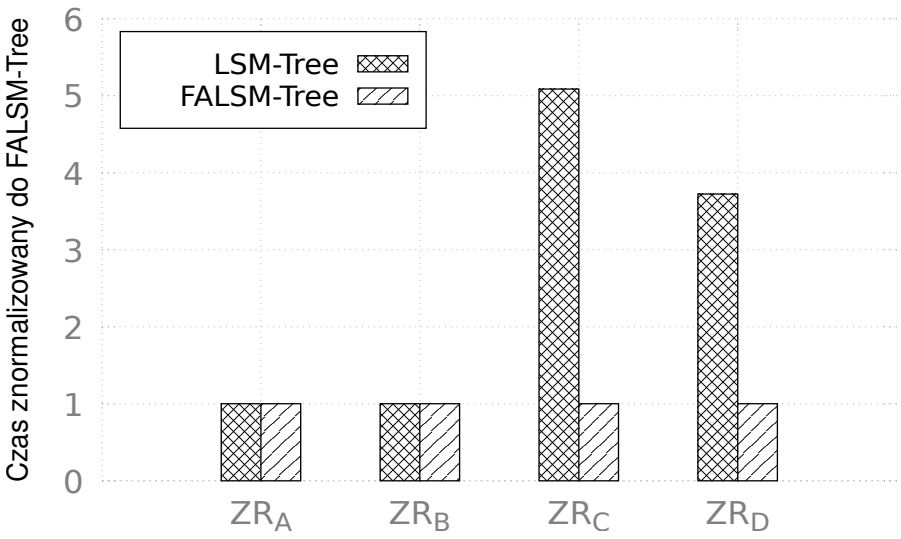
Rysunek 6.12: Znormalizowany czas
Dysk: Toshiba VX500
Tabela: Sklep (113B)

4. ZR_D - 10 serii, każda seria zawiera operacje dodawania 1000000 rekordów, 10 wyszukiwań o selektywności 1% oraz usuwanie 10000 rekordów.

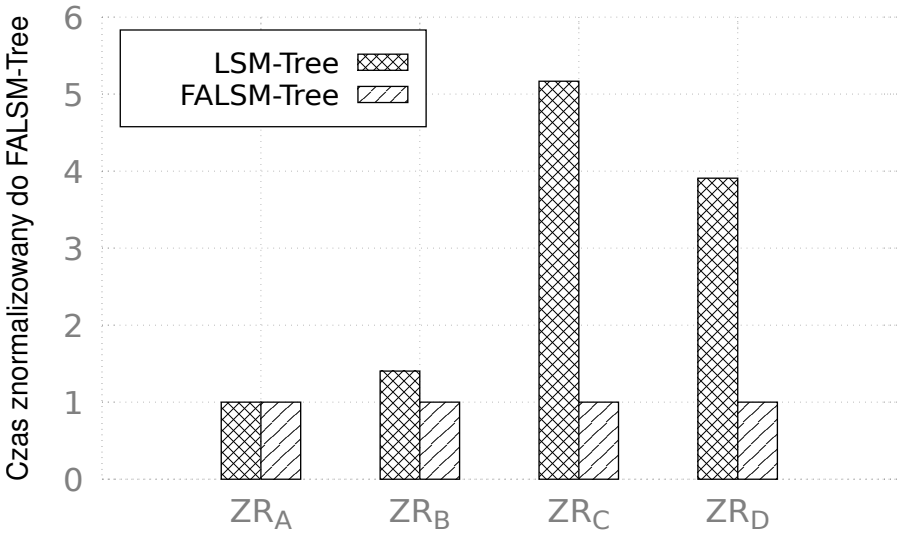
Zaproponowany zestaw kwerend wykonujemy na indeksie zawierającym już wiele danych. W naszym przypadku, najpierw wstawiliśmy do indeksów 10mln danych a następnie wykonaliśmy kwerendy i zmierzyliśmy czas jak i wykorzystanie pamięci dysku SSD za pomocą odpowiednich liczników dostarczanych przez symulator SIPS. Przypomnijmy, że w tych eksperymentach FALSM ma ustawiony parametr T na stałą wartość 5.



Rysunek 6.13: Znormalizowany czas
Dysk: Intel DCP4511
Tabela: Sklep (113B)



Rysunek 6.14: Znormalizowany czas
Dysk: Samsung 840
Tabela: Nowe zamówienie (20B)



Rysunek 6.15: Znormalizowany czas
Dysk: Samsung 840
Tabela: Klient (719B)

Analizując wyniki wszystkich eksperymentów przedstawionych na rysunkach: 6.11, 6.12, 6.13, 6.14, 6.15, możemy stwierdzić, że ani charakterystyka dysku ani rozmiar rekordu nie wpływa na trend i przewagę FALSM nad LSM. Niezależnie od modelu pamięci czy tabeli, LSM uzyskał taki sam czas jak FALSM dla zestawu ZR_A. To spowodowane jest bardzo małą liczbą nowych rekordów, które prawie w całości trafiły do bufora i nie zostały przepisane na dysk. Dla zestawu ZR_B drzewo FALSM uzyskało wynik o 12% lepszy od struktury LSM. Z kolei w przypadku zestawów ZR_C i ZR_D FALSM okazał się lepszy ponad 5 krotnie od klasycznego indeksu. Jak widać, gdy w zestawie kwerend

występuje dużo zbiorczych dodawań, to mimo iż czas wyszukiwania jest zwiększony przez obecność węzłów dopisanych, całkowity czas zestawu jest o wiele lepszy na FALSM. Nasza nowa struktura uzyskuje lepszy czas poprzez minimalizację liczby zapisów na dysk, a więc całkowite zużycie pamięci (ang. wearout) też jest mniejsze. Tabela 6.2 przedstawia całkowitą liczbę zapisanych bajtów podczas przeprowadzania eksperymentów. Jak widać mamy dokładnie ten sam trend co na wykresach omawianych wcześniej. FALSM zapisuje nawet 5 krotnie mniej bajtów na dysku SSD niż indeks LSM.

	Nowe zamówienie		**Sklep**		**Klient**	
Zestaw	**LSM**	**FALSM**	**LSM**	**FALSM**	**LSM**	**FALSM**
$\boldsymbol{ZR_A}$	0	0	0	0	33KB	33KB
$\boldsymbol{ZR_B}$	58KB	58KB	416KB	349KB	4,4MB	2,9MB
$\boldsymbol{ZR_C}$	24MB	4,4MB	170MB	28MB	1,3GB	250MB
$\boldsymbol{ZR_D}$	2MB	500KB	14,2MB	2,4MB	110MB	27MB

Tabela 6.2: Zużycie pamięci dysku SSD na różnych tabelach
Dysk: Samsung 840

	Nowe zamówienie		**Sklep**		**Klient**	
Zestaw	**LSM**	**FALSM**	**LSM**	**FALSM**	**LSM**	**FALSM**
$\boldsymbol{ZR_A}$	3,9s	3,9s	20s	20s	122s	122s
$\boldsymbol{ZR_B}$	0,4s	0,4s	2s	1,7s	18s	13s
$\boldsymbol{ZR_C}$	87s	17s	10min	1,8min	80min	16min
$\boldsymbol{ZR_D}$	7,7s	2s	52s	10s	7min	1,5min

Tabela 6.3: Czas wykonania kwerend na różnych tabelach
Dysk: Samsung 840

	Samsung 840		**Toshiba VX500**		**Intel DCP4511**	
Zestaw	**LSM**	**FALSM**	**LSM**	**FALSM**	**LSM**	**FALSM**
$\boldsymbol{ZR_A}$	20s	20s	20,2s	20,2s	5,6s	5,6s
$\boldsymbol{ZR_B}$	2s	1,7s	2s	1,7s	0,2s	0,2s
$\boldsymbol{ZR_C}$	10min	1,8min	10,5min	1,9min	198s	34s
$\boldsymbol{ZR_D}$	52s	10s	53s	10s	16,8s	3,2s

Tabela 6.4: Czas wykonania kwerend na różnych dyskach SSD
Tabela: Sklep (113B)

W poprzednich eksperymentach, gdy sprawdzaliśmy wpływ parametru T na efektywność struktury FALSM, dowiedzieliśmy się, że rozmiar strony dysku SSD ma wpływ na wybór T. Analizując wyniki zestawu umieszczone w tabeli 6.4 widzimy, że przy ustalonym już T, rozmiar strony i bloku nie mają wpływu na czas wykonania się całego zestawu. Oczywiście im lepsze parametry dysku, szybszy czas zapisu i odczytu tym szybciej uda się obsłużyć wszystkie kwerendy. Stąd zawsze najmocniejszy dysk w naszym zestawieniu Intel DCP4511 osiąga najlepsze czasy we wszystkich zestawach. Wynika to głównie z faktu, iż ani LSM ani FALSM nie używają losowych zapisów. Ich procedury dodawania nowych elementów działają w taki sposób, że zawsze kontroler dysku może działać w trybie sekwencyjnym. Dlatego mimo bardzo wolnego zapisu losowego Intel uzyskuje najkrótszy

czas wykonania kwerend. Podobne wnioski uzyskujemy ze sprawdzenia efektywności indeksów na różnych tabelach. Mimo iż rozmiar rekordu wpływał na optymalne T, to przy ustalonym parametrze, nie ma on wpływu na różnice pomiędzy LSM a FALSM. Tak więc zostaje zachowany trend dokładnie taki sam jak w przypadku poprzednich eksperymentów. Oznacza to, iż mimo że mniejsze rekordy potrzebują mniej czasu na ich zapisanie, to nadal FALSM jest 5 krotnie szybszy od indeksu LSM w zestawie ZR_C. Jedyna zmiana zaistniała w zestawie ZR_D. Przy użyciu tabeli Nowe zamówienie o rozmiarze 20B, FALSM uzyskał wynik około 4 razy lepszy niż LSM, a na tabeli Sklep oraz Klient uzyskał wyniki około 5 razy lepsze od klasycznej struktury.

6.4 Indeksowanie kolumnowe

Chociaż to indeksowanie wierszowe dalej jest domyślnym sposobem zapisu bazy danych w wielu systemach, to zapis kolumnowy w ostatnich latach znacząco zyskał na popularności. Wynika to z faktu, że większość zapytań do baz danych nie potrzebuje informacji ze wszystkich kolumn. Zatem w wierszowej implementacji marnujemy odczyty z dysku na wczytanie całego wiersza i wyłuskanie tylko potrzebnych atrybutów. W podejściu kolumnowym na jednej stronie w pamięci przetrzymywane są wartości pojedynczej kolumny dla kilku rekordów, dzięki czemu podczas wyszukiwania wczytujemy tylko potrzebne informacje. Niestety kolumnowe podejście jest wolniejsze podczas zapisywania nowych rekordów, ponieważ musimy zapisać podzielony rekord na kolumny w kilku miejscach na dysku [10], [11], [12].

Historycznie, za pierwszy silnik w pełni wspierający kolumnowy zapis tabeli uznaje się C-Store [31]. Opisany on został dokładnie w rozdziale **2**. Przypomnijmy jednak najważniejsze aspekty tego systemu. C-Store zapisywał tabelę w dwóch osobnych sekcjach: sekcji zorientowanej na szybkie odczyty, która była zapisana kolumnowo, oraz sekcji zoptymalizowanej pod częste zapisy i aktualizacje danych. Druga sekcja zapisana była wierszowo aby umożliwić szybkie wstawianie i usuwanie rekordów. C-Store sam decydował kiedy przenieść dane z jednej sekcji do drugiej.

Indeksowanie kolumnowe w ostatnich latach tak mocno zyskało na popularności, że większość nowoczesnych silników baz danych wspiera ten sposób zapisu tabeli. Należą do nich między innymi: MonetDB [45], PostGreSQL [46], VectorWise [47], Druid [48] i eXtremeDB [49]. Najpopularniejszą strukturą danych używaną obecnie do indeksowania rekordów zapisanych kolumnowo jest PDT [50] (ang. Positional Delta Tree). Indeks ten wykorzystywany jest w bazach danych zapisanych w pamięci operacyjnej komputera (ang. in-memory database) oraz na dyskach twardych HDD. Niestety ze względu na sposób działania nie zaleca się używania tej struktury gdy indeks zapisany jest na dysku SSD. Nowoczesne pamięci takie jak flash, SSD czy PCM posiadają odmienną charakterystykę. Zapis na tych pamięciach jest kilkukrotnie wolniejszy niż odczyt, a więc klasyczne struktury danych nie osiągają dobrych rezultatów w ułożeniu kolumnowym. Algorytm musi nie tylko spełniać wszystkie wymogi kolumnowego zapisu takie jak posiadanie szybkiej operacji scalania, ale musi także wykorzystywać techniki przyspieszające zapis danych adekwatne do używanych pamięci. Dobrymi przykładami jest system PAX [51], który polega na wirtualnym podziale strony dysku SSD na mini strony. Każda mini strona zawiera tylko dane jednej kolumny. Mamy zatem ułożenie kolumnowe kilku rekordów w obrębie jednej fizycznej strony. Taki mechanizm nie zmniejsza liczby bajtów wczytywanych z dysku, gdyż najmniejszą jednostką na której pracujemy jest strona, ale przyspiesza

pracę procesora ze względu na lepsze wykorzystanie pamięci podręcznej procesora (ang. cache). Kolejnym przykładem jest struktura FBDSM [52] (ang. Flash-based Decomposition Storage Model), która dostosowała oryginalny pomysł DSM [30] do pamięci flash. Zamiast jednej tabeli, są dwie. Pierwsza to tabela główna PT, w której zapisywane są dane dodane do tabeli, które nie zostały usunięte ani zmienione. Druga tabela to tabela LT, która zawiera wszystkie zmiany na tabeli PT a więc: usunięcia rekordów oraz zmianę wartości. Dzięki temu nadpisywanie danych jest odroczone w czasie oraz realizowane później za pomocą paczek danych, co jest o wiele szybsze w przypadku pamięci flash. Struktura ta jednak nie rozwiązuje głównego problemu, czyli braku relacji porządku. Ponadto tabela nie jest posortowana po kluczu, a po czasie, aby zachować pozycje atrybutów dla danego rekordu, co umożliwia szybkie scalanie kolumn w jeden pełny rekord.

6.5 Columned FD-Tree

Ponieważ, żadna z wcześniejszych wymienionych struktur nie wspiera w pełni jednocześnie kolumnowego zapisu, szybkiego wyszukiwania po kluczu oraz wsparcia dla dysków SSD, zaproponowałem nową strukturę danych, która spełnia wszystkie te trzy wymogi. Struktura CF-Tree (ang. Columned FD-Tree) [3] to zbiór drzew [25], przy czym jeden z nich KCF (ang. Key Column FD Tree) przechowuje atrybut klucza, a pozostałe nazwane NCF (ang. Non-key Column FD Tree) przechowują po jednym atrybucie, który kluczem nie jest. Za pomocą specjalnego algorytmu scalania atrybutów w rekord stworzonego na potrzeby tej struktury, w każdym momencie możemy połączyć wszystkie kolumny i otrzymać pełny rekord. Struktura CF-Tree charakteryzuje się bardzo szybkim wstawianiem nowych elementów, nie odbiegającym znacząco od szybkości oryginalnego drzewa FD, który zapisywał dane wierszowo, szybkim wyszukiwaniem z zakresu, oraz wolniejszym wyszukiwaniem punktowym. Dzięki swojej strukturze jak i algorytmowi scalania atrybutów w rekord, szybkość wyszukiwania zależy od liczby kolumn jakie użytkownik chciał wczytać oraz od wielkości samego atrybutu. Podczas eksperymentów mogliśmy zaobserwować znaczącą przewagę struktury CFT nad FBDSM w każdych warunkach, oraz przewagę nad wierszowym FD, gdy liczba wczytanych kolumn była mniejsza niż liczba atrybutów, na który składa się cały rekord.

6.5.1 Struktura CF-Tree

Nasza nowa struktura CF-Tree składa się z tylu drzew FD ile atrybutów ma rekord w tabeli. Aby zapewnić szybki zapis i odczyt na dyskach SSD, pojedyncze drzewo w tej strukturze składa się z kilku poziomów. Na samej górze, na poziomie L_0, mamy bardzo małe drzewo B+ o wielkości zaledwie kilku stron dysku SSD. Każdy poziom według zasady 2 (opisanej później w tym rozdziale) musi posiadać tyle samo danych. Ta zasada dotyczy również bufora. Z tego wynika, że rozmiary buforów dla różnych drzew w zbiorze CFT nie muszą być równe. Ich wielkości dostosowywane są w taki sposób aby każde drzewo mogło pomieścić w buforze tyle samo danych co kolumna klucza na KCF. Ponieważ ten poziom jest bardzo mały, z łatwością zmieści się w pamięci RAM. Poziom L_0 służy do buforowania danych zanim trafią do dalszych poziomów (L_i, $i > 0$), które są zapisane na dysku SSD. Aby uzyskać strukturę drzewa, liście w drzewie B+ zawierają wskaźniki do L_1, czyli pierwszego poziomu zapisanego na dysku. Ponadto, każdy kolejny poziom zawiera zbiór wskaźników do odpowiednich stron pamięci poziomu niżej. Dzięki takiej strukturze możemy w łatwy sposób przejść po naszym drzewie od góry do dołu, aby wyszukać rekord

o wskazanym kluczu. Sama procedura wyszukiwania ze względu na kolumnowe ułożenie danych jest dość skomplikowana, ponieważ wymaga użycia nowego algorytmu scalania atrybutów w rekord. Procedura wyszukiwania szczegółowo opisana jest w dalszej części tego rozdziału.

Każdy z poziomów posiada swoją maksymalną pojemność wyrażoną w liczbie stron dysku SSD. Każda operacja alokacji pamięci odbywa się poprzez alokację pełnej strony. Aby zminimalizować koszty reorganizacji struktury, podobnie jak w przypadku drzewa B+ czy LSM, wprowadziliśmy współczynnik k. Każdy kolejny poziom, zawiera k razy więcej bloków ($|L_{i+1}| = |L_i| \cdot k$). Podczas eksperymentów zauważyliśmy, że współczynnik k powinno dobrać się odpowiednio do rodzaju kwerendy. Im większe k tym drzewo jest niższe, czyli koszt wyszukiwania jest mniejszy, ale koszt reorganizacji jest o wiele większy. Z tego względu, albo powinno dobierać się ten współczynnik pod odpowiedni zestaw kwerend, albo powinno ustalać się neutralną wartość. Z przeprowadzonych eksperymentów wynika, że neutralne wartości k są z przedziału $< 5; 10 >$.
W naszej nowej strukturze wyróżniamy 4 rodzaje danych:

1. Normalna dana (ang. normal entry) - jest to pojedynczy atrybut rekordu. Dla drzewa KCF będzie to klucz rekordu, dla pozostałych drzew NCF będą to zwykłe wartości danej kolumny, która kluczem nie jest.

2. Usunięta dana (ang. deleted entry) - aby zminimalizować liczbę zapisów na dysku SSD, skorzystaliśmy z leniwego mechanizmu usuwania dostępnego w drzewie FD. Zamiast usuwać rekord natychmiastowo, co doprowadziłoby do reorganizacji wszystkich drzew, wstawiamy nowy rekord oznaczony jako d_{key} o takim samym kluczu jak rekord, który chcemy usunąć. Tak więc dzielimy rekord na atrybuty i wstawiamy wartości poszczególnych atrybutów do odpowiadających im drzewom oznaczając przy tym wartość jako „do usunięcia”. Gdy dane podczas nieuniknionej migracji poziomów w dół spotkają się, są usuwane ze struktury. Operacja ta dokładnie opisana jest w kolejnych sekcjach tego rozdziału.

3. Zewnętrzny wskaźnik (ang. external fence) - ten element struktury zawiera trzy pola. Pierwszym z nich jest identyfikator strony, na którą wskazuje (PID) z poziomu niżej. Drugim polem jest wartość pierwszego (czyli najmniejszego) klucza strony, na którą wskazuje. Warto zwrócić uwagę, że wskaźnik zawiera wartość klucza na każdym drzewie KCF i NCF. Ponieważ operacja scalania poziomów wykonywana jest jednocześnie na wszystkich drzewach w obrębie CFT, to istnieje możliwość skopiowania odpowiedniej wartości klucza z KCF podczas tworzenia wskaźnika w drzewach NCF. Ostatnim polem jest tablica $numEntriesBeforePageArray$, która zawiera wartości $numEntriesBeforePage$ kolejnych stron na ścieżce od obecnej strony do liścia w drzewie z kluczem (lub wartością inne atrybutu skorelowanego z kluczem) równym wartości klucza dane wskaźnika. Podczas operacji scalania poziomów, dla każdej strony poziomu L_{i+1} tworzony jest nowy wskaźnik do tej strony i umieszczany jest w poziomie nad nim, czyli w L_i. Dzięki temu struktura drzewa jest zachowana po każdej reorganizacji.

4. Wewnętrzny wskaźnik (ang. internal fence) - wskaźnik ten zawiera tylko 2 pola. Pierwszym z nich jest wskaźnik na stronę z poziomu niższego (PID). Drugim polem jest wartość klucza. Tym razem nie jest to klucz strony, na którą wskazuje, ale klucz najmniejszego klucza na poziomie, na którym znajduje się wewnętrzny wskaźnik. Tak samo jak w przypadku zewnętrznego wskaźnika, wartość klucza brana jest

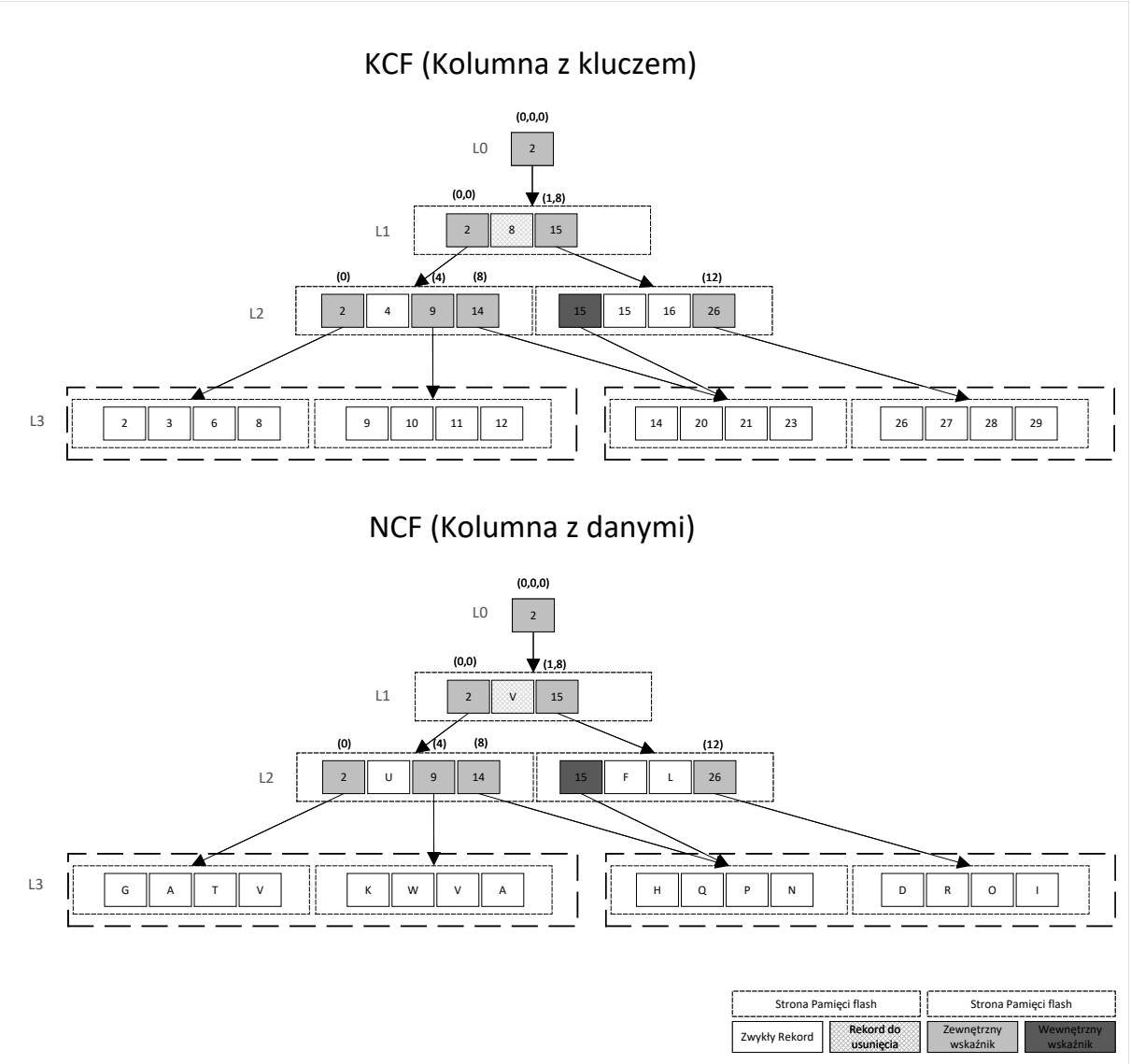
z drzewa KCF niezależnie czy wskaźnik znajduje się w drzewie KCF czy NCF. Każda strona jako pierwszy element, musi posiadać wskaźnik aby utrzymać strukturę drzewa i wszystkie własności struktury FD jak i CFT. Jeśli strona nie zawiera zewnętrznego wskaźnika, wówczas tworzony jest wewnętrzny wskaźnik, który wskazuje na tę samą stronę co poprzedni wskaźnik danego poziomu.

Struktura CFT posiada 7 głównych własności, które nie mogą zostać zaburzone podczas działania systemu.

1. Liczba danych w każdym FD-Tree jest taka sama,
2. Liczba danych w każdym FD-Tree na danym poziomie jest taka sama,
3. Liczba zewnętrznych wskaźników (ang. external fences) na poziomie L_i jest równa liczbie stron an poziomie L_{i+1},
4. Jeśli poziom nie jest liściem, to pierwszą daną (ang. entry) na stronie musi być wskaźnik,
5. Wszystkie wskaźniki na każdym drzewie w zbiorze CF-Tree posiadają wartość pola *key* równą wartości klucza braną z KCF,
6. Każda strona dysku SSD p na każdym poziomie posiada pole $p.numEntriesBeforePage$, zawierające liczbę danych zapisanych na tym poziomie przed stroną p,
7. Każdy zewnętrzny wskaźnik ptr posiada tablicę $numEntriesBeforePageArray$ składającą się z $ptr.page.numEntriesBeforePage$, $ptr'.page.numEntriesBeforePage$, $ptr''.page.numEntriesBeforePage$ ..., gdzie ptr', ptr'' ... to skrajnie lewe wskaźniki na drodze od strony na której zapisany jest ptr aż do liścia z wartością $ptr.key$.

Rysunek 6.16 przedstawia przykładową strukturę CFT dla rekordu o 2 atrybutach. Pierwszym atrybutem jest liczba - klucz rekordu, drugim atrybutem jest pojedyncza litera. Dla uproszczenia przykładu zakładamy, że oba atrybuty mają taki sam rozmiar. Strona dysku SSD jest w stanie pomieścić 4 wartości. Parametr k jest ustawiony na wartość 2. Tak więc każdy kolejny poziom ma 2 razy większą pojemność. Ze względu na to, iż rekord składa się z tylko 2 atrybutów, CF-Tree zawiera 2 drzewa: KCF i jedno NCF. Warto zauważyć, że o ile wartości w drzewie KCF nie mogą się powtarzać, ponieważ klucz jest unikalny, to wartości w NCF mogą się powtarzać, ponieważ nie są kluczem rekordu.
Jak już wiemy, w drzewie wyróżniamy 4 rodzaje rekordów:

1. Normalne - dane, które użytkownik dodał do bazy (na rysunku np $< 2,G >$, $< K,9 >$.
2. Do usunięcia - dane, które użytkownik chce usunąć, ale ze względu na system leniwego usuwania, zamiast przeprowadzać nadmiarową reorganizację drzewa, dodajemy rekord do usunięcia (na rysunku mamy $< 8,V >$ na poziomie L_3, oraz $< 8,V >$ do usunięcia na poziomie L_1)
3. wskaźnik wewnętrzny - na rysunku jest to $< 15 >$ na poziomie L_2. Warto zauważyć, że wartość tego wskaźnika jest taka sama drzewie KCF jak i NCF.
4. wskaźnik zewnętrzny - wskaźnik na poziom niżej, który prócz wartości i wskaźnika na stronę zawiera także tablicę $numEntriesBeforePageArray$. Tablica ta zawiera



Rysunek 6.16: Przykładowa struktura CF-Tree

liczbę danych, jaka jest zapisana przed stronami na ścieżce od wskaźnika do liścia. Przykładowo wskaźnik $< 15, (1,8) >$ wskazuje na stronę $P2$ poziomu L_1. Przed tą stroną mamy tylko jedną normalną daną $< 4,U >$, później nasza ścieżka prowadzi do $P3$ na poziomie L_2. Przed tą stroną mamy aż 8 normalnych wartości, a więc tablica ma wartości (1,8).

Taka struktura drzew pozwala nam w bardzo łatwy sposób znaleźć odpowiednie atrybuty i złożyć wszystkie kolumny w pełny rekord. Przykładowo chcemy znaleźć rekord o kluczu 10. Najpierw szukamy klucza 10 w drzewie KCF. Ta procedura nie różni się od zwykłego wyszukiwania w drzewie FD. Początkowo mamy w buforze tylko wskaźnik 2. Zatem odwiedzamy stronę $P1$. Tutaj szukamy wskaźnika o największej wartości mniejszej niż 10. Wskaźnik 2 spełnia ten warunek. Wczytujemy zatem kolejną stronę - $P1$. Tym razem to wskaźnik o wartości 9 jest największym wskaźnikiem, zatem odczytujemy kolejną stronę i znajduje wartość 10. Wiemy zatem, że nasz rekord został zapisany na poziomie L_3, na stronie $P2$ na pozycji 6. Mając wszystkie informacje, możemy wykonać algorytm scalania atrybutów. Wykorzystując tablicę $numEntriesBeforePageArray$

musimy znaleźć atrybut na pozycji 6. Zatem odczytujemy bufor drzewa NCF. Mamy tutaj tylko 1 wskaźnik, a więc wczytujemy stronę $P1$. Teraz szukamy takiego wskaźnika, który ma ostatnią wartość tablicy maksymalną ale nie większą niż 6 (ostatnią wartość, ponieważ szukamy danej w ostatnim poziomie). Wskaźnik 2 spełnia ten warunek, ponieważ wartość $numEntriesBeforePageArray[2] = 0$, a dla kolejnego wskaźnika 15 wartość $numEntriesBeforePageArray[2] = 8$. Odczytujemy kolejną stronę zawierającą (2,U,9,14). Kolejny raz musimy znaleźć odpowiedni wskaźnik na podstawie tablicy $numEntriesBeforePageArray$. Wskaźnik o wartości 9 spełnia nasz warunek wyszukiwania, a więc odczytujemy stronę $P2$. Wiemy, że ta strona ma przed sobą 4 wartości, zatem szukany atrybut znajduje się na pozycji 2 obecnej strony. Zwracamy zatem wartość W i kończymy algorytm.

6.5.2 Procedury indeksu CF-Tree

Wstawianie

Ponieważ CF-Tree to zbiór drzew FD, samo wstawianie odbywa się tak samo jak w drzewie FD z tą tylko różnicą, że rekord jaki otrzymujemy od użytkownika dzielimy na pojedyncze atrybuty, klucz rekordu wstawiamy do KCF-Tree, a pozostałe kolumny do NCF-Tree. Pamiętajmy, że każde z drzew FD w zbiorze posiada swój bufor w pamięci RAM oraz, że każde drzewo musi posiadać tyle samo danych na każdym poziomie, również w buforze. Oznacza to, że rozmiary buforów są różne, dostosowane do rozmiaru atrybutu jaki przechowują. Gdy bufor drzewa KCF jest pełen, następuje scalanie poziomu L_0 (bufora) z poziomem L_1. Scalanie wykonujemy równolegle na każdym drzewie tak, aby zachować wszystkie własności struktury CF.

Wyszukiwanie

Wyszukiwanie podzielone jest na dwie fazy. Pierwsza to znalezienie poziomu i pozycji w tym poziomie dla szukanego klucza w KCF. Następnie trzeba znaleźć wszystkie wartości, które zażądał użytkownik w kwerendzie. Ponieważ mamy ułożenie kolumnowe, to nie musimy łączyć wszystkich atrybutów w cały rekord. Wystarczy że połączymy tylko te kolumny, które będą potrzebne w zapytaniu. Dzięki temu możemy znacząco zmniejszyć koszt wyszukiwania względem wierszowego ułożenia, gdzie zawsze musimy wczytać cały rekord. Podczas wyszukiwania punktowego potrzebujemy tylko pojedynczego rekordu (lub jego fragmentu). Tak więc gdy znamy już pozycję i poziom klucza, możemy wykonać algorytm scalania atrybutów, aby znaleźć pozostałe wartości. Oznacza, to że za każdym razem musimy przeszukiwać logarytmicznie wiele drzew. Procedurę tę powtarzamy dla każdego zapytania. Ponieważ koszt odczytu kilku bajtów jest taki sam jak koszt odczytu całej strony na dysku SSD, to wyszukiwanie punktowe jest wolniejsze na kolumnowym ułożeniu, ponieważ zamiast pojedynczego przejścia po drzewie, mamy tyle przejść, ile kolumn musimy zwrócić w zapytaniu. O wiele lepszą sytuację mamy podczas wyszukiwania całego zakresu. Za pierwszym razem, gdy musimy znaleźć pierwsze atrybuty i ich pozycje używamy algorytmu scalania atrybutów. Później jednak wiemy, że z drzewa KCF musimy odczytać kolejno n wartości i te same n kolejnych wartości odczytujemy z pozostałych drzew bez konieczności ponownego wyszukiwania ich pozycji w drzewie. Zatem ułożenie kolumnowe posiada bardzo szybkie wyszukiwanie zakresu kluczy. Będzie ono tym szybsze od ułożenia wierszowego, im większa jest selektywność i mniejszy podzbiór wartości rekordu do wczytania.

Usuwanie

Usuwanie rekordu o podanym kluczu nie następujące natychmiastowo. Zamiast tego wstawiamy nowy rekord z kluczem, który mamy usunąć i ustawiamy typ tego rekordu jako *Usunięta dana*. Gdy podczas reorganizacji poziomów natrafimy na oba rekordy: oryginalny oraz ten do usunięcia, wtedy usuwamy oba rekordy. Ta operacja realizowana jest jako brak zapisu poszczególnych rekordów ze starych zbiorów do nowego zbioru. Oczywiście operacja usuwania jak i dodawania rekordu do usunięcia odbywa się na wszystkich drzewach KCF i NCF jednocześnie.

Scalanie poziomów

Reorganizacja drzewa przeprowadzana jest w przypadku, gdy któryś z poziomów nie ma wolnego miejsca na kolejne elementy. Wtedy uruchamiamy proces scalania dwóch poziomów: obecnego i kolejnego pod nim. Jeśli nie istnieje poziom niżej, wtedy jest on tworzony z odpowiednio dużą pojemnością (k razy większą od poprzedniego poziomu). Każde z drzew zawiera tyle samo danych, a jego pojemność jest dostosowana tak, aby na każdym poziomie mogło pomieścić tyle samo danych, co w pozostałych drzewach. Dlatego gdy istnieje potrzeba wykonania reorganizacji na głównym drzewie KCF, to również wykonujemy ją na wszystkich pozostałych drzewach NCF. Scalanie poziomów odbywa się według porządku narzuconego przez wartości kluczy w drzewie KCF. Oznacza to, że jeśli w danym momencie powinniśmy przepisać wartość z poziomu L_i, bo wartość obecnego klucza na tym poziomie jest najmniejsza wśród wszystkich kluczy, to przepisujemy odpowiednie wartości (dane z odpowiednich pozycji i poziomów) ze wszystkich drzew w obrębie CFT. Dzięki temu zachowujemy wszystkie własności struktury. Podczas reorganizacji pamiętamy, że jeśli poziom L_{i-1} będzie zawierał rekord o tym samym kluczu co rekord na poziomie L_i ale z odpowiednim znacznikiem do usunięcia, wtedy pomijamy oba rekordy (czyli wszystkie dane na wszystkich drzewach) i żadnego z nich nie wpisujemy do nowego poziomu L'_i. Możliwe, że podczas tego procesu poziom L'_i również nie będzie miał wystarczająco miejsca aby pomieścić wszystkie dane. Wtedy wykonujemy scalanie poziomów L'_i z L_{i+1}. Proces ten powtarzamy rekurencyjnie do momentu uzyskania poziomu mieszczącego wszystkie rekordy.

Scalanie atrybutów

Scalanie atrybutów w jeden rekord to najważniejszy i zarazem najtrudniejszy do zaprojektowania algorytm w kolumnowym układzie indeksu. Bez tego algorytmu nie bylibyśmy w stanie połączyć z powrotem atrybutów zapisanych w drzewach KCF i NCF w cały rekord, co oznaczałoby zerową przydatności struktury CFT. Algorytm scalania oparty jest na założeniu, że każde drzewo posiada tyle samo danych na każdym z poziomów. Oznacza to, że jeśli znaleźliśmy atrybut na pozycji *pos* danego poziomu *lvl*, to należy je połączyć z innymi atrybutami na tej samej pozycji tego samego poziomu. Dokonać tego możemy dzięki tablicy $numEntriesBeforePageArray$, która zawiera liczbę danych zapisanych przed stronami na ścieżce od obecnego wskaźnika (również od korzenia) do liścia. Wiedząc, że każda strona musi zawierać przynajmniej jeden taki wskaźnik, wiemy że w każdym momencie mamy dostęp do tej informacji. Za jej pomocą możemy jednoznacznie wyznaczyć stronę, która zawiera atrybut na szukanej pozycji *pos*. Sama procedura jest bardzo prosta i względnie szybka, ponieważ na każdym poziomie wystarczy odczytać tylko jedną stronę. Zatem koszt takiego scalenia jest logarytmiczny, podobnie jak wyszukiwania

punktowego w oryginalnym drzewie FD.

6.5.3 Opis algorytmów

Pseudokod 6.3 przedstawia dokładną listę kroków, jaką należy wykonać, aby scalić ze sobą oba poziomy w obrębie całego zbioru CFT. Sam algorytm jest bardzo podobny do scalania posortowanych list. Porządek reorganizacji ustalają klucze zapisane w drzewie KCF. Wiemy, że każdy poziom zawiera tyle samo danych oraz, że każdy element poziomu na pozycji *pos* jest częścią tego samego rekordu. Zatem reorganizacja musi zostać wykonana na wszystkich drzewach KCF i NCF w taki sam sposób, aby nie zaburzyć własności drzew. Linia 5 odpowiedzialna jest za pominięcie, czyli usunięcie wszystkich wskaźników na poziomie L_{i-1}. Możemy tak postąpić, ponieważ poziom L_i będzie usuwany i nadpisany nowym poziomem L'_i. Zatem wszystkie wskaźniki w poziomie (L_{i-1}) staną się nieaktualne. Poziom L_{i+1} zostanie bez zmian (chyba że rekurencyjnie wywołamy algorytm z linii 40), a więc nie możemy usunąć wskaźników zewnętrznych. Gdybyśmy to zrobili, stracilibyśmy bezpowrotnie dostęp do poziomu L_{i+1}. Z kolei wskaźniki wewnętrzne na poziomie L_i nie są już potrzebne. Zostały stworzone tylko dlatego, żeby był spełniony warunek mówiący o tym, że pierwszy rekord na stronie musi być wskaźnikiem. Ponieważ zmieni się układ poziomu L_i, który zostanie przepisany do L'_i, to również zmienią się wskaźniki wewnętrzne. Zatem pominięcie wszystkich wskaźników wewnętrznych w linii 6 jest prawidłowe. Linie 7-8 odpowiedzialne są za usuwanie rekordów. Tak jak wspominaliśmy, usuwanie elementów nie odbywa się natychmiastowo. Zamiast tego wstawiany jest rekord o tym samym kluczu odpowiednio oznaczony jako rekord do usunięcia. Gdy podczas scalania poziomów napotkamy na parę takich rekordów, wówczas usuwamy je pomijając oba rekordy w algorytmie. Oznacza to, że żaden z nich nie zostanie wpisany do nowego poziomu L'_i. Następnym krokiem algorytmu jest wybór poziomu, z którego musimy przepisać dane znajdujące się w pamięci wskazywanej przez obecne wskaźniki e_i i e_{i-1}. Oczywiście przeprowadzając reorganizację musimy zachować liniowy porządek w drzewie. Musimy zatem wybrać wskaźnik, który ma mniejszą wartość klucza. Jeśli to poziom L_{i-1} został wybrany, sytuacja jest bardzo prosta. Poziom ten zawiera tylko normalne dane, ponieważ wskaźniki zostały usunięte w linii 5, a rekordy do usunięcia w linii 8. Ponieważ klucze zapisane w drzewie KCF nadają porządek w całym zbiorze CFT, to bierzemy dane ze wszystkich drzew jednocześnie, mając na uwadze, że musimy je wczytać z poziomu L_{i-1} (linie 9-12). Gdy wybierzemy poziom L_i sytuacja jest trochę bardziej skomplikowana. Na tym poziomie oprócz normalnych danych są jeszcze zewnętrzne wskaźniki, które są potrzebne do budowania struktury $numEntriesBeforePageArray$. Wykorzystywana jest ona w algorytmie scalania atrybutów (6.4). Pamiętajmy, że gdy pierwsza dana na stronie dysku SSD nie jest wskaźnikiem zewnętrznym, to tworzony jest jego wewnętrzny odpowiednik, który przyjmuje wartość wskaźnika taką jak ostatni zewnętrzny wskaźnik na tym poziomie. Z tego względu gdy napotykamy na wskaźniki na poziomie L_i, to zapisujemy je w tymczasowych zmiennych, aby zawsze mieć do nich dostęp (linie 14-18). Ponieważ każdy atrybut może mieć różny rozmiar (wielkość wyrażoną w bajtach), to dany poziom w obrębie drzew CFT może składać się z różnej liczby stron dysku SSD. Zatem może zawierać różną liczbę wskaźników zewnętrznych. Przepisywanie normalnych danych musimy synchronizować pomiędzy wszystkie drzewa. W tym celu w liniach 7-26 przechodzimy po drzewach NCF tak długo, aż nie napotkamy na normalną daną. Oczywiście podczas przepisywania danych i wskaźników z L_i do nowego poziomu L'_i musimy pamiętać, aby odpowiednio zbudować nowy poziom oraz zachować strukturę drzewa poprzez odpowiednie połączenia z poziomu wyż-

szego do nowego poziomu (linie 19-24). Gdy tworzymy nową stronę to musimy stworzyć odpowiedni wskaźnik na tę stronę. Zapisujemy do niego pierwszy klucz strony, wskaźnik na obecną stronę oraz tworzymy tablicę $numEntriesBeforePageArray$. Tablica ta zawiera liczbę danych jaka jest zapisana przed stroną, na którą wskazuje oraz na kolejnych poziomach na ścieżce do liścia. Dlatego jako $numEntriesBeforePageArray[0]$ przypisujemy wartość $p.numEntriesBeforePage$, którą łatwo policzyć, ponieważ reorganizacja przeprowadzana jest na całym poziomie. Dzięki temu możemy w łatwy sposób śledzić liczbę zapisanych danych. Kolejne wartości tej tablicy są takie same jak ścieżki poniżej. Gdy mamy już wybrane dane (linie 10-12 lub 27-29) należy je wpisać do nowego poziomu L'_i, uważając aby nie zaburzyć struktury i własności drzewa. Po pierwsze jeśli tworzymy nową stronę i obecnie wpisywaną daną nie jest wskaźnik zewnętrzny, to tworzymy nowy wewnętrzny wskaźnik o wartościach ostatniego zewnętrznego wskaźnika. Procedurę w liniach 30-38 wykonujemy dla KCF oraz wszystkich NCF. Tak samo jak poprzednio (linie 21-24) teraz również musimy pamiętać, aby wpisać nowy wskaźnik do poziomu wyżej, gdyż stworzyliśmy nową stronę. Również w tym przypadku krok ten wykonujemy dla KCF i wszystkich drzew NCF. Do ostatnich kroków algorytmu należą sprawdzenie czy nowy poziom jest w stanie pomieścić wszystkie wpisane do niego wartości. Jeśli nie, to musimy rekurencyjnie wykonać nasz algorytm dla poziomów L'_i i L_{i+1}. Gdy skończymy przepisywać wszystkie rekordy, podmieniamy poziomy na nowe w naszej strukturze drzewa i kończymy proces reorganizacji.

Pseudokod 6.4 opisuje procedurę znajdowania wartości kolumny i dla danego rekordu e. Działanie algorytmu opiera się głównie na wartościach tablicy $numEntriesBeforePageArray$, która wypełniana jest podczas reorganizacji poziomów (6.3). Własność 2 mówi, że *Liczba danych w każdym FD-Tree na danym poziomie jest taka sama.* Z tego względu rekord możemy stworzyć łącząc ze sobą atrybuty zapisane w tym samym miejscu (na tej samej pozycji w danym poziomie) ze wszystkich drzew. Początkowo musimy przejść przez małe, buforowane w pamięci RAM drzewo B+ i znaleźć odpowiednią stronę drzewa FD. Drzewo B+ jak również wszystkie poziomy drzewa FD zawierają wskaźniki wraz z tablicą $numEntriesBeforePageArray$, która określa jak wiele danych zostało zapisane przed stroną na którą wskazuje wskaźnik. Wiemy zatem, że poszukiwania w drzewie FD musimy zacząć od strony, która miała największą wartość ale mniejszą od $e.pos$ (linia 3). Posiadając taką stronę możemy zacząć poszukiwania. Na każdym poziomie drzewa FD również szukamy takiego wskaźnika, dla którego wartość $numEntriesBeforePageArray$ jest największa ale mniejsza od $e.pos$. (linie 6-8). Warto zauważyć, że ta procedura wczytuje tylko jedną stronę na każdym poziomie. Gdy znajdujemy się na odpowiednim poziomie, musimy znaleźć wartość atrybutu na naszej stronie. Skoro znamy jego pozycję, wystarczy przesunąć się na odpowiednie miejsce (linie 12-15) i zwrócić szukaną wartość.

6.5.4 Dowód poprawności algorytmu scalania atrybutów

Teraz postaramy się udowodnić poprawność algorytmu $columnMerge$, to znaczy pokażemy, że potrafimy jednoznacznie dopasować wartość na każdej z kolumn.

Najpierw wprowadźmy niezbędne oznaczenia:

- Niech PTR_{L_i} będzie zbiorem wskaźników na poziomie L_i,

Pseudokod 6.3: cftLevelMerge(L_{i-1}, L_i)

```
1  Niech KLvl_{i-1}, KLvl_i, NLvl_{i-1}, NLvl_i będą poziomami z KCF i NCF
2  Niech Ke_{i-1}, Ke_i będą pierwszymi danymi na poziomach KLvl_{i-1} i KLvl_i
3  Niech Ne_{i-1}, Ne_i będą pierwszymi danymi na poziomach NLvl_{i-1} i NLvl_i
4  while Ke_{i-1} ≠ NULL and Ke_i ≠ NULL do
5      Pomiń wszystkie wskaźniki na KLvl_{i-1} oraz na Nlvl_{i-1}
6      Pomiń wszystkie wewnętrzne wskaźniki na KLvl_i oraz na NLvl_i
7      while Ke_{i-1}.type = DELETED and Ke_i.type = NORMAL and
         Ke_{i-1}.key = Ke_i.key do
8          Pomiń obie dane na wszystkich drzewach CFT
9      if Ke_{i-1}.key < Ke_i.key then
10         KEntryToInsert := Ke_{i-1}
11         NEntryToInsert := Ne_{i-1}
12         Niech Ke_{i-1}, Ne_{i-1} będą następnymi danymi w KLvl_{i-1} i NLvl_{i-1}
13     else
14         if Ke_i.type = EXTERNAL_FENCE then
15             KLastFence := Ke_i
16         while Ne_i.type ≠ NORMAL do
17             if Ne_i.type = EXTERNAL_FENCE then
18                 NLastFence := Ne_i
19             if Obecna strona p na NLvl'_i jest pusta then
20                 External.key := Ne_i.key
21                 External.pid := p
22                 External.numEntriesBeforePageArray[0] =
                     p.numEntriesBeforePage
23                 Skopiuj NLastFence.numEntriesBeforePageArray do
                     External.numEntriesBeforePageArray[i], i > 1
24                 Zapisz External na NLvl'_{i-1}
25             Zapisz Ne_i na NLvl'_i
26             Niech Ne_i będzie następną daną na NLvl'_i
27         KEntryToInsert := Ke_i
28         NEntryToInsert := Ne_i
29         Niech Ke_i, Ne_i będą następnymi danymi w KLvl_i i NLvl_i
30     if Obecna strona p na KLvl'_i jest pusta then
31         if KEntryToInsert.type ≠ ExternalFence then
32             Internal.key := KEntryToInsert.key
33             Internal.pid := KLastFence.pid
34             Zapisz Internal na KLvl'_i
35         External.key := KEntryToInsert.key
36         Wykonaj linie [21-24] dla KCF
37     Zapisz KEntryToInsert na KLvl'_i
38     Wykonaj linie [30-37] dla NCF
39     if KLvl'_i jest przepełniony then
40         cftLevelMerge(L'_i, L_{i+1})
41 Zastąp KLvl_{i-1}, KLvl_i, NLvl_{i-1}, NLvl_i przez Klvl'_{i-1} KLvl'_i, NLvl'_{i-1}, Nlvl'_i
```

Pseudokod 6.4: cftColumnMerge(input: Entry e, Int i)

```
1  Niech e.lvl będzie poziomem entry e
2  Niech e.pos będzie pozycją entry e na poziomie e.lvl
3  Niech p będzie stroną wskazywaną przez Wskanik f pochodzącego z
     Buforowanego B+ z kolumny i, takiego że
     f.numEntriesBeforePageArray[e.lvl] ma największą wartość z mniejszych od
     e.pos
4  while f.lvl < e.lvl do
5      Niech f będzie pierwszym wskaźnikiem na stronie p
6      while f.numEntriesBeforePageArray[e.lvl − f.lvl] < e.pos do
7          ptr := f.pid
8          Niech f będzie następnym wskaźnikiem na stronie p
9      p := ptr
10 counter := p.numEntriesBeforePage
11 Niech e′ będzie pierwszą daną na stronie p
12 while counter < e.pos do
13     if e′.type = NORMAL then
14         counter := counter + 1
15     Niech e′ będzie następną daną na stronie p
16 return e′
```

- Niech $PAGE_{L_i}$ będzie zbiórem stron na poziomie L_i,
- Niech $ptr'_{L_i} \prec ptr_{L_i}$ oznacza, że wskaźnik ptr'_{L_i} jest zapisany przed ptr_{L_i} na poziomie L_i naszego drzewa FD (niezależnie czy to KCF czy NCF),
- Niech $page'_{L_i} \prec page_{L_i}$ oznacza, że strona $page'_{L_i}$ jest zapisana przed stroną $page_{L_i}$ na poziomie L_i.
- Niech $PTR_{L_i,page}$ będzie zbiorem wskaźników zapisanych na stronie $page$ poziomu L_i,
- Niech $PTR^n_{L_i,page}$ będzie podzbiorem $PTR_{L_i,page}$ zawierającym wszystkie te wskaźniki, które spełniają warunek $ptr.numEntriesBeforePageArray[lvl - i] < n$.
- Ponieważ wskaźniki mają zdefiniowany porządek, możemy wyłonić z nich element maksymalny ptr_{max}. Dokładniej mówiąc, ptr_{max} jest elementem maksymalnym w zbiorze PTR, jeśli dla każdego $ptr' \in PTR$ takiego, że $ptr' \neq ptr_{max}$ zachodzi $f' \prec ptr_{max}$.

Lemat 6.1 *Jeśli* $ptr'_{L_i} \prec ptr_{L_i}$ *to* $ptr'.pid'_{L_{i+1}} \prec ptr.pid_{L_{i+1}}$ *lub* $ptr'.pid_{L_{i+1}} = ptr.pid_{L_{i+1}}$

Dowód. Dowód jest bardzo prosty. Wystarczy, że spojrzymy na algorytm scalający poziomy. Wskaźniki zapisywane są dokładnie według wcześniej zdefiniowanego porządku, zatem później zapisany wskaźnik nie może wskazywać na wcześniej zapisaną stronę. Mówiąc dokładniej będzie on wskazywał na tę samą stronę co poprzedni wskaźnik, jeśli jest to Wewnętrzny wskaźnik lub będzie wskazywał na kolejną stronę w przypadku, gdy jest to zewnętrzny wskaźnik. ♦

Twierdzenie 6.1 *Algorytm znajduje poprawną ścieżkę ze strony $page_{L_i}$ do strony $page_{L_j}$, która zawiera szukaną daną na pozycji npos, jeśli algorytm ten na każdym poziomie L_k $(i \leqslant k < j)$ wybierze maksymalnego ptr_{max} ze zbioru $PTR^{npos}_{L_k,page}$ i przejdzie do strony wskazywanej przez niego.*

Dowód. Udowodnimy to przez sprzeczność. Niech ptr' i ptr_{max} będą wskaźnikami ze zbioru $PTR^{npos}_{L_k,page}$ oraz niech ptr_{max} będzie elementem maksymalnym z tego zbioru. Załóżmy, że nasz algorytm nie wybiera maksymalnego elementu, czyli, że wybiera ptr' takie, że $ptr' \prec ptr_{max}$. Niech $PAGE'_{L_t,ptr'}$ i $PAGE_{L_t,ptr_{max}}$ zawierają strony z poziomu L_t dla poddrzew ukorzenionych odpowiednio w $ptr'.pid$ i w $ptr_{max}.pid$.

Przez $page_{min}$ oraz $page_{max}$ oznaczamy minimalne i maksymalne strony ze zbioru $PAGE_{L_t,ptr}$. Z definicji i własności CFT wynika, że

$$page'_{min}.numEntriesBeforePage = ptr'.numEntriesBeforePageArray[lvl - t]$$

oraz

$$page_{min}.numEntriesBeforePage = ptr_{max}.numEntriesBeforePageArray[lvl - t]$$

Z lematu 6.1, mamy:

$$page'_{max} \leqslant page_{min}$$

skąd

$$page'_{max}.numEntriesBeforePage \leqslant page_{min}.numEntriesBeforePage$$

Zatem, jeśli szukamy danej na pozycji $npos$, dla której

$$npos > ptr'.numEntriesBeforePageArray[lvl - t]$$

to

$$page'_{max}.numEntriesBeforePage \leqslant page_{min}.numEntriesBeforePage < n \qquad (6.2)$$

Zatem, zaczynając od strony wskazywanej przez wskaźnik ptr' możemy dotrzeć do wszystkich stron $page'_{min} < page < page'_{max}$. W konsekwencji możemy znaleźć wszystkie dane na pozycji $p'_{min}.numEntriesBeforePage < npos < p'_{max}.numEntriesBeforePage$. Na mocy równania 6.2, wiemy że nie możemy w ten sposób dojść do danej na pozycji $npos$. Zatem zawsze musimy wybierać maksymalny element. ♦

6.5.5 Analiza kosztów algorytmów

Zanim przeprowadzimy analizy kosztów algorytmów, wprowadźmy niezbędne oznaczenia.

- c - liczba kolumn (drzew) w zbiorze CFT
- n - liczba wstawianych danych
- R_{size} - rozmiar atrybutu wyrażony w bajtach
- P_{size} - rozmiar strony wyrażony w bajtach

- $P_{cap} = \left\lceil \frac{P_{size}}{R_{size}} \right\rceil$ - liczba atrybutów możliwych do zapisania na stronie
- L_i^j - liczba danych na poziomie i zaraz po skończeniu scalania j
- W_{cost} - koszt zapisu pojedynczej strony
- R_{cost} - koszt odczytu pojedynczej strony
- FD_{height} - liczba poziomów pojedynczego drzewa FD, pamiętajmy że każde drzewo w zbiorze CFT ma tyle samo poziomów.
- k - współczynnik pojemności 2 poziomów ($k = \frac{|L_{i+1}|}{|L_i|}$)
- m_i - liczba wywołań algorytmu scalania po n danych dla poziomu i
- N_i^{write} - sumaryczna liczba danych do zapisania podczas m_i scaleń dla poziomu i
- N_i^{read} - sumaryczna liczba danych do wczytania podczas m_i scaleń dla poziomu i
- t_{insert} - czas potrzebny na dodanie n elementów i przeprowadzenie m_i scaleń dla każdego poziomu i

Analiza kosztu algorytmu scalania atrybutów

Twierdzenie 6.2 *Niech T oznacza koszt scalania kolumn. Wtedy*

$$T = (c-1) \cdot \left\lceil \log_k \left(\frac{n}{|L_0|} \right) \right\rceil,$$

gdzie c oznacza liczbę kolumn (drzew) w zbiorze CFT, $k = \frac{|L_{i+1}|}{|L_i|}$ jest współczynnikiem pojemności 2 poziomów, n jest liczbą wstawianych danych oraz $|L_0|$ jest pojemnością bufora (poziomu 0).

Dowód. Podczas scalania atrybutów na każdym poziomie i czytamy wartości pól

$$numEntriesBeforePageArray[FD_{height} - i]$$

Strona może zawierać wiele wskaźników zewnętrznych, czyli może być wiele wartości $numEntriesBeforePageArray$. Wybieramy jednak taki wskaźnik, dla którego wartość

$$ptr.numEntriesBeforePageArray[FD_{height} - i]$$

jest największa z wartości mniejszych niż szukana pozycja *npos*. Następnie przechodzimy niżej na stronie *ptr.pid* wybranego wskaźnika. Zatem na każdym poziomie odczytamy tylko pojedynczą stronę i od razu przejdziemy do poziomu niżej. Zatem koszt połączenia atrybutu z innym atrybutem jest równy:

$$t_{columnMerge} = h$$
$$t_{columnMerge} = \left\lceil \log_k \left(\frac{n}{|L_0|} \right) \right\rceil$$

Z tego wynika, że koszt stworzenia rekordu mając wybraną wartość dowolnej kolumny jest równy:

$$t_{columnMerge} = (c-1) \cdot \left\lceil \log_k \left(\frac{n}{|L_0|} \right) \right\rceil$$

♦

Analiza kosztu wstawiania rekordu

Algorytm wstawiania dzieli się na dwie fazy. Pierwsza jest to podział rekordy na atrybuty i wstawienie poszczególnych wartości kolumn do odpowiednich drzew KCF i NCF. Początkowo wstawianie odbywa się do buforu zapisanego w pamięci RAM, ponieważ nasza analiza skupia się na operacjach WE/WY dysku SSD, koszt pierwszej fazy jest równy 0. Druga faza to reorganizacja poziomów. Ze względu na to, iż algorytm scalania wykonywany jest na wszystkich drzewach ze zbioru CFT bez dodatkowych kosztów scalania atrybutów, analizę możemy podzielić na dwie części: analizę scalania na pojedynczym drzewie FD oraz analizę pełnego scalania poziomów w obrębie całego zbioru CFT. Drzewo FD jest bardzo podobne do drzewa FA przedstawionego w poprzednim rozdziale **5**.

Przytoczmy zatem kilka niezbędnych faktów, które udało nam się ustalić podczas analizy drzewa FA, a które nie różnią się na drzewie FD.

Wiemy że wskaźników wewnętrznych jest maksymalnie tyle ile stron na danym poziomie, czyli

$$|internal_i| = \left\lceil \frac{|L_i|}{P_{cap}} \right\rceil .$$

Podobnie, wskaźników zewnętrznych jest dokładnie tyle ile stron na niższym poziomie, czyli

$$|external_i| = \left\lceil \frac{|L_{i+1}|}{P_{cap}} \right\rceil .$$

Wiemy, że zaraz po wykonaniu algorytmu scalania nowy poziom wyższy ma tylko wskaźniki zewnętrzne, stąd

$$|L_{i-1}^j| = |external_{i-1}|, \quad j > 0$$
$$|L_{i-1}^j| = \left\lceil \frac{|L_i|}{P_{cap}} \right\rceil, \quad j > 0$$

Koszt scalania to sumaryczny koszt zapisów danych dla obu poziomów zaraz po zakończeniu scalania, czyli

$$N_i^{write} = \sum_{j=1}^{m_i} (|L_{i-1}|^j + |L_{i-1}^j|)$$

Wiemy, że poziom L_{i-1} zawiera tylko wskaźniki zewnętrzne. Na podstawie ich liczby możemy oszacować ile było danych poniżej.

$$|L_i^j| = |external_{i-1}| \cdot P_{cap}$$

Zatem mamy

$$N_i^{write} = \sum_{j=1}^{m_i}(|external_{i-1}| \cdot P_{cap} + |L_{i-1}^j|)$$

$$N_i^{write} = \sum_{j=1}^{m_i}(|L_{i-1}^j| \cdot P_{cap} + |L_{i-1}^j|)$$

$$N_i^{write} = \sum_{j=1}^{m_i}((1 + P_{cap}) \cdot |L_{i-1}^j|)$$

$$N_i^{write} = (1 + P_{cap}) \cdot \sum_{j=1}^{m_i}|L_{i-1}^j| \tag{6.3}$$

Gdy założymy, że wszystkie rekordy n zostały scalone do najniższego poziomu, dostajemy

$$n = \sum_{j=1}^{m_i}(|L_{i-1}| - |external_{i-1}^j|)$$

$$n = \sum_{j=1}^{m_i}(|L_{i-1}| - |L_{i-1}^j|)$$

$$n = m_i \cdot |L_{i-1}| - \sum_{j=1}^{m_i}|L_{i-1}^j|$$

$$n = m_i \cdot |L_{i-1}| - \sum_{j=1}^{m_i}\left\lceil \frac{|L_i^j|}{P_{cap}} \right\rceil$$

$$\sum_{j=1}^{m_i}|L_{i-1}^j| = (m_i \cdot |L_{i-1}| - n) \tag{6.4}$$

Łącząc ze sobą 6.3 z 6.4 dostajemy

$$N_i^{write} = (1 + P_{cap}) \cdot (m_i \cdot |L_{i-1}| - n)\ . \tag{6.5}$$

Z rozdziału **5** pamiętamy, że aby przeprowadzić scalanie musimy odczytać oba poziomu, mają one tyle danych ile wpisaliśmy po zakończeniu poprzedniego scalania, stąd

$$N_i^{read} = \sum_{j=1}^{m_i}(|L_{i-1}^{j-1}| + |L_i^{j-1}|)\ .$$

Aby wyrazić N_i^{read} za pomocą N_i^{write} skorzystajmy z faktu, że możemy założyć, iż liczba danych na poziomie i po scaleniu j jest równa sumie 2 poziomów utworzonych po poprzednim scaleniu

$$|L_{i-1}^j| = |L_{i-1}^{j-1}| + |L_i^{j-1}|$$

Dzięki temu, możemy łatwo wyrazić N_i^{read} za pomocą N_i^{write}.

$$N_i^{write} = \sum_{j=1}^{m_i}(|L_{i-1}^j| + |L_i^j|)$$

$$N_i^{write} = \sum_{j=1}^{m_i}(|L_{i-1}^{j-1}| + |L_i^{j-1}| + |L_i^j|) \tag{6.6}$$

$$N_i^{read} = N_i^{write} - \sum_{j=1}^{m_i}|L_{i-1}^j|$$

$$N_i^{read} = N_i^{write} - \frac{n}{P_{cap}} \tag{6.7}$$

Jak widać odczytujemy mniej danych, ponieważ nie musimy odczytywać nowych wskaźników zewnętrznych na poziomie L_{i-1}^j.

Możemy teraz oszacować czas potrzebny na scalanie wszystkich poziomów. Zanim to zrobimy, warto przypomnieć sobie górne ograniczenie na liczbę scaleń.

$$m_i < \frac{n}{|L_{i-1}| - \frac{|L_{i-1}|}{P_{cap}} - -\frac{|L_i|}{P_{cap}}}$$

$$m_i < \frac{P_{cap}}{P_{cap} - k - 1} \cdot \frac{n}{|L_{i-1}|}$$

Teraz w końcu jesteśmy gotowi aby oszacować czas potrzebny na wstawianie wszystkich elementów n oraz wszystkich reorganizacji poziomów.

$$t_{insert} = \frac{1}{n} \cdot \sum_{i=1}^{FD_{height}-1}\left(\frac{R_{size} \cdot N_i^{write}}{W_{cost}} + \frac{R_{size} \cdot N_i^{read}}{R_{cost}}\right)$$

$$t_{insert} < \frac{W_{cost} + R_{cost}}{W_{cost} \cdot R_{cost}} \cdot P_{size} \cdot \sum_{i=1}^{FD_{height}-1}\left(\frac{m_i \cdot |L_{i-1}|}{n} - 1\right)$$

$$t_{insert} < \frac{W_{cost} + R_{cost}}{W_{cost} \cdot R_{cost}} \cdot P_{size} \cdot \sum_{i=1}^{FD_{height}-1}\left(\frac{P_{cap}}{P_{cap} - k - 1} - 1\right) \tag{6.8}$$

Wiemy, że wysokość trzeba FD_{height} jest logarytmiczna, uzależniona od współczynnika k oraz wielkości drzewa B+ w RAM czyli poziomu L_0. Wynosi więc ona:

$$FD_{height} = \left\lceil \frac{\log_k n}{|L_0|} \right\rceil$$

Możemy zatem przepisać nierówność 6.8, korzystając z powyższej równości. Dostajemy

$$t_{insert} < \frac{W_{cost} + R_{cost}}{W_{cost} \cdot R_{cost}} \cdot \frac{k+1}{P_{cap} - k - 1} \cdot P_{size} \cdot \left\lceil \frac{\log_k n}{|L_0|} \right\rceil$$

więc

$$t_{insert} = \mathrm{O}\left(\frac{k}{P_{cap} - k} \log n \right) \tag{6.9}$$

Zatem amortyzowany czas na wstawianie atrybutu do drzewa jest logarytmiczny ze względu na liczbę wstawianych elementów i wynosi

$$\mathrm{O}\left(\frac{k}{P_{cap} - k} \log n \right)$$

Twierdzenie 6.3 *Niech T oznacza amortyzowany czas wstawiania rekordu do indeksu CFT, wtedy*

$$T = \mathrm{O}\left(\frac{c \cdot k}{P_{cap} - k} \log n \right)$$

gdzie c jest liczbą kolumn (drzew) w zbiorze CFT, k jest współczynnikiem pojemności 2 poziomów, P_{cap} jest liczbą atrybutów możliwych do zapisania na stronie, zaś n jest liczbą wstawianych danych.

Dowód. Ponieważ wykonujemy c niezależnych reorganizacji poziomów, z których każda posiada tyle samo danych, możemy bardzo łatwo rozszerzyć analizę na wszystkie kolumny. Oczywiście parametry:

- R_{size} - rozmiar atrybutu wyrażony w bajtach
- $P_{cap} = \left\lceil \frac{P_{size}}{R_{size}} \right\rceil$ - liczba atrybutów możliwych do zapisania na stronie

mogą ulec zmianie, ponieważ każda kolumna może posiadać inny rozmiar, to jednak wpływa to tylko na liczbę użytych stron oraz liczbę wskaźników jakie trzeba zapisać. Powyższa analiza pokazała, że jest ich na tyle mało, że możemy pominąć koszt ich zapisu, zaś zmiana parametru P_{cap} nie jest asymptotyczna, różnice w wielkościach można wyrazić pewną stałą, a zatem nie wpłyną na koszt amortyzowany wyrażony notacją duże O. Zatem sumaryczny koszt dodawania całego rekordu do struktury CFT można zapisać jako

$$\mathrm{O}\left(\frac{c \cdot k}{P_{cap} - k} \cdot \log_k n \right) .$$

♦

6.5.6 Eksperymenty

W tej części rozdziału przedstawimy i omówimy eksperymenty wykonane za pomocą symulatora SIPS, który został dokładnie opisany w rozdziale 4. Do eksperymentów wybrano 3 dyski SSD o odmiennej charakterystyce. Tabela 6.5 zawiera szczegółowe parametry wybranych modeli. Zwykle eksperymenty zostają przeprowadzane na trzech tabelach (Sklep, Nowe zamówienie oraz Klient). Ponieważ tym razem przedmiotem eksperymentów jest kolumnowe indeksowanie baz danych, użyjemy tylko dwóch tabel, które posiadają dużą liczbą kolumn. Atrybuty wybranych tabel pokazane są w tabeli 6.9. Naszą domyślną tabelą będzie Klient, ponieważ ostatni atrybut jest znacznie większy od pozostałych. Oznacza to, że brak wczytywania tej kolumny podczas wyszukiwania znacząco poprawi wydajność obsługi tej kwerendy.

		Pojemność		Prędkość losowego		Prędkość sekwencyjnego	
Model	Interfejs	Strony	Bloku	Odczytu	Zapisu	Odczytu	Zapisu
Samsung 840	SATA	8KB	512KB	390MB/s	182MB/s	585MB/s	535MB/s
Toshiba VX500	SATA	4KB	256KB	379MB/s	267MB/s	568MB/s	525MB/s
Intel DCP4511	NVMe	4KB	256KB	1,2GB/s	240MB/s	2GB/s	1.47GB/s

Tabela 6.5: Wybrane modele dysków SSD

Podstawowy zestaw kwerend

Podstawowy zestaw kwerend służy do sprawdzenia wydajności testowanego indeksu w sytuacji wykorzystania tego indeksu podobnego do zwykłej struktur danych. Pierwotnie zestaw podstawowy opisany dokładnie w rozdziale 4 składał się z trzech podstawowych operacji: wstawiania, wyszukiwania i usuwania. Każda z tych operacji przeprowadzana była na jednym rekordzie. Również wyszukiwanie zwracało tylko jeden rekord, ponieważ było to wyszukiwanie punktowe. Ponieważ kolumnowe ułożenie danych, głównie optymalizuje operacje wyszukiwania, to do zestawu kwerend wprowadziliśmy również wyszukiwanie zakresu kluczy o podanej selektywności (1%, 5% i 10%). Z tego samego powodu zrezygnowaliśmy ze wzorców: ZP_{zapis}, ZP_{odczyt}, ZP_{balans} i skupiliśmy się tylko na wzorcu ZP_{odczyt}. Wzorzec ten składa się z 15% operacji wstawiania pojedynczego rekordu, 80% operacji wyszukiwania punktowego lub zakresowego z użyciem klucza, 5% operacji usuwania pojedynczego rekordu. Zestaw podstawowy określa tylko stosunek operacji modyfikacji indeksu do wyszukiwań rekordów, a liczba tych operacji jest dowolna. Domyślnie jest to 100tys. operacji. Ponieważ wybrane indeksowania opisane poniżej posiadają buforowanie, to 100tys. operacji nie wystarczyły aby uzyskać stabilne wyniki. Dlatego też testy przeprowadzono dla 1mln operacji.

Każdy eksperyment przeprowadzony został z użyciem trzech indeksowań. Wybrano indeksowanie z użyciem klasycznego drzewa FD, gdzie rekordy zapisane są wierszowo, tak jak to ma miejsce w oryginalnym drzewie FD. Dodatkowo wybraliśmy strukturę FBDSM [52], która jest zmodyfikowaną wersją struktury DSM [30], dostosowaną do charakterystyki pamięci flash i dysków SSD. Niestety chociaż ta struktura zapisuje dane kolumnowo, to nie jest posortowana po kluczu. Oznacza to, że operacje takie jak wyszukiwanie i usuwanie potrzebuje wczytać cały zbiór danych, aby znaleźć pozycję wybranego atrybutu, a następnie odczytać pozostałe atrybuty, znajdujące się na tej samej pozycji ale w innym obszarze pamięci. Dzięki zastosowanemu buforowaniu i podziale na dwie tabele PT (tabele główną) i LT (tabelę zmodyfikowanych wartości) znacząco poprawiono operacje dodawania rekordów oraz ich modyfikacji względem pierwotnego pomysłu DSM. Ostatnią strukturą danych jest zaproponowana struktura CF-Tree.

Zazwyczaj testujemy indeksy operujące na danych zapisanych wierszowo. Wtedy liczba atrybutów żądanych przez użytkownika w kwerendzie nie ma znaczenia, ponieważ i tak musimy odczytać z dysku cały rekord. Ponieważ tym razem testujemy kolumnowe bazy danych, to każdy z eksperymentów został przeprowadzony kilka razy. Za każdym razem kwerenda szukająca żądała więcej kolejnych atrybutów. Czyli gdy testowaliśmy indeksy używając danych z tabeli Sklep, to gdy kwerenda miała prosić o wyszukiwanie wartości dla 3 kolumn (klucz zawsze mamy podany i zwracamy w rekordzie), były to

Nazwa kolumny	Rozmiar kolumny
C_ID	4B
C_D_ID	4B
C_W_ID	8B
C_FIRST	16B
C_MIDDLE	2B
C_LAST	16B
C_STREET_1	20B
C_STREET_2	20B
C_CITY	20B
C_STATE	2B
C_ZIP	9B
C_PHONE	16B
C_SINCE	16B
C_CREDIT	2B
C_CREDIT_LIM	16B
C_DISCOUNT	8B
C_BALANCE	16B
C_YTD_PAYMENT	16B
C_PAYMENT_CNT	4B
C_DELIVERY_CNT	4B
C_DATA	500B

Tabela 6.6: Klient TPC-C

Nazwa kolumny	Rozmiar kolumny
W_ID	8B
W_NAME	10B
W_STREET_1	20B
W_STREET_2	20B
W_CITY	20B
W_STATE	2B
W_ZIP	9B
W_TAX	8B
W_YTD	16B

Tabela 6.7: Sklep TPC-C

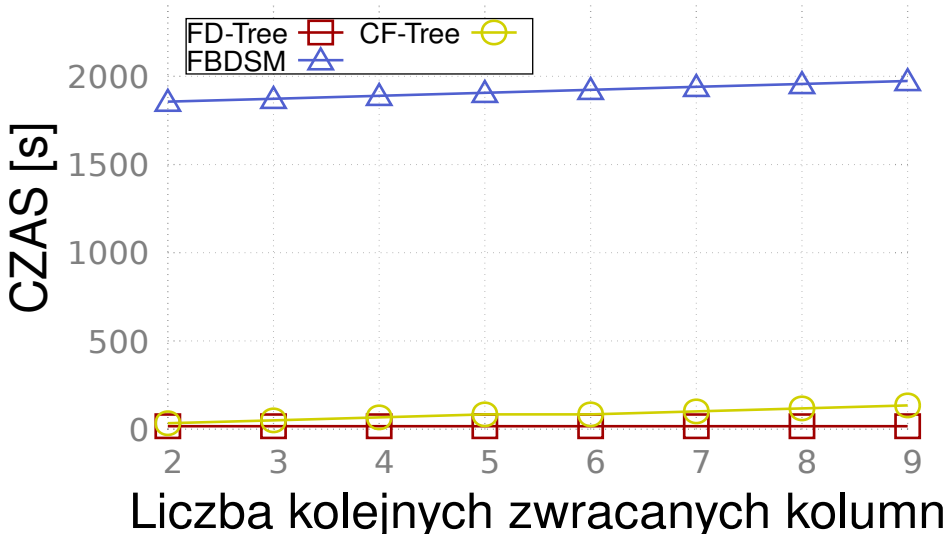
Nazwa kolumny	Rozmiar kolumny
NO_O_ID	8B
NO_D_ID	4B
NO_W_ID	8B

Tabela 6.8: Nowe Zamówienie TPC-C

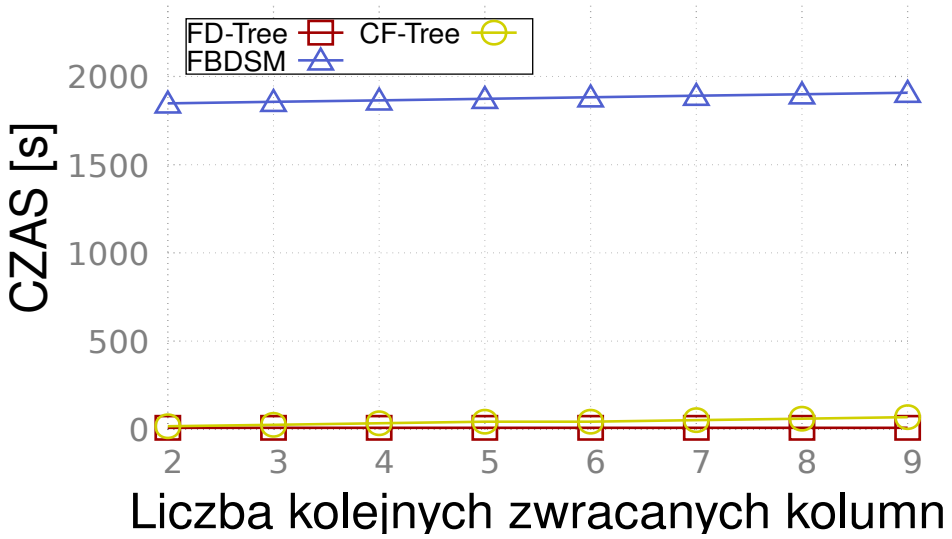
Tabela 6.9: Atrybuty wybranych tabel TPC-C

kolumny W_ID (klucz), W_NAME, W_STREET_1 i W_STREET_2. Dzięki temu możemy zobaczyć wpływ liczby zwracanych kolumn i liczby bajtów na wydajność kwerend podczas gdy baza danych zapisana jest na wybranych trzech indeksach.

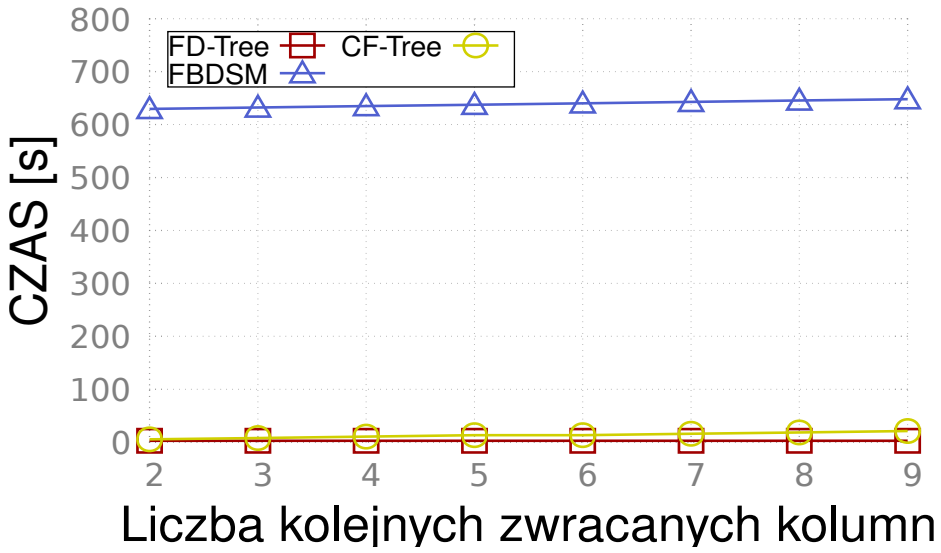
Rysunki 6.17, 6.18 oraz 6.19 przedstawiają wyniki eksperymentów dla odpowiednio dysków: Samsung 840, Toshiba VX 500 oraz Intel DCP4511. Eksperymenty przeprowadzona na tabeli Sklep z użyciem zestawu podstawowego ZP_{odczyt}, w którym jako wyszukiwania użyto punktowego wyszukiwania pojedynczego rekordu. Analizując wykresy widzimy, że struktury zapisujące dane kolumnowo osiągają gorszy rezultat od oryginalnego drzewa FD zapisanego wierszowo. FBDSM nie jest przystosowany do obsługi kwerend wyszukujących, zatem każda operacja przeprowadzana jest jako pełen skan obszarów pamięci potrzebnych do stworzenia wyniku kwerendy. Im więcej kolumn użytkownik zażądał w zapytaniu, tym więcej obszarów pamięci musimy wczytać. Z kolei struktura CF-Tree w trakcie ostatniej fazy wyszukiwania używa algorytmu scalania atrybutów. Im więcej kolumn zwracamy, tym więcej razy musimy wykonać tę procedurę.



Rysunek 6.17: Czas wykonania zestawu kwerend ZP_{odczyt}
Dysk: Samsung 840
Tabela: Sklep (113B)

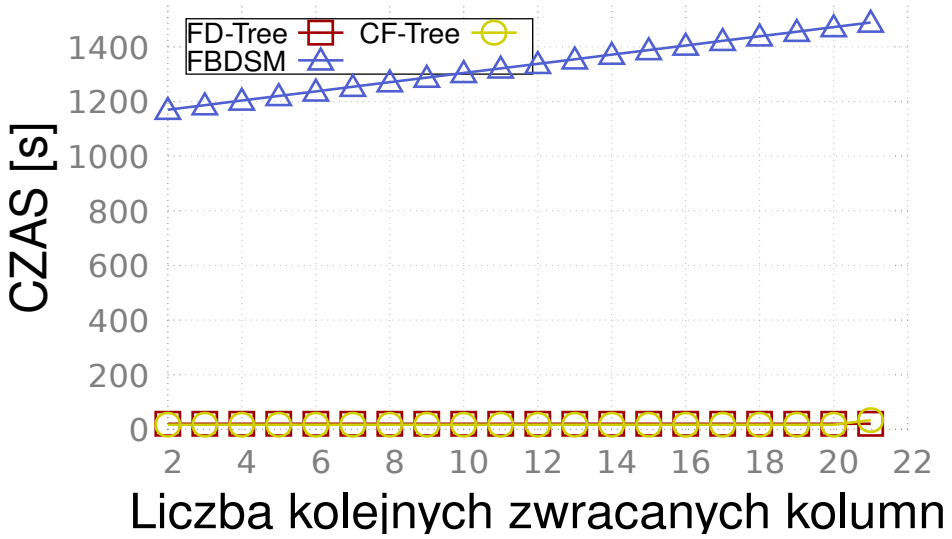


Rysunek 6.18: Czas wykonania zestawu kwerend ZP_{odczyt}
Dysk: Toshiba VX500
Tabela: Sklep (113B)

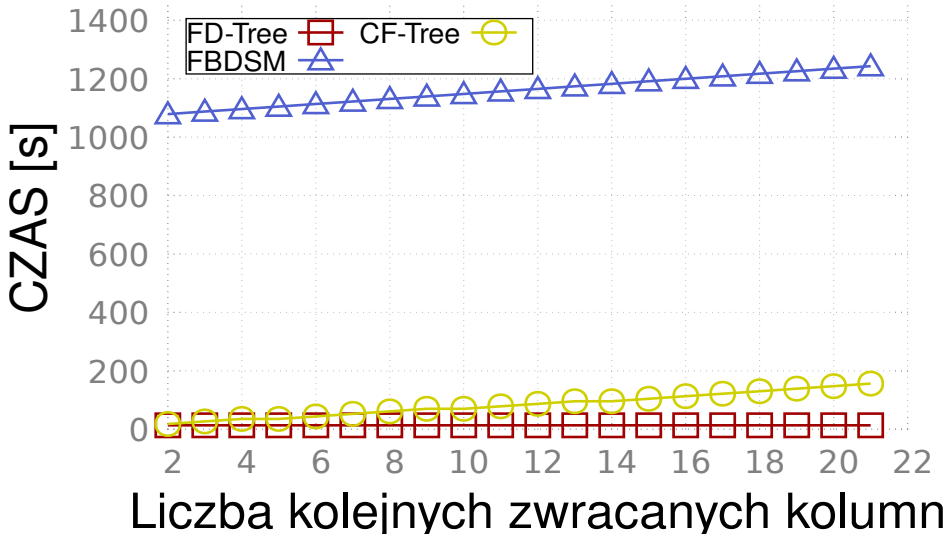


Rysunek 6.19: Czas wykonania zestawu kwerend ZP_{odczyt}
Dysk: Intel DCP4511
Tabela: Sklep (113B)

Jej koszt jest bardzo niski (logarytmiczny) względem liczby danych znajdujących się w indeksie. Jednak ze względu na to, że wyszukiwanie w strukturach FD jak i CF-Tree również jest logarytmiczne, to koszt scalania znacząco wpływa na całkowity koszt wyszukiwania punktowego. Wierszowa implementacja drzewa FD, wyszukując podany rekord, przechodzi drzewo z góry na dół tylko raz, wczytując tylko jedną stronę każdego poziomu. Mimo, iż na końcu zamiast wczytać tylko potrzebne atrybuty, wczytujemy cały rekord, to w przypadku tego wyszukiwania nie wpływa to na czas wykonania kwerendy, ponieważ rekord mieści się na jednej stronie.

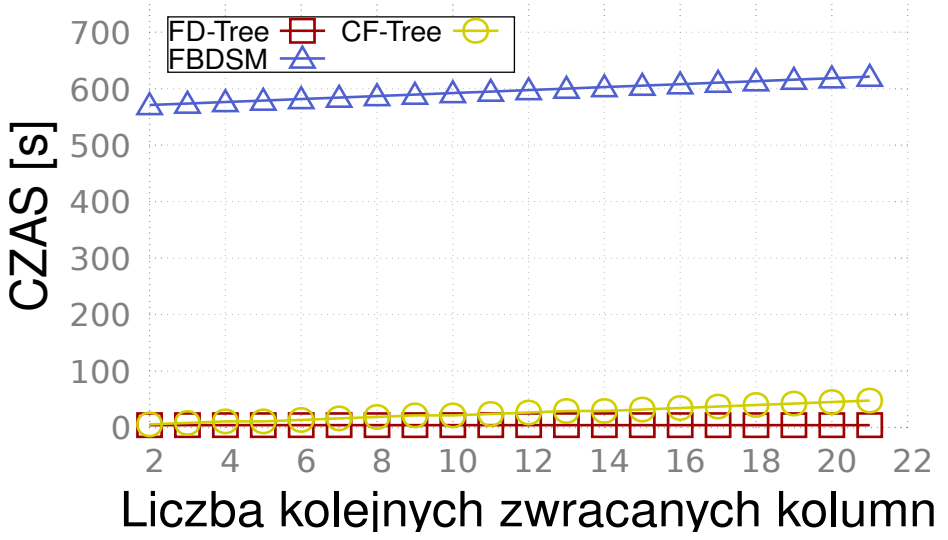


Rysunek 6.20: Czas wykonania zestawu kwerend ZP_{odczyt}
Dysk: Samsung 840
Tabela: Klient (719B)



Rysunek 6.21: Czas wykonania zestawu kwerend ZP_{odczyt}
Dysk: Toshiba VX500
Tabela: Klient (719B)

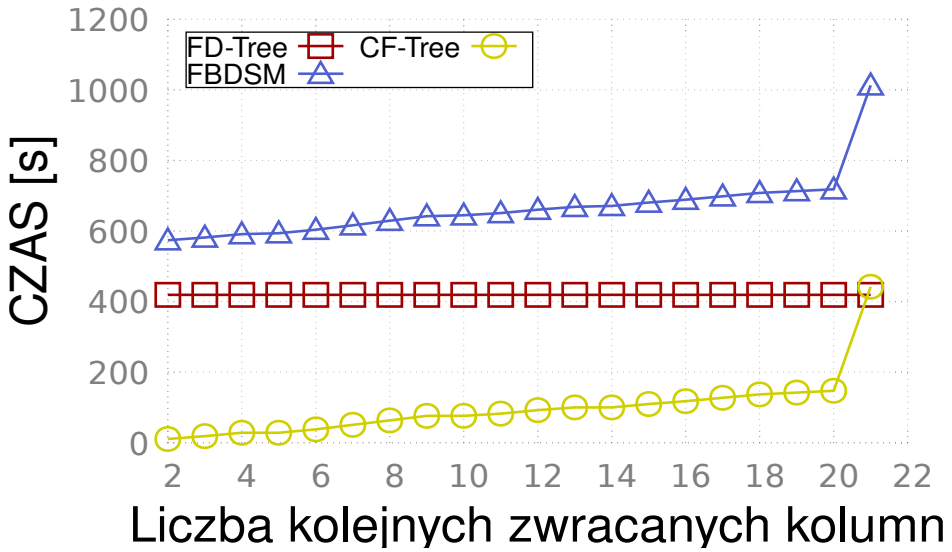
Pamiętajmy, że czas odczytu kilku bajtów jak i jednej całej strony jest taki sam na dyskach SSD, ponieważ operujemy na całych stronach. Mimo iż wybrane dyski posiadają różne charakterystyki, to na wszystkich wykresach obserwujemy ten sam trend.



Rysunek 6.22: Czas wykonania zestawu kwerend ZP_{odczyt}
Dysk: Intel DCP4511
Tabela: Klient (719B)

Rysunki 6.20, 6.21 oraz 6.22 ponownie przedstawiają wyniki eksperymentów dla odpowiednio dysków: Samsung 840, Toshiba VX 500 oraz Intel DCP4511. Tym razem użyto tabeli Klient. Ponownie za każdym razem wyszukiwanie było przeprowadzane punktowo. Jak możemy zauważyć, mimo że rekord tabeli Klient jest ponad 6 razy większy od rekordu tabeli Sklep, to obserwujemy tę samą sytuację. Wierszowa implementacja struktury FD

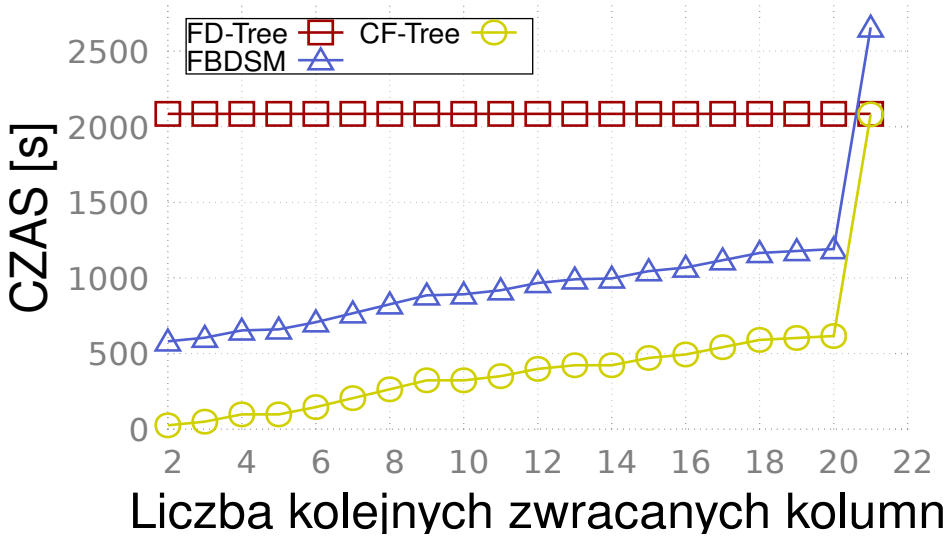
zawsze wykonuje zestaw kwerend w najkrótszym czasie. Ponownie wynika to z faktu, iż podczas wyszukiwania punktowego wielkość rekordu nie ma znaczenia, ponieważ zawsze musimy wykonać jeden odczyt, niezależnie od liczby bajtów jakie musimy odczytać. Warto zauważyć, że czas wyszukiwania z użyciem struktury FBDSM wraz ze wzrostem liczby zwracanych kolumn rośnie teraz szybciej niż w poprzedniej serii eksperymentów. Wynika to z faktu, iż struktura FBDSM skanuje w pełni obszary pamięci, im atrybut większy, tym większy obszar pamięci do wczytania, co znacząco wydłuża czas wykonania kwerendy.



Rysunek 6.23: Czas wykonania zestawu kwerend ZP_{odczyt}
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)

Z poprzednich dwóch serii eksperymentów wiemy, że charakterystyka dysku ma wpływ na czas wykonania się zestawu kwerend (szybszy odczyt strony to szybsze wyszukiwanie), jednak nie ma wpływu na trend jaki obserwowaliśmy. Zawsze FBDSM miało najgorszy czas, zawsze wierszowe ułożenie danych było najlepszym wyborem do wyszukiwania punktowego. Z tego względu w tej serii eksperymentów zdecydowaliśmy się na pokazanie wyników tylko dla najszybszego z wybranych dysków: Intel DCP4511. Dodatkowo eksperymenty przeprowadziliśmy tylko na tabeli Klient, ponieważ jej charakterystyka ma duży wpływ na szybkość wykonywania się kwerend wyszukiwania. Zamiast wyszukiwania punktowego użyliśmy tym razem wyszukiwania kluczy z całego zakresu dla różnych selektywności. Rysunki 6.23, 6.24 i 6.25 przedstawiają wyniki eksperymentów dla zestawu ZP_{odczyt} z użyciem wyszukiwań dla odpowiednio 1%, 5% oraz 10%. Tabela 6.10 zawiera dokładnie wyniki tych eksperymentów wraz z liczbą zwracanych bajtów przez kwerendy wyszukujące dla użytych w eksperymencie selektywności 1%, 5% oraz 10%.

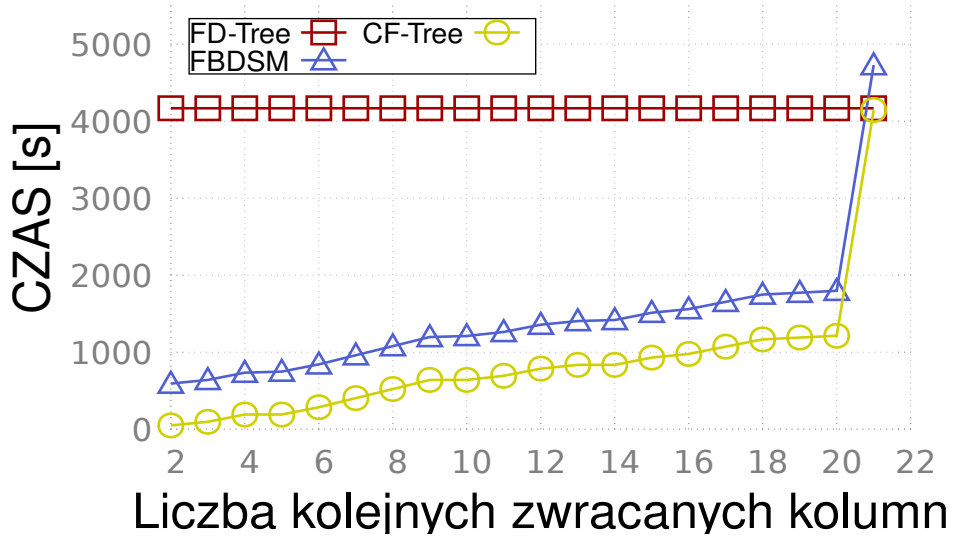
Dla selektywności 1% najgorsze wyniki osiągnęła znowu struktura FBDSM. Przy tak małej selektywności zysk wynikający z odczytywania tylko potrzebnych fragmentów rekordu zniwelowany jest poprzez potrzebę skanowania całego obszaru pamięci. Należy zauważyć, że mimo iż podczas wyszukiwania punktowego struktura CF-Tree osiągała gorszy czas od wierszowej implementacji FD-Tree, to tym razem nawet pomimo małej selektywności, zaproponowana kolumnowa struktura osiąga znacznie lepsze wyniki od klasycznej implementacji indeksu. Tym razem procedura scalająca rekordy wywoływana jest



Rysunek 6.24: Czas wykonania zestawu kwerend ZP_{odczyt}
Selektywność 5%
Dysk: Intel DCP4511
Tabela: Klient (719B)

tylko raz podczas każdego wyszukiwania, gdy ma miejsce pierwsze scalanie atrybutów. Następnie wystarczy przesuwać się o tyle samo pozycji równoległe na każdym z drzew. Dzięki temu dodatkowy czas potrzebny na scalanie kolumn jest zminimalizowany, a efekt odczytu mniejszej liczby stron z dysku SSD znacząco wpłynął na poprawę wydajności obsługi kwerendy. Im mniej kolumn musimy zwrócić użytkownikowi, tym mniej pamięci musimy odczytać z dysku. Dlatego czas wykonania kwerendy rośnie liniowo wraz ze wzrostem odczytanych bajtów w celu wykonania zapytania. Dodatkowo zauważmy, że nawet w przypadku zwracania całego rekordu, kolumnowy indeks CF-Tree wykonuje kwerendę w porównywalnym czasie co wierszowa implementacja drzewa FD.

Wyniki dla selektywności 5% i 10% są bardzo podobne. Przy tak dużej selektywności skan pełnego obszaru pamięci dla pewnych atrybutów osiąga lepszy wynik niż wierszowe indeksowanie. Jednak struktura FBDSM wykonuje kwerendę dłużej niż wierszowe drzewo FD w przypadku obsługi kwerendy dla całego rekordu. Z kolei nowa struktura CF-Tree radzi sobie świetnie. Za każdym razem osiąga krótszy czas wykonania zestawu kwerend nie tylko do indeksu FD-Tree, ale także od struktury FBDMS. Analizując podstawowy zestaw kwerend, możemy zauważyć że struktura CF-Tree osiąga zawsze gorsze wyniki od wierszowego indeksu w sytuacji, gdy przeprowadzamy wyszukiwanie punktowe i zawsze lepsze wyniki, gdy przeprowadzamy wyszukiwanie z zakresu kluczy. Dodatkowo CF-Tree zawsze wykonuje kwerendy szybciej od struktury FBDSM, ponieważ nie tylko zawiera kolumnowe ułożenie danych, ale także jest w pełni posortowana. Procedury modyfikacji indeksu są równie szybkie co w strukturze FBDSM, a samo wyszukiwanie nie wykonuje skanowania części tabeli, co znacząco przyspiesza wykonanie tego zapytania w stosunku do implementacji w FBDSM.



Rysunek 6.25: Czas wykonania zestawu kwerend ZP_{odczyt}
Selektywność 10%
Dysk: Intel DCP4511
Tabela: Klient (719B)

	Selektywność 1%			Selektywność 5%			Selektywność 10%		
Liczba zwracanych kolumn (bajtów)	**FD**	**FBDSM**	**CFT**	**FD**	**FBDSM**	**CFT**	**FD**	**FBDSM**	**CFT**
2 (8B)	420s	573s	10s	2084s	581s	25s	4167s	592s	48s
3 (16B)	420s	581s	18s	2084s	605s	49s	4167s	640s	96s
4 (32B)	420s	591s	28s	2084s	653s	97s	4167s	734s	190s
5 (34B)	420s	593s	28s	2084s	659s	97s	4167s	747s	190s
6 (50B)	420s	603s	37s	2084s	707s	145s	4167s	842s	284s
7 (70B)	420s	616s	50s	2084s	766s	204s	4167s	960s	403s
8 (90B)	420s	629s	63s	2084s	826s	264s	4167s	1078s	521s
9 (110B)	420s	641s	76s	2084s	885s	323s	4167s	1197s	640s
10 (112B)	420s	644s	76s	2084s	891s	323s	4167s	1210s	640s
11 (121B)	420s	650s	82s	2084s	918s	350s	4167s	1262s	692s
12 (137B)	420s	660s	92s	2084s	966s	398s	4167s	1357s	787s
13 (153B)	420s	668s	100s	2084s	990s	422s	4167s	1405s	835s
14 (155B)	420s	671s	100s	2084s	997s	422s	4167s	1418s	835s
15 (171B)	420s	680s	109s	2084s	1045s	470s	4167s	1512s	929s
16 (179B)	420s	688s	117s	2084s	1069s	494s	4167s	1560s	977s
17 (195B)	420s	698s	127s	2084s	1117s	542s	4167s	1645s	1072s
18 (211B)	420s	707s	136s	2084s	1165s	590s	4167s	1749s	1166s
19 (215B)	420s	713s	142s	2084s	1178s	603s	4167s	1773s	1190s
20 (219B)	420s	718s	147s	2084s	1190s	616s	4167s	1797s	1214s
21 (719B)	420s	1012s	441s	2084s	2656s	2081s	4167s	4728s	4145s

Tabela 6.10: Dokładnie wyniki eksperymentów dla ZP_{odczyt} z wyszukiwaniem zakresowym
Dysk: Intel DCP4511
Tabela: Klient (719B)

Rozszerzony zestaw kwerend

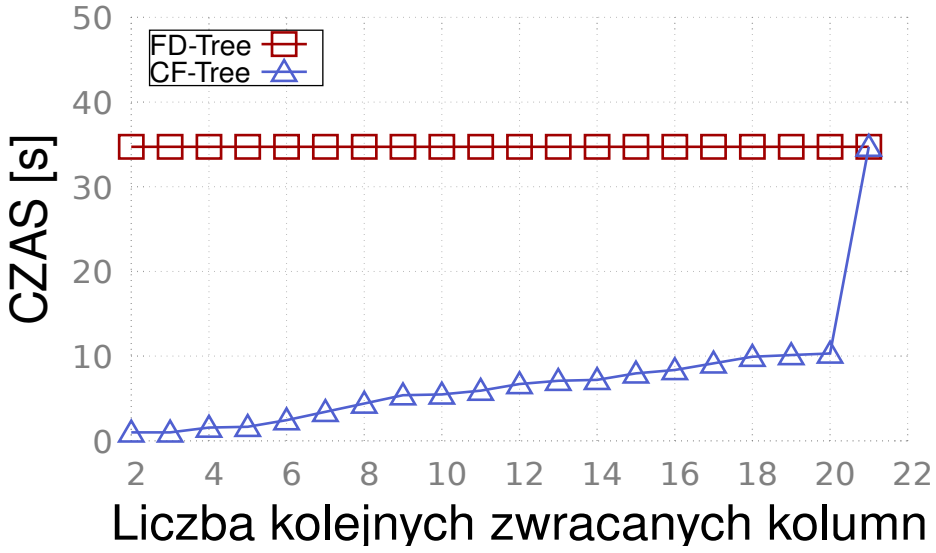
Celem zestawu rozszerzonego jest symulacja hurtowni danych na podstawie zestawów w TPC-C. Początkowo tabela zawiera 10mln rekordów. Następnie przeprowadzamy kolejno dodawanie, wyszukiwanie i usuwanie rekordów. Każde wyszukiwanie ma ustawioną selektywność na poziomie 1%, 5% lub 10% całej tabeli. Charakterystyka wszystkich czterech zestawów opisana w rozdziale **4** jest następująca:

1. ZR_A - 100 serii, każda seria zawiera operacje dodawania 5 rekordów, 10 wyszukiwań o podanej selektywności oraz usuwanie 5 rekordów,

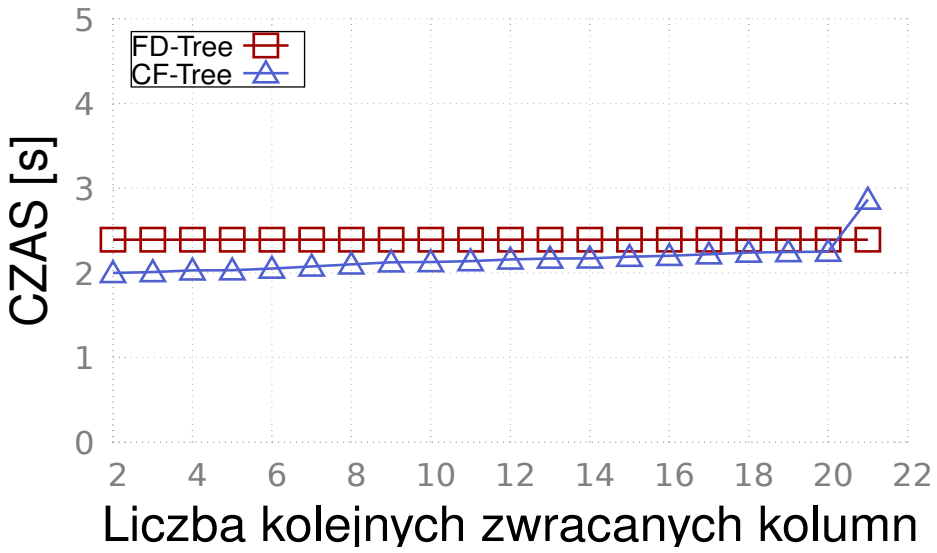
2. ZR_B - 5 serii, każda seria zawiera operacje dodawania 100000 rekordów, 5 wyszukiwań o podanej selektywności oraz usuwanie 100000 rekordów,

3. ZR_C - 10 serii, każda seria zawiera operacje dodawania 10000000 rekordów, 20 wyszukiwań o podanej selektywności oraz usuwanie 1000000 rekordów,

4. ZR_D - 10 serii, każda seria zawiera operacje dodawania 1000000 rekordów, 10 wyszukiwań o podanej selektywności oraz usuwanie 10000 rekordów.



Rysunek 6.26: Czas wykonania zestawu kwerend ZR_A
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)

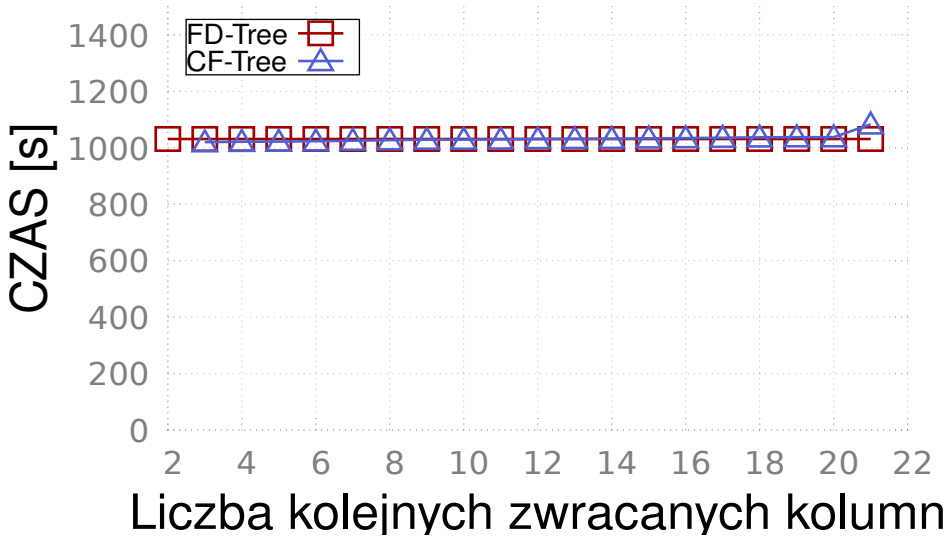


Rysunek 6.27: Czas wykonania zestawu kwerend ZR_B
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)

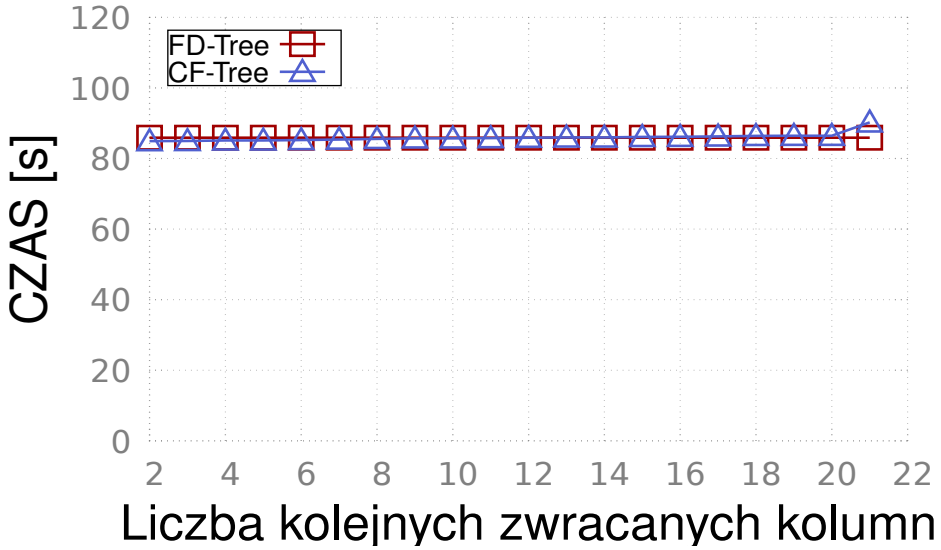
W poprzedniej serii eksperymentów struktura FBDSM osiągnęła najgorszy czas obsługi kwerend. Zarówno wyszukiwanie punktowe jak i wyszukiwanie zakresu o małej selektywności wykonywało się dłużej od indeksowania CFT oraz od wierszowej implementacji drzewa FD. Również w tej serii FBDSM prezentowała najsłabsze osiągi. Aby lepiej zobrazować różnice pomiędzy FD a CFT, na wykresach nie umieściliśmy wyników struktury FBDSM. Mimo iż różne modele dysków SSD osiągały różne czasy (ponieważ mają różny czas wykonania odczytu i zapisu), to jednak charakterystyka obserwowalna na wykresach była dokładnie taka sama. Z tego względu w pracy umieściliśmy tylko wyniki dla najszybszego dysku Intel DCP4511.

Rysunki: 6.26, 6.27, 6.28, 6.29 przedstawiają całkowity czas potrzebny do wykonania zestawów odpowiednio ZR_A, ZR_B, ZR_C i ZR_D ze względu na liczbę zwracanych kolejnych kolumn z tabeli Klient (717B). Do eksperymentów użyto dysku Intel DSP4511, a selektywność ustawiono na 1%. Tabela 6.11 zawiera dokładnie wyniki tych eksperymentów. Zestaw ZR_A nastawiony jest na dużą liczbę wyszukiwań, przy małej liczbie modyfikacji tabeli. Przy takim użyciu bazy, preferowanym sposobem przechowywania danych jest kolumnowe ułożenie rekordów. Dzięki temu podczas wyszukiwania wczytujemy tylko potrzebne fragmenty rekordu. Wyniki eksperymentów potwierdzają tę tezę. Początkowo gdy użytkownik potrzebuje tylko 2 kolumny, czas wykonania kwerendy przez CF-Tree jest ponad 30 krotnie mniejszy niż oryginalnego drzewa FD. Oczywiście używając indeksu CFT im więcej kolumn musimy wczytać, tym czas potrzebny na wykonanie zapytania jest większy. Jednak zawsze kolumnowy indeks osiąga lepsze wyniki niż drzewo FD. Nawet gdy musimy wczytać cały rekord, obie struktury wykonują zestaw kwerend w podobnym czasie.
Zestaw ZR_B nastawiony jest na modyfikacje tabeli, zawiera tylko 5 wyszukiwań i 200tys. operacji modyfikacji (dodawania i usuwania). Ze względu na charakterystykę tego zestawu, czas obsługi zapytań w momencie gdy zwracamy ponad 12 kolumn jest bardzo podobny dla obu struktur. Poniżej tej liczby zwracanych kolumn, CFT osiąga wyniki około 10% lepsze od drzewa FD. Wynika to z faktu, iż czas uzyskany przez wyszukiwania jest

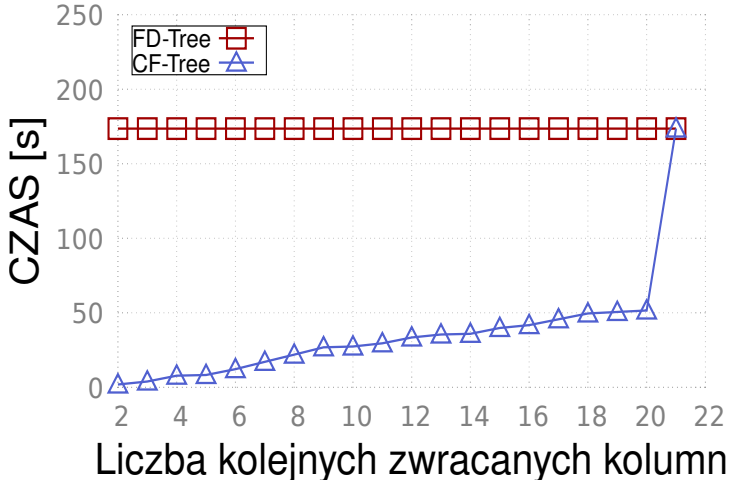
tłumiony przez modyfikacje, które zajmują więcej czasu, ze względu na charakterystykę dysku SSD. Warto również zauważyć, że w poprzednich eksperymentach, czas potrzebny na wykonanie kwerend podczas zwracania całego rekordu w zapytaniu był podobny dla obu indeksów. Tym razem oryginalne drzewo FD osiągnęło o 15% lepszy rezultat od kolumnowego indeksu CF-Tree.



Rysunek 6.28: Czas wykonania zestawu kwerend ZR_C
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)



Rysunek 6.29: Czas wykonania zestawu kwerend ZR_D
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)



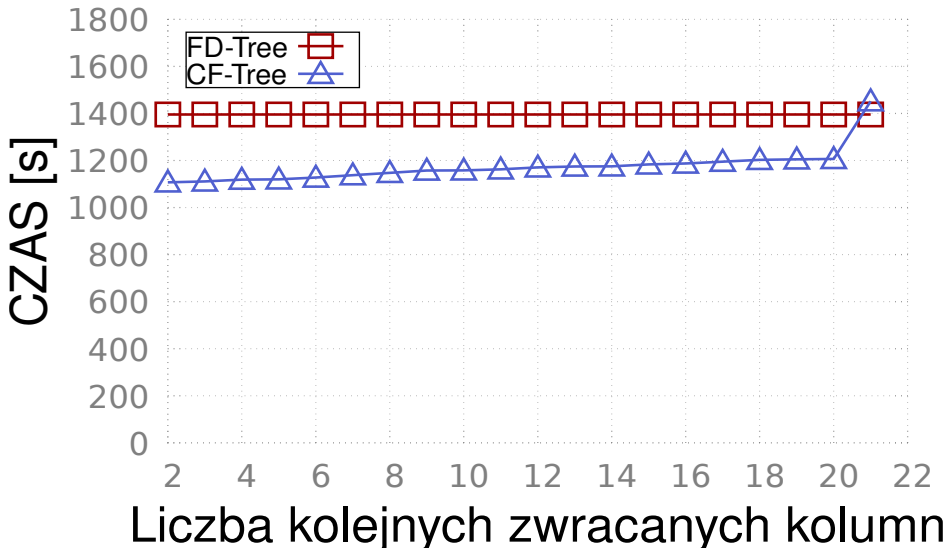
Rysunek 6.30: Czas wykonania zestawu kwerend ZR_A
Selektywność 5%
Dysk: Intel DCP4511
Tabela: Klient (719B)

	ZR_A		ZR_B		ZR_C		ZR_D	
Liczba zwracanych kolumn (bajtów)	**FD**	**CFT**	**FD**	**CFT**	**FD**	**CFT**	**FD**	**CFT**
2 (8B)	34,7s	1s	2.39s	1,99s	1030s	1019s	86s	84s
3 (16B)	34,7s	1s	2.39s	2,0s	1030s	1019s	86s	84s
4 (32B)	34,7s	1,56s	2.39s	2,02s	1030s	1021s	86s	85s
5 (34B)	34,7s	1,66s	2.39s	2,03s	1030s	1021s	86s	85s
6 (50B)	34,7s	2,44s	2.39s	2,05s	1030s	1022s	86s	85s
7 (70B)	34,7s	3,42s	2.39s	2,07s	1030s	1024s	86s	85s
8 (90B)	34,7s	4,41s	2.39s	2,09s	1030s	1026s	86s	85s
9 (110B)	34,7s	5,38s	2.39s	2,12s	1030s	1028s	86s	85s
10 (112B)	34,7s	5,47s	2.39s	2.12s	1030s	1028s	86s	85s
11 (121B)	34,7s	5,91s	2.39s	2,13s	1030s	1029s	86s	85s
12 (137B)	34,7s	6,7s	2.39s	2,15s	1030s	1030s	86s	85s
13 (153B)	34,7s	7.09s	2.39s	2,16s	1030s	1031s	86s	85s
14 (155B)	34,7s	7,19s	2.39s	2,17s	1030s	1031s	86s	85s
15 (171B)	34,7s	7,97s	2.39s	2,19s	1030s	1033s	86s	86s
16 (179B)	34,7s	8,36s	2.39s	2,2s	1030s	1033s	86s	86s
17 (195B)	34,7s	9,14s	2.39s	2,21s	1030s	1035s	86s	86s
18 (211B)	34,7s	9,92s	2.39s	2,23s	1030s	1036s	86s	86s
19 (215B)	34,7s	10,12s	2.39s	2,24s	1030s	1037s	86s	86s
20 (219B)	34,7s	10,32s	2.39s	2,24s	1030s	1037s	86s	86s
21 (719B)	34,7s	34,73s	2.39s	2,86s	1030s	1042s	86s	88s

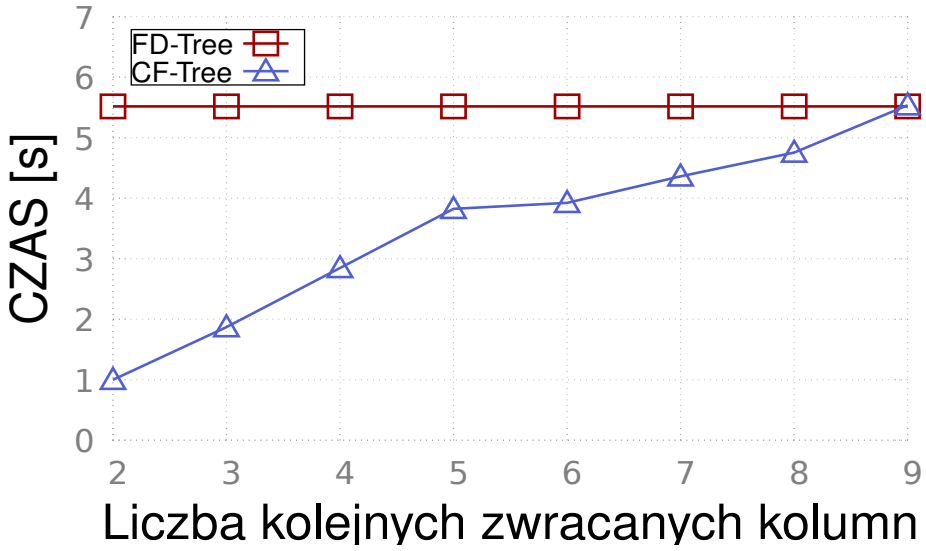
Tabela 6.11: Dokładnie wyniki eksperymentów dla zestawów rozszerzonych
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)

Cechy charakterystyczne zestawów ZR_C i ZR_D są dość podobne. Tak jak poprzedni ZR_B, obydwa zestawy nastawione są na modyfikacje tabeli. Jednak tym razem te dysproporcje są o wiele większe. Zestaw ZR_C zawiera 20 wyszukiwań i aż 11 milionów modyfikacji. Zestaw ZR_D składa się z 10 wyszukiwań i 1 milion modyfikacji. Chociaż wyniki obu tych eksperymentów różnią się całkowitym czasem wykonania, to jednak posiadają ten sam trend i charakterystykę. W obu przypadkach modyfikacje tabel wykonują się tak długo, że czas potrzebny na wyszukiwania jest pomijalny. Warto jednak zauważyć, że kolumnowe indeksy są głównie zoptymalizowane pod wyszukiwania rekordów. Nasza zaproponowana struktura CFT nie tylko dobrze radzi sobie z zestawami, gdzie dominują odczyty danych, ale również z modyfikacjami tabeli. Czas uzyskany podczas obu zestawów jest bardzo podobny (nie różni się więcej niż 5%) dla obu indeksów. Taki wynik jest wręcz imponujący, z tego możemy wywnioskować, że CFT jest dobrym wyborem niezależnie od charakterystyki kwerend wykonujących się na bazie danych.

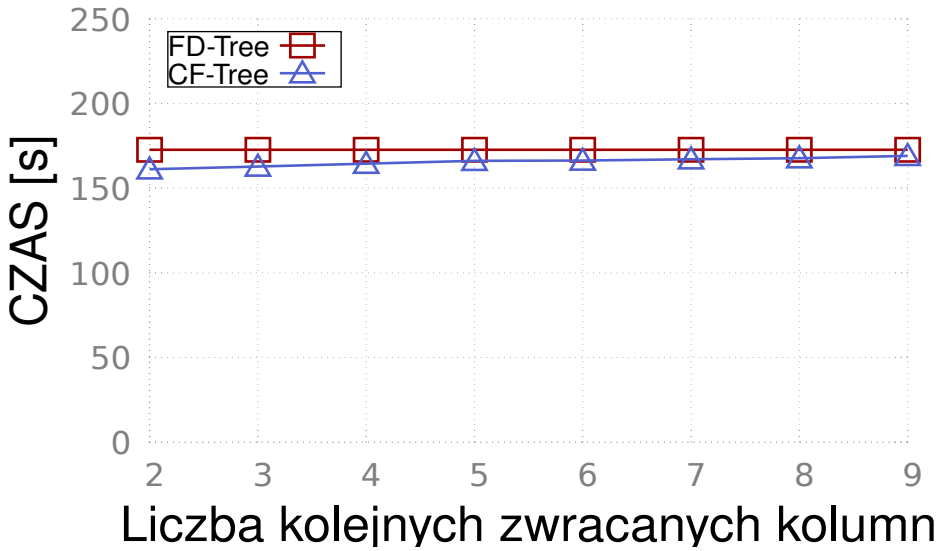
W kolejnej serii eksperymentów sprawdziliśmy jak indeksy radzą sobie z zapytaniami o większej selektywności. Im więcej danych mamy do wczytania, tym więcej zyskujemy na kolumnowym ułożeniu danych, ponieważ pomijamy więcej pamięci, którą wierszowa implementacja musi odczytać. Rysunki 6.30 oraz 6.31 przedstawiają wyniki tej serii eksperymentów. Tym razem przedstawiliśmy tylko zestawy ZR_A o ZR_C, czyli zestawy o skrajnych charakterystykach. Wyniki jakie możemy zaobserwować potwierdzają nasza hipotezę. Przy znaczniej większej niż poprzednio selektywności (równej 5%), kolumnowy indeks CF-Tree osiąga lepsze czasy nie tylko w zestawie ZR_A, ale także w ZR_C. Dzieje się tak ponieważ, przy dużej selektywności operacja wyszukiwania nie jest już kosztem pomijalnym. Tym razem zysk uzyskany w operacji wyszukiwania znacząco wpływa na całkowity czas wykonania kwerend, co widoczne jest na omawianych wykresach.



Rysunek 6.31: Czas wykonania zestawu kwerend ZR_C
Selektywność 5%
Dysk: Intel DCP4511
Tabela: Klient (719B)



Rysunek 6.32: Czas wykonania zestawu kwerend ZR_A
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Sklep (113B)



Rysunek 6.33: Czas wykonania zestawu kwerend ZR_C
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Sklep (113B)

Ostatnia seria eksperymentów została wykonania na mniejszej tabeli - Sklep. Rekord tej tabeli ma rozmiar tylko 113B, w przeciwieństwie do poprzedniej tabeli Klient o rozmiarze rekordu aż 719B. Celem tej serii jest zbadanie wpływu charakterystyki tabeli (rozmiar rekordu, liczba kolumn) na różnicę pomiędzy wynikami osiągniętymi przy użyciu kolumnowego indeksu i wierszowego indeksu. Rysunki 6.32 i 6.33 przedstawiają wyniki ostatniej serii eksperymentów dla odpowiednio zestawów ZR_A i ZR_C. Selektywność tym razem była ustawiona z powrotem na 1%. Mniejsza tabela, zawierająca mniej

kolumn, to mniejszy zysk z pominięcia fragmentu rekordu. Widzimy to na prezentowanych wykresach. Przy wykonywaniu zestawu ZR_A, kolumnowy indeks CFT osiągał ponad 30 krotnie lepszy czas od indeksu FD w przypadku, gdy użytkownik prosił o dwie kolumny z tabeli Klient. Tym razem ten zysk jest tylko 6 krotny i zmniejsza się znacznie szybciej niż w przypadku poprzednich kwerend. Charakterystyka wykresu 6.33 jest taka sama jak charakterystyka wykresu 6.28. W obu przypadkach czas wykonywania kwerend z wykorzystaniem obu struktur jest bardzo podobny. Patrząc na charakterystykę obu wykresów, możemy stwierdzić, że rozmiar rekordu nie miał dużego wpływu na czas wykonania zestawu kwerend, ponieważ zestaw nastawiony jest na modyfikacje bazy a nie na odczyty. Operacje dodawania i usuwania przeprowadzana są podczas reorganizacji drzew na całych rekordach, dlatego liczba kolumn obsługiwana przez kwerendy nie ma wpływu na ten zestaw.

Podsumowując nowa struktura CF-Tree dzięki kolumnowemu ułożeniu danych wykonuje wyszukiwania znacznie szybciej niż oryginalna wierszowa implementacja drzewa FD. Niestety podczas wyszukiwania punktowego, CFT musi wykonać dodatkową procedurę scalania atrybutów w rekord. Chociaż algorytm posiada logarytmiczny czas, to znacząco wpływa na pogorszenie wyników osiąganych przez CFT względem FD nawet w przypadku zwracania niepełnego rekordu. Całkiem inną sytuację indeks prezentuje podczas wyszukiwania całego zakresu, wtedy algorytm scalania wykonywany jest tylko raz, przez co ma praktycznie zerowy wpływ na czas obsługi kwerendy. W zapytaniach potrzebujących mały podzbiór atrybutów nowy indeks CF-Tree osiąga nawet 30 krotnie lepszy czas od drzewa FD. Oczywiście przewaga uzależniona jest od liczby zwracanych kolumn, selektywności zapytania oraz rozmiaru rekordu. Dodatkowo warto zauważyć, że CF-Tree radzi sobie świetnie nie tylko w zestawach nastawionych na wyszukiwania, ale również osiąga porównywalny czas względem wierszowego FD w zestawach modyfikujących bazę danych. Przeprowadzona analiza oraz eksperymenty pokazują praktyczną przydatność nowej, zaproponowanej struktury CF-Tree dla baz danych przechowywanych na dyskach SSD.

6.6 Indeksowanie częściowe

Stworzenie idealnego zbioru indeksów zawierającego wszystkie potrzebne struktury to bardzo trudne zadanie, przed którym stają administratorzy baz danych. Z tego więc powodu dąży się do automatyzacji procesu tworzenia indeksów [53], [54], [55]. Stosuje się podejścia, które próbują dostosowywać porządek bazy danych i kolejne indeksy wraz ze zmianą kwerend. Kolumny po których musimy sortować dane podczas zapytania są monitorowane i analizowane. Dzięki temu silnik bazy stwierdza, czy potrzebny jest nowy indeks oraz czy istniejący indeks przestał być już używany. W przeciągu kilku ostatnich lat pojawiło się kolejne podejście do automatyzacji procesu indeksowania. W [56] [57] zaproponowano Indeks Cracking jako nową metodę automatycznego, iteracyjnego tworzenia indeksu. W wyniku działania kwerendy część tabeli jest reorganizowana, aby dostosować się do wzorca obecnego zapytania. Proces ten próbuje zamortyzować koszt tworzenia indeksu dzieląc go na wiele części. Indeks Cracking używa metody podziału danych na partycje (ang. partition) względem elementów z zakresu zapytania i działa podobnie do sortowania szybkiego (ang. quicksort) [58]. Chociaż Indeks Cracking jest bardzo dobrym podejściem do automatyzacji tworzenia indeksu, to jednak metoda ta ma kilka wad. Przede wszystkim częste reorganizacje na małych częściach tabeli są stosunkowo wolne w przypadku gdy używamy dysków SSD. Po drugie, szybkość tworzenia indeksu zależna jest

od wzorca kwerend. Tak jak w sortowaniu szybkim, złożoność obliczeniowa zależna jest od wyboru elementu rozdzielającego partycję (ang. pivot). Kolejna metoda automatycznego tworzenia indeksu podczas kwerend została zaproponowana w [13]. Adaptive Merging w przeciwieństwie do Indeks Cracking korzysta z algorytmu scalania posortowanych list tak jak sortowanie przez scalanie (ang. mergesort) [62]. Gdy po raz pierwszy musimy wyszukać elementy biorąc pod uwagę wartości z danej kolumny, kopiujemy tabelę wczytując tyle danych na raz, ile mieści się w buforze. Następnie sortujemy wartości w obrębie danej partycji. W kolejnych zapytaniach będziemy usuwać dane z posortowanych partycji i tworzyć nową. W ten sposób uzyskamy po pewnym czasie zbiór partycji, które nie tylko są posortowane wewnętrznie, ale także tworzą posortowany ciąg. Obie te metody zostały dokładnie omówione w rozdziale **2**. Ze względu na schemat działania obu algorytmów Indeks Cracking stosuje się dla baz danych zapisanych w szybkiej pamięci operacyjnej komputera, a Adaptive Merging w sytuacji, gdy tabela przechowywana jest na pamięciach blokowych takich jak dyski HDD czy dyski SDD. Niestety w oryginalnej wersji Adaptive Merging nie brano pod uwagę różnic pomiędzy dyskami HDD a innymi pamięciami blokowymi jak nowsze modele dysków SDD. Z tego powodu zaproponowaliśmy nowy system do częściowego indeksowania danych, dostosowany w pełni do charakterystyki pamięci flash i dysków SSD - Lazy Adaptive Merging [4].

6.7 System Lazy Adaptive Merging

Lazy Adaptive Merging (LAM) [4] to kolejne iteracyjne podejście tworzenia indeksu częściowego jako efekt uboczny każdej kwerendy. Jest dostosowany do blokowej pamięci flash i dysków SSD oraz posiada wszystkie cechy dobrego systemu do przechowywania danych na tej pamięci. Należą do nich:

- **Buforowanie** - dyski SSD najszybciej przeprowadzają zapis sekwencyjny, ponieważ mogą zapisać dane równolegle na kilku kościach w tym samym czasie. Tak samo jak w przypadku baz danych zapisanych w RAM (ang. in-memory database) ważna jest pamięć podręczna procesora (ang. cache), tak dla dysków SSD ważne jest umiejętne buforowanie danych w pamięci operacyjnej komputera (RAM). Zamiast zapisywać każdą daną pojedynczo, co wywoła wiele operacji kasowań dużych bloków, lepiej jest wprowadzić mały bufor o rozmiarze kilku kilobajtów tak aby wykonywać zapis pełnych stron.

- **Wyrównanie (ang. alignment)** - flash typu NAND, który używany jest w dyskach SSD, potrafi odczytywać i zapisywać tylko pełne strony. Tak wiec jeśli chcemy zapisać jeden bajt, musimy zapisać całą stronę. Jeszcze gorzej wypada kasowanie bloku. Operacja ta możliwa jest do wykonania na pełnym bloku (32-64 strony). Dlatego ważne jest, aby zapisywać dane obok siebie, by móc jedną operacją wczytać wiele potrzebnych danych bez marnowania czasu na odczyt przypadkowej zawartości strony.

- **Leniwe usuwanie** - operacja kasowania bloku jest najwolniejszą operacją na pamięci flash. Ze względu na pewne fizyczne ograniczenia opisane w rozdziale **3** musimy skasować cały blok przed ponownym zapisem danych. Zatem jeśli chcemy zmienić wartość na danej stronie, musimy wczytać cały blok do pamięci RAM, zaktualizować dane, usunąć blok, a następnie jeszcze raz go zapisać. Cały proces jest długi i kosztowny, a więc warto go optymalizować. Dobrym sposobem jest wprowadzenie

dzienników pozwalających odroczyć wymazanie bloku w czasie, a także agregować potrzeby wykonania tej operacji.

- **Zbieranie informacji** - często można gromadzić metadane, aby móc zoptymalizować pewne operacje. Gdy wiemy, że tabela częściej używana jest do odczytu niż zapisu, wówczas warto tak przeorganizować dane, aby przyspieszyć wyszukiwanie kosztem wstawiania. Nie warto jednak przesadzać z wielkościami tych danych, tak aby nie zdominowały kosztu operacji. Najlepiej zapisywać je w szybkiej pamięci RAM, a podczas awarii systemu odzyskiwać je podczas ponownego uruchamiania.

Podczas opracowywania struktury danych do indeksowania nie musimy skupiać się na wyżej wymienionych cechach systemu, ponieważ nową strukturę umieszczamy wewnątrz silnika bazy danych, który posiada te i wiele innych mechanizmów przyspieszających pracę na zgromadzonych danych. Warto jednak zauważyć, że LAM nie jest strukturą danych, ale całym system, który wspiera każdą operację podczas tworzenia kolejnych części indeksu. Nie może on skorzystać z gotowych rozwiązań silnika baz danych, ponieważ wprowadza nowe, nieznane dla silnika operacje. Z tego powodu, próba implementacji dobrych cech systemu pracującego na dyskach SSD jest ważnym elementem zaproponowanego algorytmu.

6.7.1 Struktura LAM

Lazy Adaptive Merging w przeciwieństwie do klasycznego Adaptive Merging nie usuwa danych podczas tworzenia nowej partycji. W zamian za to zapisuje zużycie bloku w specjalnej tablicy, aby móc analizować liczbę nieusuniętych danych. Nadaje się do zapisu wierszowego jak i kolumnowego, tworząc indeks algorytmem wstawiania zbiorczego (ang. bulkload). Zatem wszystko zależy od wybranej struktury do indeksowania. Jeśli struktura posiada kolumnowy zapis tabeli, wówczas LAM w pełni to wykorzysta tworząc częściowy kolumnowy indeks. Nasz nowy system potrafi także współpracować ze strukturami używanymi przez indeksy zoptymalizowane dla pamięci flash takie jak LA ([26]) czy FD ([25]). System Lazy Adaptive Merging korzysta z następujących struktur danych:

1. **Partycja** - podstawowa struktura przechowująca zbiór posortowanych danych. Wielkość partycji zawsze jest wielokrotnością rozmiaru bloku kości flash, aby zminimalizować narzut związany z zapisem, odczytem, a także usuwaniem danych. Podczas pierwszej kwerendy dane wczytywane są do bufora i sortowane. Następnie zapisywane są na dysku SSD. Kolejne partycje tworzone są podczas procesu scalania kilku partycji, aby zmniejszyć liczbę używanych bloków. Dane są posortowane wewnątrz każdej partycji, a więc możemy wyszukać potrzebny klucz w czasie logarytmicznym używając klasycznego wyszukiwania binarnego. Dodatkowo każda partycja posiada dwa atrybuty, najmniejszą i największą wartość klucza jaka jest w partycji. Atrybuty te zapisane są w pamięci RAM lub w drzewie PB+ [63]. Dzięki nim bez odczytu partycji możemy stwierdzić czy zawiera potrzebne dane czy nie.

2. **Drzewo PB+ (ang. Partitioned B+ tree)** - drzewo służy do zarządzania zbiorem partycji. Struktura jak i algorytmy wykonywane na tym drzewie są bardzo podobne do zwykłego drzewa B+. Każda partycja posiada klucz sztuczny, według którego jest przechowywana w drzewie. Mając drzewo z adresami partycji, możemy łatwo nimi manipulować tzn. dodawać nowe i usuwać stare. Wyszukiwanie danych może odbywać się równoległe na kilku partycjach. Strukturę najlepiej zapisać w pa-

mięci RAM, ponieważ można ją łatwo zbudować ze zbioru partycji zapisanych na nieulotnej pamięci. Pełny opis struktury można znaleźć w [63].

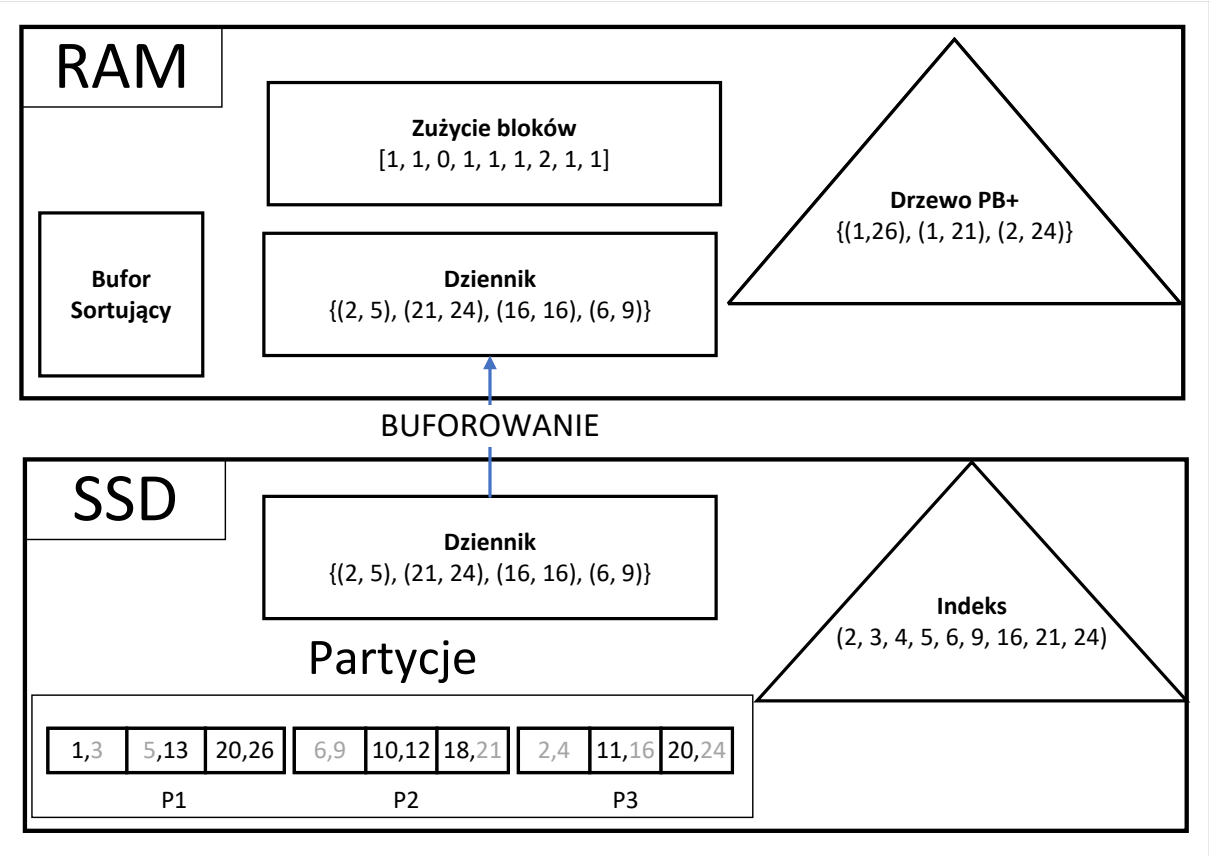
3. **Bufor do sortowania** - fragment szybkiej pamięci RAM o ustalonym rozmiarze BS (ang. Buffor Size) używany do sortowania danych przed zapisem do partycji. Im większy bufor, tym więcej danych można posortować za jednym razem. Zatem im większe BS tym mniej będzie partycji, czyli szybciej będziemy mogli wyszukiwać dane podczas zapytania.

4. **Tablica zużycia danych (ang. usage)** - tablica o rozmiarze liczby bloków potrzebnych do zapisania tabeli lub kolumny (w zależności od sposobu zapisu bazy). Każdy blok ma swoje własne pole w tablicy, w którym zapisana jest liczba użytych danych (zapisanych już w indeksie). Przykładowo, jeśli blok piąty ma rozmiar 32 stron (na każdej stronie mieści 5 rekordów) i 10 danych zapisanych jest już w indeksie to $usage[5] = 10$. Tablica ta jest najważniejszą strukturą używaną przez LAM, ponieważ dzięki niej algorytm może podjąć decyzję o reorganizacji i usunięciu zduplikowanych danych. Reorganizacja przeprowadzana jest wtedy, gdy liczbę bloków można zredukować o wartość większą lub równą parametrowi MT (ang. Merge Treshold). Struktura zapisana jest w szybkiej pamięci RAM. Aby uniknąć utraty informacji podczas braku zasilania, możliwe jest wprowadzenie cyklicznego zapisu tych danych na dysk.

5. **Indeks** - do indeksowania można użyć dowolnej struktury, która jest dostosowana do pamięci flash oraz posiada wszystkie cechy wymagane nasz system.

6. **Dziennik (ang. journal)** - ponieważ nie usuwamy danych natychmiastowo podczas wykonywania kwerendy, musimy stwierdzić skąd czytamy dane: z indeksu czy z partycji. Samo wyszukiwanie w indeksie jest stosunkowo tanie, jednak skan partycji jest kosztowny. Aby zredukować te koszty, wprowadzony został dziennik. Jest to lista par uporządkowanych. Każda krotka zawiera zakres kluczy, które znajdują się w indeksie, czy to poprzez skopiowanie ich z partycji do indeksu czy poprzez wstawienie nowych danych bezpośrednio do niego. Dodatkowo zapisujemy tam również dane, które usunięto z partycji. Dzięki temu podczas wyszukiwania możemy stwierdzić, że rekord znajduje się w indeksie. Za pomocą szybkiego wyszukiwania w strukturze indeksu dowiemy się, czy rekord o podanym kluczu został usunięty. Taka operacja jest o wiele szybsza od skanowania partycji w celu stwierdzenia nieobecności usuniętej danej. Przykładowo jeśli podczas pierwszego zapytania zapiszemy klucze z zakresu $b - g$, a podczas drugiego $a - c, u - z$, to dziennik będzie zawierał krotki: $\{(b, g), (a, b), (u, z)\}$. Ze względu na informacje odnośnie usuniętych danych, których nie da się odzyskać po restarcie systemu, dziennik zapisany jest na dysku oraz częściowo buforowany w pamięci RAM, aby przyspieszyć jego odczyt.

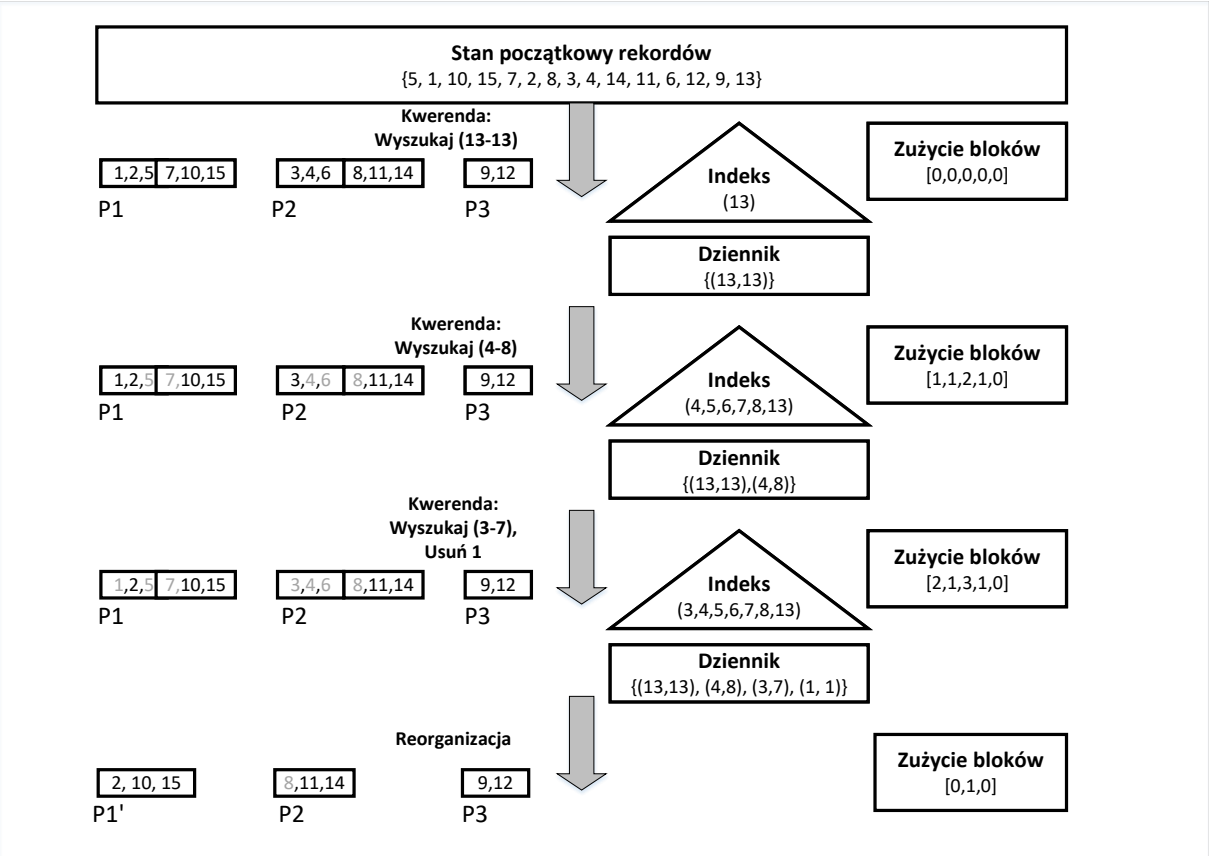
Rysunek 6.34 przedstawia przykładowy stan procesu tworzenia indeksu częściowego przez LAM. Dla uproszczenia załóżmy, że tabela ma 18 rekordów. Kolumna, po której zaczynamy sortować, ma układ w pamięci: (1, 26, 13, 5, 3, 20, 21, 10, 9, 12, 18, 6, 2, 4, 16, 20, 24, 11). Bufor może pomieścić tylko 6 rekordów na raz. Sytuacja przedstawiona na rysunku ma miejsce po wykonaniu czterech kwerend: 2 - 5, 21 - 24, 16 - 16, 6 - 9. Wielkość partycji w tym momencie ograniczona jest przez wielkość bufora, w którym odbywa się sortowanie. Na rysunku przedstawione są trzy partycje P1, P2, P3, każda o pojemności trzech bloków. Każdy blok może pomieścić dwa rekordy. Zamiast całego rekordu przedstawione zostały wartości liczbowe, które przedstawiają wartość klucza. Rekordy podczas



Rysunek 6.34: Przykład struktury LAM

pierwszego zapytania są sortowane w buforze i zapisywane do partycji. Dlatego dane wewnątrz każdej partycji są posortowane. Dostęp do partycji odbywa się poprzez drzewo PB+. Kolor szary symbolizuje dane wstawione do indeksu, które jednak ze względu na minimalizację liczby modyfikacji bloków dysku SSD nie zostały jeszcze usunięte. Czytając dziennik możemy łatwo zauważyć, jakie dane zostały przepisane do indeksu. Zatem jeśli kolejna kwerenda będzie żądała wartości z zakresu (7 - 11), to analizując dziennik stwierdzimy, że (7-9) wczytamy z indeksu a (10 - 11) z partycji. Tablica zużycia bloków monitoruje liczbę danych, które czekają na usunięcie. Przykładowo blok pierwszy z partycji pierwszej (P1) zawiera jedną przepisaną daną (w tym przykładzie strona jest w stanie pomieścić tylko jeden rekord), stąd $usage[0] = 1$, trzeci blok nie ma żadnej danej wstawionej do indeksu, dlatego $usage[3] = 0$. Dzięki tej tablicy algorytm rozwiązujący problem ciągłego pakowania może znaleźć sposób na takie połączenie bloków w inną partycję, które zmniejsza znacząco liczbę bloków. Wtedy następuje reorganizacja: usuwane są stare partycje, a dodawane nowe. Dzięki strukturze PB+ operacje są proste do wykonania i nie zaburzają pracy algorytmu.

Rysunek 6.35 przedstawia cztery kroki systemu Lazy Adaptive Merging podczas tworzenia indeksu częściowego. Dla uproszczenia załóżmy, że tabela ma 15 rekordów, a każda liczba przedstawia wartość atrybutu, który będzie nowym kluczem. Kolumna, po której zaczynamy sortować, ma układ w pamięci: (5, 1, 10, 15, 7, 2, 8, 3, 4, 14, 11, 6, 12, 9, 13). Strona dysku SSD może pomieścić tylko jeden rekord, a blok maksymalnie 3 rekordy. Bufor może mieć rozmiar 6 rekordów, a więc tylko tyle możemy sortować jednocześnie. Zatem partycja może składać się z nie więcej niż z 6 rekordów (2 bloków). Aby upro-



Rysunek 6.35: Przykładowy proces tworzenia częściowego indeksu z użyciem systemu LAM

ścić rysunek, pominięte zostało drzewo PB+. Pamiętajmy jednak, że zapisane w nim są minimalne i maksymalne wartości kluczy każdej partycji oraz fizyczny adres na dysku. Ustalmy także parametr $MT = 2$ (ang. Merge Treshold). W związku z tym możemy przeprowadzić reorganizację dopiero wtedy, gdy istnieje możliwość zredukowania liczby bloków o przynajmniej dwa. Na początku dostajemy kwerendę z zakresu (13-13). Dane wczytywane są do bufora i sortowane, następnie zapisywane w partycjach. Sam zakres kwerendy dodawany jest do indeksu. Z tego powodu w indeksie zapisaliśmy klucz 13, a także utworzyliśmy pierwszy wpis w dzienniku: parę (13,13). Kolejno mamy zapytanie o zakresie (4 - 8). Po pierwsze czytamy nasz dziennik aby sprawdzić, czy zakres zapytania pokrywa się z kluczami zapisanymi już w indeksie. Niestety wartość 13 jest poza zakresem (4 - 8), a więc musimy wczytać wszystkie wartości z partycji. Dzięki atrybutom partycji zapisanych w drzewie PB+ stwierdzamy, że partycja $P3$ nie zawiera potrzebnych danych, więc nie będziemy jej wczytywać. Używając informacji zapisanych w szybkiej pamięci RAM określimy, które strony z dysku musimy wczytać. Z partycji $P1$ kopiujemy 5 i 7, a z partycji $P2$ kopiujemy 4,6 oraz 8. Nie usuwamy danych z dysku, aby uniknąć niepotrzebnego wymazywania. W zamian wstawiamy dane do indeksu i zapisujemy przedział do dziennika. W naszym przypadku będzie to para (4,8). Aby zobrazować użyte lecz nieusunięte dane, na rysunku takie klucze zostały zaznaczone szarym kolorem. Aktualizujemy także zużycie bloków, z których coś skopiowaliśmy. Przykładowo, ponieważ z bloku pierwszego z partycji $P1$ skopiowaliśmy 5, to $usage[1] = 1$. Ostatnim krokiem jest sprawdzenie, czy możliwe jest takie połączenie bloków w inne partycje, które zmniejszy ich liczbę o co najmniej 2. Niestety okazuje się, że takie nie istnieje. Po wykonaniu

wszystkich powyższych kroków, możemy zwrócić użytkownikowi wynik jego kwerendy. Następnie mamy sytuację, w której dostajemy zapytanie o zakresie (3 - 7). Jak poprzednio, czytamy najpierw dziennik. Widzimy, że (4-7) możemy wyszukać w indeksie. Resztę wczytamy z partycji. Kolejny raz stwierdzamy, że $P3$ nie zawiera potrzebnych danych, więc nie będziemy jej wczytywać. Z $P2$ kopiujemy klucz 3. Wstawiamy klucz do indeksu i zapisujemy kolejną parę (3, 7) do dziennika. Usuwamy także dane z kluczem 1. Znajduje się on w partycji $P1$. Zamiast fizycznego usuwania, zaznaczamy go jako przepisany do indeksu. Dodajemy wpis do dziennika: (1, 1). Aktualizujemy tablicę użycia bloków analogicznie jak poprzednio oraz sprawdzamy, czy opłaca się wykonać reorganizację. Okazuje się, że tak. Możemy połączyć blok pierwszy z drugim oraz w pełni usunąć trzeci. Dzięki temu liczba bloków zmniejszy się o 2. Zatem wczytujemy wymienione bloki i za pomocą dziennika stwierdzamy, które dane można usunąć. Dolna część rysunku pokazuje sytuację po scaleniu i usunięciu bloków. Zatem nadal mamy 3 partycje (w miejsce $P1$, stworzyliśmy $P1'$, z $P2$ usunęliśmy blok), przy czym każda z nich ma tylko jeden blok. Zmieniona została także tablica użycia bloków, ponieważ zmienił się ich układ. Reorganizacja została zakończona, a więc możemy zwrócić wynik użytkownikowi.

Omówiony przykład bardzo dobrze pokazuje zysk wynikający z użycia naszego nowego systemu. Podczas rozpatrywanych zapytań, klasyczne podejście wykonałby aż 6 usunięć bloków oraz 8 zapisów potrzebnych, aby na nowo przepisać dane z usuwanego bloku do nowego. Zaproponowany algorytm wykonał tylko 3 usuwania i 3 przepisania danych wynikające z usunięcia bloku. Warto zauważyć, że liczba odczytów jest podobna. Tak więc optymalizacja liczby kasowań nie wpłynęła na szybkość wyszukiwań. Kosztem dodatkowej pamięci, która zajęta jest do czasu reorganizacji, skróciliśmy czas wykonywania kwerendy.

6.7.2 Procedury systemu LAM

Wstawianie

W przeciwieństwie do Indeks Cracking i Adaptive Merging wstawianie w zaproponowanym sposobie tworzenia indeksu jest bardzo proste. Ponieważ używamy gotowej struktury danych do indeksowania rekordów, wstawianie odbywa się tak jak w używanej strukturze indeksowej. Nie potrzeba więc dodawać wartości do posortowanych partycji i zaburzać tym samym porządek danych. Ponieważ dane które wstawiamy dodajemy bezpośrednio do struktury indeksu, zapisujemy ten fakt w naszym dzienniku. Dzięki temu podczas wyszukiwania będziemy wiedzieli, że należy rekordu szukać w indeksie a nie w partycjach.

Wyszukiwanie

Wyszukiwanie to najważniejszą operacją w procesie tworzenia indeksu częściowego. To właśnie podczas wyszukiwania dokładamy kolejne rekordy do indeksu zmieniając przy tym układ danych. Samo wyszukiwanie jest dość proste. Gdy przychodzi kwerenda od użytkownika, czytamy dziennik indeksu i próbujemy określić, które dane możemy wczytać z gotowego indeksu, a które z partycji. Ponieważ dziennik to tylko lista zakresów kluczy przepisanych lub dodanych do indeksu, musimy odczytać całą listę. Dlatego trzymamy go w pamięci RAM jako struktura posortowanego drzewa, aby przyspieszyć ten proces.

Oczywiście najlepszym przypadkiem jest zapytanie, które możemy obsłużyć czytając tylko indeks. Gdy jednak tak nie jest, musimy wczytać dane z partycji. Odczytujemy wartość klucza uwzględniając klucz sztuczny w drzewie PB+ i znajdujemy partycję która może posiadać tę daną. Przydatność partycji określamy na podstawie jej atrybutów. W fazie przygotowania zbieramy wszystkie potrzebne informacje zapisane w pamięci komputera, a następnie wykonujemy sekwencyjny odczyt na potrzebnych stronach z dysku. To, że klucz zapisany został w dzienniku nie oznacza, że nadal jest w indeksie. Podczas usuwania lub aktualizacji rekordów w indeksie zmieniamy indeks, jednak nie zmieniamy dziennika. Robimy tak, aby przyspieszyć proces wyszukiwania oraz zachować jego poprawność. Gdybyśmy usunęli dane z dziennika, musielibyśmy odszukać go w zbiorze partycji. Ponieważ nie usuwamy danych natychmiastowo, to możliwe że znajdziemy klucz, który miał być usunięty. Podamy wtedy nieprawdziwą informację użytkownikowi. Nie usuwając wpisu, w czasie logarytmicznym wyszukujemy klucz w indeksie lub dochodzimy do wniosku, że dany klucz został usunięty. Chociaż po wczytaniu stron i ich analizie możemy zwrócić wynik użytkownikowi, to nie kończymy tym operacji. Dzięki znajomości zakresu kwerendy możemy dostosować układ danych tak, aby kolejne takie samo zapytanie lub zbliżone do niego było o wiele szybsze. W tym celu wszystkie dane wczytane z partycji sortujemy w buforze i wstawiamy do indeksu. Następnie krotkę opisującą wstawiony zakres zapisujemy w dzienniku. Nie usuwamy jednak danych z partycji. Kosztem dodatkowej pamięci minimalizujemy czas potrzebny na wykonanie operacji wymazania na niepełnym bloku. Zamiast fizycznego usuwania rekordów zapisujemy kolejne zużycie danego bloku w odpowiadającym dla niego polu w tablicy *usage*. Jeśli zachodzi taka potrzeba, wykonujemy reorganizację na zbiorze partycji.

Usuwanie

Aby zminimalizować liczbę operacji usuwania bloków pamięci flash znajdującej się w dysku SSD, wprowadzono mechanizm leniwego usuwania. Gdy wartość zapisana jest w indeksie, wystarczy wykonać potrzebną operację na wybranej do indeksowania strukturze. Gdy rekord zapisany jest w partycji, nie należy go usuwać z dysku. Zamiast tego dodajemy wpis do dziennika, aby podczas wyszukiwania algorytm zdecydował się na odczyt indeksu. W dość szybkim czasie (większość indeksów gwarantuje logarytmiczny czas wyszukiwania) stwierdzimy, że rekordu nie ma w indeksie, a zatem został usunięty. Podczas przeprowadzania tej operacji na partycji musimy także pamiętać o aktualizacji tablicy zużycia bloku, aby móc podjąć decyzję o reorganizacji opisanej w tym rozdziale.

Reorganizacja

Jak już wcześniej wspomnieliśmy, LAM nie usuwa kluczy od razu, ale zapisuje zużycie bloku w specjalnej tablicy i czeka na moment, gdy będzie można zmniejszyć liczbę bloków przynajmniej o wartość parametru MT (ang. Merge Treshold). Tak więc aby wykonać reorganizację, musimy znaleźć taką kombinację połączenia bloków, aby znacząco zmniejszyć ich liczbę. Zauważmy, że dane będą sortowane ponownie, a więc nie musimy używać dokładnie całych pozostałych fragmentów bloku. Możemy je dowolnie porozdzielać między inne partycje. Zatem patrząc na fragmenty nieusuniętych danych jak na przedmioty, na bloki jak na pudełka, dostajemy ciągły problem pakowania (wariant, w którym możemy dzielić przedmioty na ułamki) [189]. W przeciwieństwie do zwykłego problemu pakowania [190], ten możemy rozwiązać bardzo łatwo. Po prostu wczytujemy tyle fragmentów, ile potrzeba do utworzenia nowych, pełnych bloków. Taki wynik uzyskamy, gdy nadamy

każdej danej objętość 1 oraz koszt (wartość) 1. Blok będzie posiadać pojemność równą liczbie danych możliwych do zapisania. Dzięki takiemu przypisaniu otrzymujemy instancję ciągłego problemu pakowania, w którym każda dana ma tą samą wartość, a zatem nie jest w żaden sposób faworyzowana w wyborze. Na końcu wyszukiwania sprawdzamy wynik powyższego algorytmu. Jeśli okaże się, że możemy istotnie zmniejszyć liczbę bloków to przeprowadzamy reorganizację. W tym celu wczytujemy potrzebne bloki do pamięci RAM. Aby znaleźć dane, które mamy usunąć, bierzemy część wspólną danych w bloku i zakresów zapisanych w dzienniku. Pozostałe dane sortujemy i tworzymy nowe partycje (zakładamy, że możemy posortować tylko tyle danych ile pomieści sortujący bufor). Ponieważ zmienił się układ bloków, aktualizujemy także tablicę zużyć. Stare partycje usuwamy z drzewa PB+ i dodajemy na ich miejsce nowe. Chociaż już usunęliśmy dane z partycji, to nie zmieniamy dziennika, gdyż służy on nie tylko do określania użytych danych ale również dba o poprawność samego wyszukiwania.

Łatwo zauważyć, że algorytm nie sprawdzi się, gdy dane są wczytywane równomiernie z całej tabeli. W takim przypadku, może się okazać, że potrafimy zmniejszyć liczbę bloków wczytując je wszystkie lub znaczną większość. Aby tego uniknąć, wystarczy wprowadzić dodatkowy warunek. Jednym z nich może być liczba bloków użyta do spakowania fragmentów danych, drugim może być wielkość zużycia bloku (np. 0.05), poniżej której blok nie jest brany pod uwagę. Oba warunki dobrze sprawdzają się w uniknięciu dużego narzutu czasowego wynikającego z reorganizacji, zwiększając tym samym ilość pamięci potrzebnej na nieużywane dane w partycjach. Wprowadzając maksymalną liczbę bloków użytych do jednorazowej reorganizacji, możemy kontrolować amortyzację kosztów usuwania. W początkowej fazie tworzenia indeksu będziemy wczytywać duże liczby rekordów z nieposortowanych partycji. Z czasem ta liczba będzie spadać, więc czas na początku będzie bardzo duży, później o wiele mniejszy (pamiętajmy, że to właśnie wymazanie bloku jest najbardziej kosztowne). Biorąc na raz pewną ustaloną liczbę bloków, zostawimy zużyte bloki na kolejne kwerendy. Dzięki temu koszt usuwania zostanie zamortyzowany na więcej kwerend. W związku z tym, sytuacja nadmiernego obciążenia systemu i znaczącego wydłużenia się obsługi kwerendy nie nastąpi.

6.7.3 Opis algorytmów

W tej sekcji dokładnie omówimy 2 najważniejsze algorytmy systemu Lazy Adaptive Merging: wyszukiwanie 6.5 oraz reorganizację 6.6. Procedura wyszukiwania 6.5 na wejściu przyjmuje posortowany zbiór klucz, dla którego mamy zwrócić odpowiadający mu zbiór rekordów. W klasycznym indeksie to byłaby jedyna praca takiego algorytmu. W systemach częściowego indeksowania jest inaczej. To właśnie podczas wykonywania procedury wyszukiwania, jako drugi główny krok dodajemy znalezione rekordy do indeksu, jeśli wcześniej ich tam nie było.

Każdy klucz może znajdować się albo w indeksie albo w partycji. Algorytm najpierw przechodzi po każdym kluczu i sprawdza, czy jest on obecny w naszym dzienniku (linie 6-10). Jeśli nie jest, to mamy pewność, że klucz nie znajduje się w indeksie, zatem musimy wyszukać go w zbiorze partycji. Jeśli klucz znajduje się w naszym dzienniku, wówczas został on przepisany z partycji lub dodany bezpośrednio do indeksu. Gdy już określiliśmy, które rekordy należy szukać w indeksie, w linii 12 podejmujemy próbę ich szukania. Pamiętajmy, że nie wszystkie klucze, które znajdują się w dzienniku znajdują się w indeksie. Możliwe że wpis do dziennika został dodany podczas usuwania rekordu z

partycji. Kolejnym krokiem jest przygotowanie listy partycji, które mogą zawierać szukane rekordy. Dzięki atrybutom każdej z partycji zapisanych w drzewie PB+ możemy stworzyć taką listę bez wykonywania zbędnego odczytu z dysku SSD (linie 14-18). Gdy już mamy gotowy zbiór partycji, które mogą zawierać szukane rekordy, równoległe wykonujemy ich odczyt (linie 20-26). Pamiętajmy, że partycje są posortowane wewnętrznie, zatem wyszukiwanie rekordów po kluczu wewnątrz partycji możemy wykonać za pomocą prostego wyszukiwania binarnego (linia 23). Oprócz tego, gdy przeniesiemy rekord, musimy odnotować to w tablicy *usage* (linia 26), gdyż bez tego nie będziemy mogli podjąć poprawnej decyzji o konieczności przeprowadzenia reorganizacji. Gdy wyszukiwanie jest zakończone, następuje dodanie znalezionych w partycjach rekordów do indeksu (linie 30-32). W tym celu najpierw dodajemy fizycznie dane do indeksu wykonując dodawanie zbiorcze (linia 30), a następnie wpisujemy do dziennika informację określającą zakres kluczy, jaki został właśnie wpisany (linie 31-32). Na końcu procedury sprawdzamy czy możliwa jest redukcja liczby bloków dysku SSD o więcej niż parametr MT (ang. Merge Threshold). Jeśli tak, to wykonujemy reorganizację partycji.

Pseudokod 6.6 opisuje proces reorganizacji. Najpierw musimy znaleźć kombinacje użytych częściowo bloków, aby móc zbudować z nich nową partycję. Zatem wykonujemy algorytm prostego wariantu pakowania i w tablicy *toLoad* zapamiętujemy bloki, które mamy użyć (linia 1). Następnie musimy oddzielić dane do usunięcia od tych do przepisania (linie 3-7). Możemy to wykonać dzięki dziennikowi. Jeśli klucz jest w dzienniku to znaczy, że należy go usunąć, ponieważ został skopiowany do indeksu. W przeciwnym przypadku musimy go przepisać do nowej partycji. Ponieważ zmieniamy strukturę LAM usuwając stare bloki, to zerujemy tablicę *usage* (linia 8). Po zgromadzeniu rekordów sortujemy dane i budujemy z nich nową partycję (linie 9-10), którą w ostatnim kroku zapisujemy ją na dysk (linia 11).

6.7.4 Wybór Indeksu dla systemu LAM

W tej sekcji omówimy cechy, jakie musi spełniać indeks, aby mógł on wykorzystać w pełni charakterystykę systemu LAM oraz dysku SSD. W poprzednim podrozdziale omówiliśmy algorytmy wykonywane podczas indeksowania. Zauważmy, że wszystkie oprócz wyszukiwania używają bezpośrednio podstawowych funkcji, jakie dostarcza każda struktura danych czyli: wstawiania pojedynczej danej, usuwania i wyszukiwania. Konstrukcja procedury powiększania indeksu pozwala na dodanie jednocześnie wielu danych, jeśli potrafimy wykonać dodawanie zbiorcze (ang. bulkload). Niestety nie wszystkie struktury posiadają takie algorytmy (min. B+). Dlatego przedstawimy metodę (ang. framework) pozwalającą w łatwy i efektywny sposób dodać dane do drzewa.

Zauważmy, że system LAM dodaje rekordy w bardzo specyficzny sposób. Sam klucz rekordów jest unikatowy w zbiorze wszystkich danych w tabeli. Nowe dane dodawane są bezpośrednio do indeksu za pomocą zwykłego wstawiania pojedynczej danej do struktury. Jednak najwięcej danych nie dodajemy w ten sposób, ale w końcowej fazie wyszukiwania rekordów z podanego zakresu kluczy. Rezultat takiej kwerendy jest zbiorem posortowanych kluczy, który jest rozłączny ze zbiorem kluczy znajdującym się już w drzewie. Dzięki temu możemy stworzyć z tych danych poddrzewo i dodać je do istniejącego indeksu za pomocą zwykłego wskaźnika do pamięci tego poddrzewa.

Fakt 6.1 *Niech R_1, R_2 będą korzeniami dwóch poddrzew w uporządkowanym drzewie k-arnym. Załóżmy także, że $R_1 \prec R_2$. Niech M będzie maksymalnym kluczem w poddrzewie*

Pseudokod 6.5: lamFind(input: Key *keys*[ ])

```
1  Entry result[ ] := ∅
2  Entry toInsert[ ] := ∅
3  Key keysFromIndeks[ ] := ∅
4  Key keysFromPartitions[ ] := ∅
5
   // na podstawie dziennika określ, które klucze wczytujemy z partycji
6  foreach Key k ∈ keys do
7      if journalContains(k) = False then
8          keysFromPartitions := keysFromPartitions ∪ k
9      else
10         keysFromIndeks := keysFromIndeks ∪ k
11
12 result := result ∪ indexFindRange(keysFromIndeks)
13
   // znajdź partycje do wczytania na podstawie ich atrybutów
14 Partition toLoad[ ] := ∅
15 foreach Key k ∈ keysFromPartitions do
16     foreach Partition p ∈ PBTree do
17         if k ⩾ p.min AND k ⩽ p.max then
18             toLoad := toLoad ∪ p
19
   // wykonaj równolegle
20 foreach Partition p ∈ toLoad do
21     foreach Key key ∈ keysFromPartition do
22         if k ⩾ p.min AND k ⩽ p.max then
               // wyszukiwanie binarne lub tylko krok naprzód
23             Page page := pageFind(p, k) Entry e := entryFind(p, k)
24             toInsert := toInsert ∪ e
25             Block b := page.block
26             usage[b.id] := usage[b.id] + 1
27
28 result := result ∪ toInsert
29
   // wstaw nowe dane, aby powiększyć częściowy indeks
30 indexBulkloadInsert(toInsert)
31 Pair p := (min(keysFromPartition), max(keysFromPartition))
32 journalAdd(p)
33
34 if blocks.len - fractionalBPP(usage).len ⩾ MT then
35     lamReorganization()
36
37 return result
```

Pseudokod 6.6: lamReorganization()

```
   // wykonaj algorytm prostego wariantu pakowania
1  Block toLoad[ ] := fractionalBPP(usage) Entry entries[ ] := ∅
2
3  foreach Block b ∈ toLoad do
4      foreach Entry e ∈ b.entries do
5          if journalContains(e.key) = False then
               // skopiuj i zapisz do nowej partycji
6              entries := entries ∪ e
7      Usuń blok ze swojej partycji
8      usage[b.id] := 0
9
   // zapisz nową posortowaną partycję na dysku
10 sort(entries)
11 Partition p := partitionCreate(entries)
12 Zapisz p na dysku
```

o korzeniu R_1. Wtedy $M < R_2.key$.

Zatem, aby nie zniszczyć porządku w drzewie, musimy dodać korzeń naszego nowego poddrzewa do węzła posiadającego największy z mniejszych kluczy od korzenia poddrzewa. Niestety, dodając w ten sposób dane możemy zepsuć strukturę drzewa. Przykładowo drzewo może przestać być zbalansowane. Dlatego po każdej naszej operacji należy sprawdzić, czy zachowaliśmy wszystkie własności indeksu i naprawić je jeśli trzeba. Warto nadmienić, że zbalansowane struktury zawsze posiadają gotowe algorytmy naprawiające je, z których możemy skorzystać. Oczywiście struktura danych musi posiadać również pewne cechy, które umożliwią dodanie elementów w taki sposób. Reasumując, struktura musi posiadać następujące własności:

1. Struktura danych musi być drzewem
2. Drzewo musi pozwalać na dodanie danej bezpośrednio do każdego węzła (a nie tylko do liścia)
3. Drzewo musi posiadać algorytm naprawczy, potrafiący przeprowadzić reorganizację i naprawić wszelkie błędy wynikające z użycia dodawania zbiorczego (np. ponowne zbalansowanie drzewa)
4. Dane wewnętrzne (ang. metadane) zmieniają się tylko na podstawie dodanych rekordów lub zmiana ma wpływ tylko na węzły ze ścieżki od korzenia do poddrzewa.

Pierwsze ograniczenie wynika z konstrukcji algorytmu (podpinanie poddrzewa). Drugie jest wymagane, ponieważ bez niego nie możemy dodać całego poddrzewa, wstawiając wskaźnik do wybranego węzła. Potrzebujemy także trzeciego ograniczenia, aby móc naprawić strukturę. Zauważmy, że nie wiemy jak wygląda nasze drzewo i jakie posiada własności. Nasz sposób wstawiania gwarantuje tylko to, że nie zaburzymy porządku kluczy. Dlatego musimy mieć możliwość naprawy indeksu. Ostatnie ograniczenie zostało wprowadzone ze względu na efektywność naszego algorytmu. Ponieważ jeśli będziemy musieli zmieniać węzły, których nie odwiedziliśmy, to wykonamy wiele dodatkowych operacji. W

takkim przypadku nasz algorytm przestanie być efektywny.

Przedstawiony algorytm jest dostosowany do pamięci blokowych takich jak dyski SSD, gdyż nie wykonuje osobnych operacji dla każdej pojedynczej danej. Zamiast tego tworzy poddrzewo i zapisuje je na dysku w sposób sekwencyjny. Samo wstawianie poddrzewa zajmuje porównywaną ilość czasu do wstawienia jednego klucza. Oprócz tego czasami musimy ponieść koszt naprawy struktury drzewa, którego używamy. Większość algorytmów stabilizujących drzewa, przeprowadzają reorganizację tylko na pojedynczej ścieżce z węzła, który zaburzył własności struktury, do korzenia. Zatem koszt takiego algorytmu jest logarytmiczny. Jednak samo wykonanie tego algorytmu może powodować dwa problemy na pamięci flash:

1. Zmiana węzła, powoduje zmianę jego fizycznego położenia w pamięci
2. Brak pamięci na dodanie klucza bezpośrednio do węzła

Pierwszy problem wydaje się być bardzo poważny. Zmiana jakiejkolwiek wartości spowoduje wywołanie operacji wymazania przed ponownym zapisem. Aby przyspieszyć ten proces, często zmienia się położenie nowej strony. Jednak wtedy wszystkie wskaźniki będą posiadały niepoprawną wartość. Aby tego uniknąć wystarczy wprowadzić mapowanie logiczno-fizyczne na blokach, które posiada między innymi FTL [89]. Dzięki temu zmiany są dla nas przeźroczyste, ponieważ odwołujemy się do adresu wirtualnego, który się nie zmienia. Obecnie każdy dysk SSD posiada wbudowany algorytm FTL. Drugi problem jest nieco trudniejszy do rozwiązania. Najlepiej zarezerwować w każdym węźle kilka bajtów na dodatkowe wskaźniki(np. 2-3) wstawione w ten sposób. Jeśli jednak nadal brakuje pamięci na kolejny klucz, najlepiej wczytać tę cześć drzewa do RAM i wymusić reorganizację tego węzła.

Pamiętajmy, że powyższe ograniczenia są spowodowane optymalizacją samego dodawania dużych zbiorów danych do indeksu podczas przepisywania ich z partycji do indeksu. Z systemem LAM może współpracować dowolny indeks. Jednak gdy nie posiada wyżej wymienionych cech, czas dodawania do indeksu znacząco się wydłuży. Zamiast wykorzystania opisanego sposobu można również użyć struktur, które buforują dane i posiadają wsparcie dla operacji dodawania zbiorczego (ang. bulkload). Przykładem takiego drzewa jest struktura FD. Ze względu na dużą efektywność dodawania i usuwania rekordów oraz brak potrzeby modyfikacji wbudowanych operacji w celu przyspieszenia dodawania zbiorczego, struktura FD jest domyślnym indeksem systemu LAM.

6.7.5 Eksperymenty

		Pojemność		Prędkość losowego		Prędkość sekwencyjnego	
Model	Interfejs	Strony	Bloku	Odczytu	Zapisu	Odczytu	Zapisu
Samsung 840	SATA	8KB	512KB	390MB/s	182MB/s	585MB/s	535MB/s
Toshiba VX500	SATA	4KB	256KB	379MB/s	267MB/s	568MB/s	525MB/s
Intel DCP4511	NVMe	4KB	256KB	1,2GB/s	240MB/s	2GB/s	1.47GB/s

Tabela 6.12: Wybrane modele dysków SSD

W tej części rozdziału przedstawimy i przeanalizujemy eksperymenty wykonane za pomocą symulatora SIPS, który został dokładnie opisany w rozdziale 4. Jako indeks systemu LAM wybrano drzewo FD ([25]), ponieważ jest dobrze zoptymalizowane pod dyski SSD. Parametr MT ustawiono na 400 bloków. Dodatkowo aby zminimalizować narzut na reorganizację, bierzemy pod uwagę tylko te bloki, które posiadają mniej niż 25% danych jeszcze nie przepisanych do indeksu. Dzięki temu unikniemy sytuacji, w której algorytm połączy ze sobą wszystkie bloki w celu zmniejszenia ich liczby o 400. Tabela 6.12 zawiera szczegółowe parametry różnych modeli dysków SSD, które są zawsze używane do testów w tej pracy. Przypomnijmy, że Samsung 840 charakteryzuje się bardzo dużą pojemnością strony jak i bloku, ponieważ pojemność strony to aż 8KB, w przeciwieństwie do pozostałych modeli o standardowej pojemności 4KB. Starsze modele z interfejsem SATA posiadają podobne prędkości sekwencyjnych operacji. Różnią się głównie prędkością losowego zapisu, przy czym model Samsunga jest gorszy w tej kwestii od modelu Toshiba aż o 80MB/s. Najlepszym dyskiem w tym zestawieniu jest Intel DCP 4511. Ten model charakteryzuje się ogromną prędkością sekwencyjnych operacji (ponad 1GB/s) oraz bardzo szybkim odczytem losowym (1,2GB/s). Największą wadą tego modelu jest bardzo wolny zapis losowy na poziomie tylko 240MB/s. Eksperymenty zostały przeprowadzone na dwóch tabelach Sklep i Klient.

Tworzenie pełnego indeksu

	Samsung 840		**Toshiba VX500**		**Intel DCP4511**	
	AM	**LAM**	**AM**	**LAM**	**AM**	**LAM**
SERIA I (Sklep, 1%)	141s	71s	178s	78s	192s	24s
SERIA II (Sklep, 5%)	88s	68s	105s	76s	70s	23s
SERIA III (Klient, 1%)	642s	537s	720s	570s	431s	176s
SERIA IV (Klient, 5%)	593s	532s	652s	566s	320s	174s

Tabela 6.13: Całkowity czas wykonania zestawu kwerend

	Samsung 840		**Toshiba VX500**		**Intel DCP4511**	
	AM	**LAM**	**AM**	**LAM**	**AM**	**LAM**
SERIA I (Sklep, 1%)	17GB	10GB	19GB	12GB	19GB	12GB
SERIA II (Sklep, 5%)	15GB	10GB	17GB	12GB	17GB	12GB
SERIA III (Klient, 1%)	125GB	82GB	130GB	82GB	130GB	78GB
SERIA IV (Klient, 5%)	123GB	82GB	120GB	78GB	128GB	78GB

Tabela 6.14: Zużycie pamięci (liczba nadpisanych i usuniętych gigabajtów)

Tworzenie pełnego indeksu to skomplikowany i często długi proces. Porównanie sumarycznych czasów nie jest wystarczające, aby zrozumieć charakterystykę systemów oraz ich słabe i mocne strony. Z tego względu, oprócz sumarycznych statystyk takich jak czas i zużycie pamięci, porównujemy również czas każdej kwerendy. Ta seria eksperymentów ma na celu analizę systemów Adaptive Merging i Lazy Adaptive Merging podczas procesu tworzenia indeksu. Polega ona na wykonaniu samych kwerend wyszukujących o ustalonej selektywności, aż do momentu gdy indeks nie zostanie w pełni utworzony. Dzięki temu będziemy mogli przeanalizować w pełni różnice i podobieństwa obu systemów. Zatem startujemy z bazą danych zawierającą 10mln rekordów, które nie są w żaden sposób posortowane. Wykonujemy kolejne kwerendy z wyszukiwaniem losowym o podanej selektywności, w czasie których dodajemy odczytane rekordy do indeksu. Eksperyment kończy się, gdy indeks będzie w pełni utworzony.

W [139] pokazano bardzo dobry sposób na prowadzenie obserwacji takiego procesu. Sposób obserwacji został dokładnie opisany w rozdziale 4. Przypomnijmy jednak główne zasady. Autorzy wyznaczyli cztery fazy, które można wyróżnić w każdym algorytmie indeksowania.

1. **sadzenie (ang. planting)** - Pierwsza faza sadzenia polega na inicjalizacji całego procesu. Indeks zawiera na tyle mało elementów, że prawdopodobieństwo jego użycia jest bliskie 0. Dlatego musimy wykonać drogie skanowanie oraz dodać kilka operacji potrzebnych na powiększenie indeksu.

2. **pielęgnacja (ang. nursing)** - Druga faza zaczyna się, gdy czas jest mniejszy niż prosty skan, ale nadal większy niż w przypadku użycia pełnego indeksowania.

3. **wzrastanie (ang. growing)** - Trzecia faza zaczyna się, gdy natrafimy na zapytanie, które możemy obsłużyć w minimalnym czasie, czyli takim samym co pełny indeks.

4. **zbiory (ang. harvesting)** - Ostatnia faza zbiorów zaczyna się, gdy indeks jest w pełni stworzony, a zatem nie mamy już czasowych narzutów na jego tworzenie. Czas na obsługę zapytania jest taki sam jak przy pełnym indeksie.

Aby zbadać wszystkie zmienne, które mogą wpłynąć na charakterystykę wyników, przeprowadziliśmy trzy serie testów. Każda z nich wykonywała wyszukiwania o wzorcu losowym, który najlepiej oddaje naturę hurtowni danych. Taki serwer ma tak wielu użytkowników, że mimo iż każdy z nich tworzy zapytania według jakiegoś wzorca, to łącząc wszystkie kwerendy w ciąg, zapytania wyglądają jak przypadkowe (losowe).

1. SERIA I - Tworzymy pełny indeks na tabeli Sklep, używamy do tego wyszukiwań z zakresu kluczy o selektywności 1%.

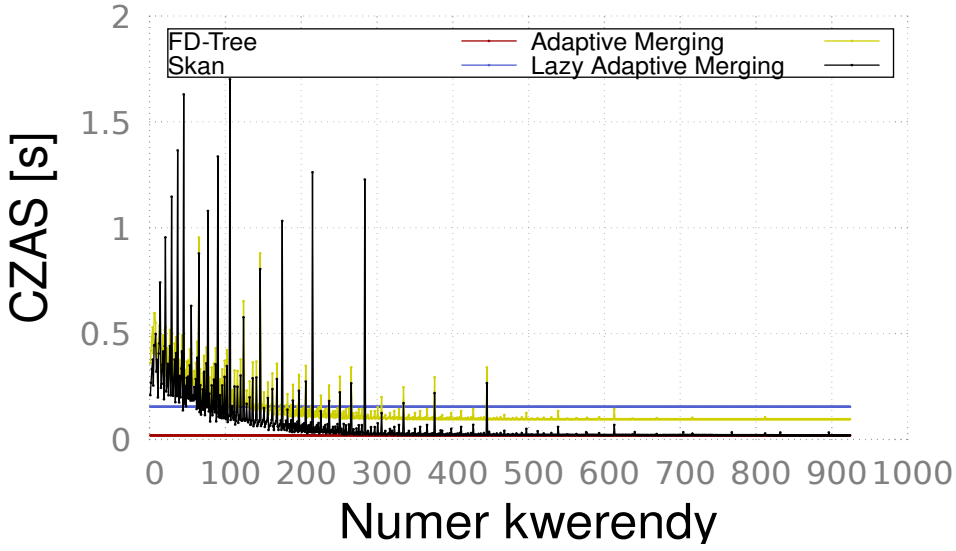
2. SERIA II - Tworzymy pełny indeks na tabeli Sklep, używamy do tego wyszukiwań z zakresu kluczy o selektywności 5%.

3. SERIA III - Tworzymy pełny indeks na tabeli Klient, używamy do tego wyszukiwań z zakresu kluczy o selektywności 1%.

4. SERIA IV - Tworzymy pełny indeks na tabeli Klient, używamy do tego wyszukiwań z zakresu kluczy o selektywności 5%.

Tabela 6.13 zawiera dokładny sumaryczny czas wszystkich serii z użyciem 3 wybranych modeli dysków SSD. Dodatkowo tabela 6.14 zawiera sumaryczne zużycie pamięci

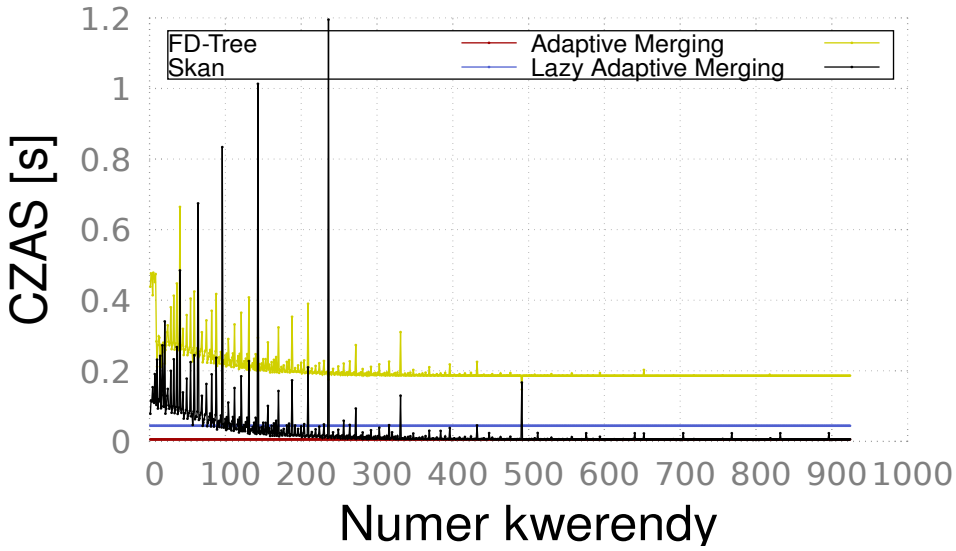
dysku SSD, czyli liczbę gigabajtów nadpisaną i usuniętą z dysku podczas procesu tworzenia pełnego indeksu. Jak możemy zauważyć, im większa selektywność, tym oryginalny system Adaptive Merging krócej tworzy pełen indeks. Co ciekawe, selektywność nie wpływa na sumaryczny czas systemu LAM. Dzieje się tak, ponieważ większa liczba danych kopiowana z partycji do indeksu powoduje większą jednoczesną reorganizację w systemie AM. Im mniej takich procesów tym mniej kasowań (ang. erase) wykonamy na dysku SSD, a więc będziemy mieli mniejszy czas. System LAM sam potrafi buforować dane i odraczać w czasie moment reorganizacji. To czy wykonamy ją po 2 czy po 200 kwerendach nie wpływa na sumaryczny czas. Hipotezę potwierdza również liczba zużytych i usuniętych bajtów z tabeli 6.14. Jak widać zużycie jest takie samo w przypadku obu selektywności, co oznacza tę samą liczbę wykonanych reorganizacji. Dlatego nie ma ona wpływu na system LAM. Z tego samego powodu największą różnicę pomiędzy systemami AM i LAM widzimy na dysku Intel. Model ten posiada bardzo powolne kasowanie bloków. Dlatego redukcja tej operacji w systemie LAM znacząco poprawiła efektywność tworzenia indeksu. Zauważmy również, iż rozmiar rekordu ma wpływ na szybkość działania obu systemów. Oczywistym faktem jest, iż tabela posiadająca większy rekord zajmuje na dysku więcej miejsca. Zatem operacje wykonywane na takiej bazie trwają dłużej. Warto jednak nadmienić, że testy na tabeli Klient pokazują mniejszą różnicę pomiędzy algorytmami AM i LAM. Dzieje się tak, ponieważ LAM używa bufora oraz parametru MT o ustalonych wielkościach, niezależnych od rozmiaru rekordu czy liczby rekordów w bazie. Zatem im większy rekord tym więcej bloków używamy do zapełnienia bazy oraz więcej bloków przepisujemy z partycji do indeksu. Wtedy reorganizację trzeba wykonać więcej razy, co spowalnia cały proces.



Rysunek 6.36: Czas tworzenia pełnego indeksu
Selektywność 1%
Dysk: Samsung 840
Tabela: Sklep (113B)

Przeanalizujmy teraz wyniki serii I. Rysunki 6.36 i 6.37 zawierają wyniki tej serii dla odpowiednio modeli Samsung i Intel. Już na początku widzimy, że AM na modelu Samsunga nie osiągnął nigdy fazy 3 (wzrastania). Czas wykonania kwerendy był zawsze większy niż czas wyszukiwania w pełnym indeksie. Dzieje się tak, ponieważ każda kwerenda wczytująca dane z partycji musi natychmiast przepisać je w inne miejsce, co powoduje

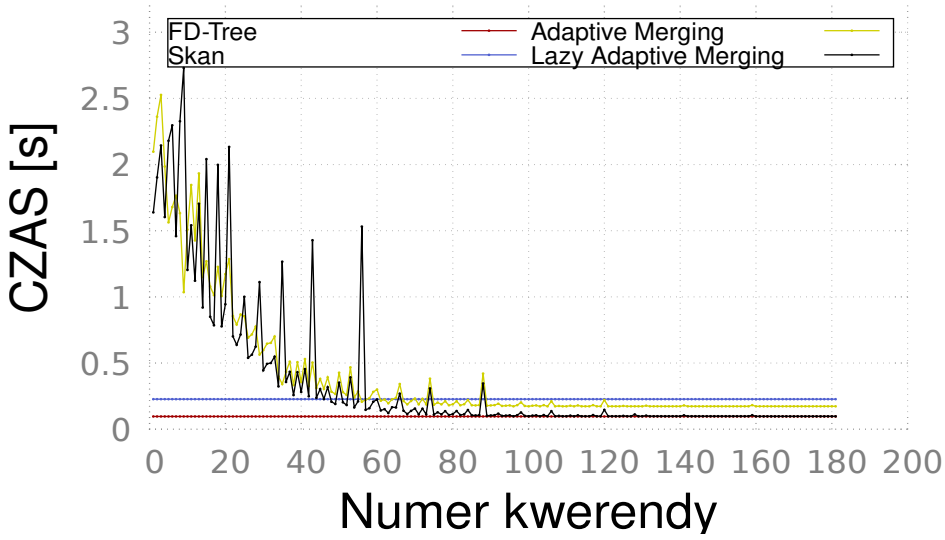
niepotrzebne i kosztowne operacje kasowania bloków. System LAM już w 51 kwerendzie uzyskuje fazę 2, a w fazę 3 w kwerendzie 182. Oczywiście istnieją pojedyncze kwerendy, które trwają znacznie dłużej niż prosty odczyt całej bazy. Takie sytuacje wynikają z dwóch powodów. Pierwszym z nich jest zestaw kluczy do wczytania. Zdarza się, że znaczną część rekordów musimy wczytać z partycji i przepisać do indeksu. Taki proces zajmuje sporo czasu. Ponieważ oba systemu używają tych samych wylosowanych kwerend, to sytuację tę możemy zaobserwować analizując wyniki uzyskanych eksperymentów, np. omawiane rysunki 6.36 i 6.37. W takim przypadku oba systemy AM i LAM wykonują zapytanie dłużej niż zwykle. Druga sytuacja to potrzeba wykonania reorganizacji. System LAM nie usuwa danych natychmiastowo, lecz czeka aż będzie mógł znacząco zredukować liczbę bloków. Gdy taka chwila nadejdzie, czas wykonania kwerendy jest o wiele większy niż normalnie.



Rysunek 6.37: Czas tworzenia pełnego indeksu
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Sklep (113B)

Przechodząc do wyników uzyskanych na dysku Intel DCP4511 widzimy, że AM na tym modelu nie osiągnął nigdy fazy 2 (pielęgnacji). Czas wykonania kwerendy był zawsze większy niż czas pełnego skanowania. Wynika to z charakterystyki tego dysku. Posiada on bardzo szybki odczyt sekwencyjny, który używany jest do skanowania tabeli oraz bardzo wolne kasowanie bloków, które jest wymagane do reorganizacji danych w strukturze AM. Całkiem inną sytuację prezentuje zaproponowany system LAM. Już w 78 kwerendzie uzyskuje on fazę 2, a w 158 kwerendzie fazę 3. Dalej istnieją pojedyncze kwerendy, które trwają znacznie dłużej niż prosty odczyt całej bazy. Powody pojawiania się takich kwerend są dokładnie takie same jak w przypadku testów przeprowadzanych na dysku Samsung. Jest to albo zestaw kluczy, który musimy wczytać w znaczącej mierze z partycji albo są to momenty reorganizacji.

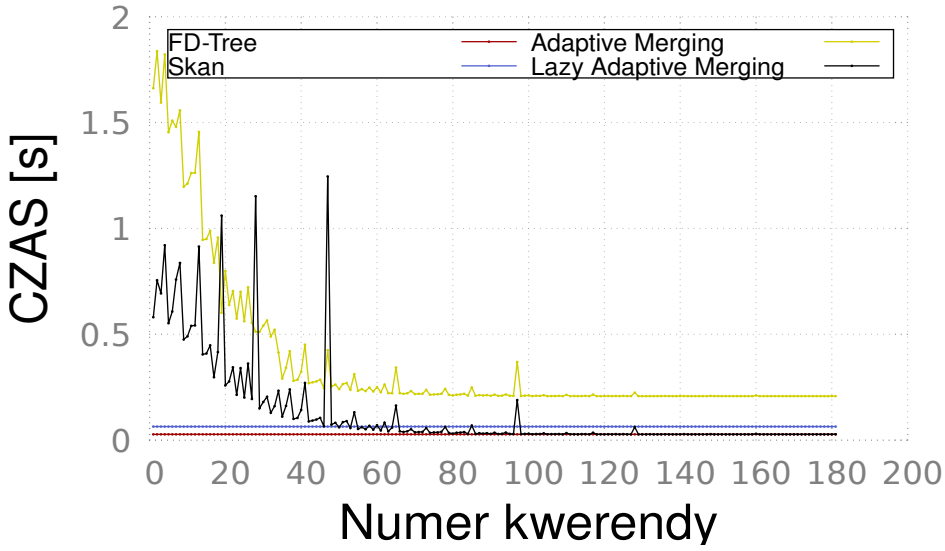
Z tej serii możemy wywnioskować, że wejście w fazę drugą zależy od charakterystyki dysku SSD. Im wolniejsze kasowanie bloku przy jednocześnie szybkim odczycie sekwencyjnym, tym dłuższa jest faza pierwsza. Podczas pierwszych kwerend zawsze musimy wczytać znaczącą część danych ze zbioru partycji, a to powoduje zwiększoną liczbę prze-



Rysunek 6.38: Czas tworzenia pełnego indeksu
Selektywność 5%
Dysk: Samsung 840
Tabela: Sklep (113B)

pisywanych rekordów oraz reorganizacji struktur. Dlatego szybkość kasowania i zapisu ma duży wpływ na czas trwania fazy pierwszej. Warto zauważyć, że na obu modelach system LAM osiągnął fazę trzecią w podobnym czasie. Dzieje się tak, ponieważ gdy natrafimy na kwerendę, która żąda klucze z indeksu, to za pomocą szybkiego wyszukiwania, dokładnie takiego samego jak w przypadku pełnego indeksu znajdujemy rekordy i kończymy wykonywanie zapytania.

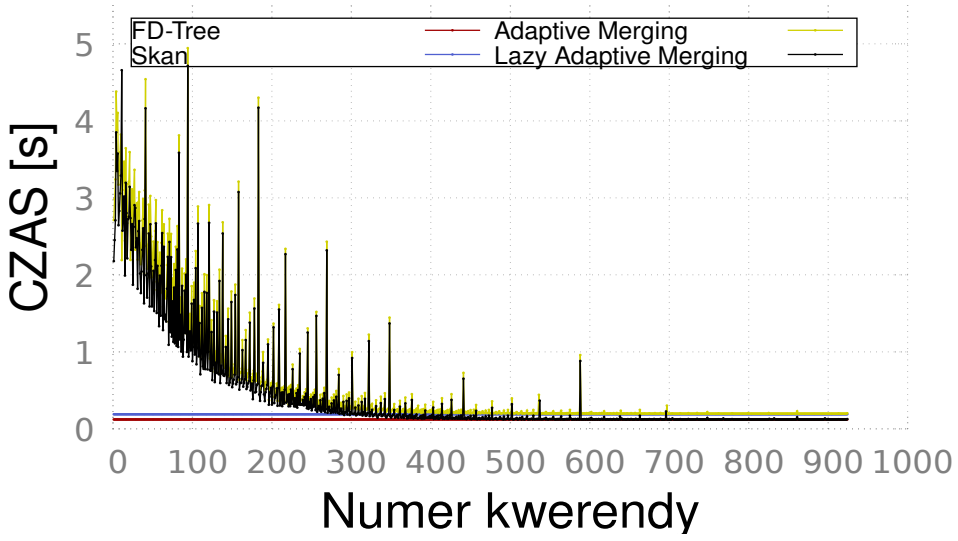
W serii II dalej pracujemy na tabeli Sklep o rozmiarze rekordu 113B. Tym razem selektywność jest ustawiona aż na 5%. Rysunek 6.38 przedstawia wyniki serii II wykonanej przy użyciu dysku Samsung 840. Z kolei rysunek 6.39 zawiera wyniki modelu Intel DCP4511. Widzimy, że przy większej selektywności liczba kwerend potrzebna na zbudowanie pełnego indeksu jest mniejsza. W serii I zbudowanie indeksu zajęło ponad 900 kwerend, w tej serii już tylko 180. Takie zachowanie jest zgodne z oczekiwaniami. Im więcej danych wczytujemy, tym więcej przepisujemy do indeksu, a co za tym idzie szybciej go utworzymy. Dodatkowo wyższa selektywność wpłynęła na długość poszczególnych faz względem długości procesu. Wcześniej przy użyciu modelu Samsung 840 (6.38) już po 51. kwerendzie, czyli po 5,5% wszystkich kwerend, system LAM uzyskał fazę 2 (pielęgnacji). Czas obsługi kolejnych kwerend był już mniejszy niż czas pełnego skanowania. Przy selektywności 5% użytej w tej serii, fazę 2 uzyskujemy dopiero w 41. kwerendzie (22,6% wszystkich kwerend), po której już po kilku kwerendach uzyskujemy fazę 3, ponieważ natrafiliśmy na zapytanie, które system LAM obsłużył wykorzystując częściowo zbudowany indeks. Sytuacja algorytmu AM jest podobna. Również moment osiągnięcia fazy 2 jest wydłużony względem całkowitego czasu potrzebnego na zbudowanie indeksu, z tą tylko różnicą, że w przeciwieństwie do systemu LAM, AM nie osiąga nigdy fazy 3. Taką samą sytuację obserwowaliśmy w serii I przy selektywności 1%. Podobnie jak w omawianym poprzednio przykładzie, podczas wykonywania testów na modelu Intel (rysunek 6.39) osiągnięcie fazy 2 trwało relatywnie dłużej. Dodatkowo patrząc na wyniki uzyskane przez system AM widzimy, że tak jak w przypadku serii I wykonanej na tym



Rysunek 6.39: Czas tworzenia pełnego indeksu
Selektywność 5%
Dysk: Intel DCP4511
Tabela: Sklep (113B)

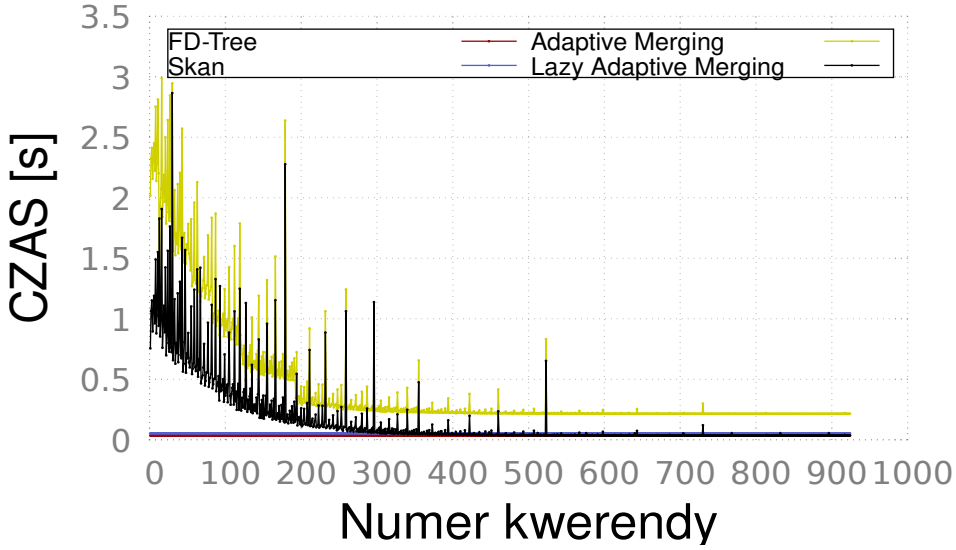
modelu pamięci, nie osiągnął on nigdy fazy 2. Każda kwerenda wykonywała się dłużej niż do pełnego skanowania. Ponownie wynika to z omawianej wcześniej charakterystyki tego dysku.

Podsumowując tę serię widzimy, że selektywność skraca liczbę kwerend potrzebną do zbudowania pełnego indeksu i jednocześnie wydłuża czas trwania fazy 1 na systemie LAM. Wynika to z większej liczby kopiowanych danych i częstszej potrzeby przeprowadzenia reorganizacji w początkowej fazie tego procesu.



Rysunek 6.40: Czas tworzenia pełnego indeksu
Selektywność 1%
Dysk: Samsung 840
Tabela: Klient (719)

Seria III była uruchamiana dla tabeli Klient o dużym rozmiarze rekordu równym aż 719B. Selektywność wyszukiwań ustawiona była na poziomie 1% wielkości całej bazy, która we wszystkich seria jest równa 10 milionom rekordów. Celem tej serii jest sprawdzenie wpływu wielkości rekordu na charakterystykę procesu tworzenia indeksu przy użyciu algorytmów AM i LAM. Rysunki 6.40 oraz 6.41 przedstawiają wyniki tej serii na modelach Samsung 840 i Intel DCP4511. Chociaż wielkość rekordu nie wpływa na liczbę kwerend jaką musimy wykonać aby zbudować pełny indeks, to parametr ten wpływa na czas trwania fazy 1 i 2.



Rysunek 6.41: Czas tworzenia pełnego indeksu
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)

Tak samo jak w serii II, czas potrzebny na uzyskanie fazy 2 znacząco się wydłużył w obu systemach AM i LAM. Wynika to z asymetrii dysków SSD. O wiele szybciej jest odczytać dane niż je zapisać. Zatem im większe rekordy tym większa będzie dysproporcja pomiędzy czasem odczytu i zapisu. Dodatkowo pamiętajmy, że system LAM tworzy partycje o ustalonych wielkościach, przy czym im większy rekord, tym więcej będzie partycji w tym zbiorze. Spowoduje to więcej reorganizacji wykonanych podczas trwania całego procesu. Przepisywanie danych najdłużej trwa właśnie w początkowej fazie, ponieważ wtedy najwięcej z nich musimy wczytać ze zbioru partycji i przepisać do indeksu, a w razie potrzeby wykonać reorganizacje na zbiorze partycji. To sprawia, że faza 1 trwa znacznie dłużej niż w serii I.

Podsumowując, rozmiar rekordu nie wpływa na liczbę kwerend potrzebną na stworzenie indeksu, ale wydłuża czas trwania fazy 1, w której czas potrzebny na obsługę kwerendy jest znacznie większy niż zwykłego skanowania nieposortowanych zbiorów danych. Seria IV zawierała podobne wyniki co seria II i III, dlatego zdecydowaliśmy się na pominięcie rysunków z tej serii w pracy. Ze względu na dużą selektywność liczba potrzebnych kwerend do zbudowania indeksu zmalała dokładnie tak samo jak w serii II. Jednocześnie przez zwiększoną selektywność jak i rozmiar rekordu faza 1 trwała aż 50% czasu całego procesu na systemie LAM przy użyciu modelu Samsung 840 jak i Intel. Algorytm AM

tak samo jak we wcześniejszych seriach nie osiągnął fazy 3 na dysku Samsung i fazy 2 na pamięci Intel. Analizując omówione eksperymenty możemy stwierdzić, że nowy system LAM radzi sobie o wiele lepiej od oryginalnego algorytmu AM na dyskach SSD. Nie tylko szybciej tworzy indeks (nawet 8 krotnie szybciej), usuwa mniej bloków z dysku SSD, ale także wykonuje szybciej pojedyncze kwerendy. We wszystkich seriach faza 1, 2 i 3 trwała znacznie krócej niż w systemie AM. Oznacza to, że użytkownik szybciej zobaczy efekty naszego procesu i znaczne przyspieszenie wykonywania kolejnych zapytań.

Rozszerzony zestaw kwerend

W poprzedniej sekcji zbadaliśmy sam proces tworzenia częściowego indeksu. Zazwyczaj do bazy trafiają różne kwerendy, a nie tylko zapytania wyszukujące rekordy po kluczu. Dlatego w tej części rozdziału przeanalizujemy wyniki otrzymane podczas wykonywania kwerend z zestawu rozszerzonego. Charakterystyka wszystkich czterech zestawów opisana w rozdziale 4 jest następująca:

1. ZR_A - 100 serii, każda seria zawiera operacje dodawania 5 rekordów, 10 wyszukiwań o podanej selektywności oraz usuwanie 5 rekordów,

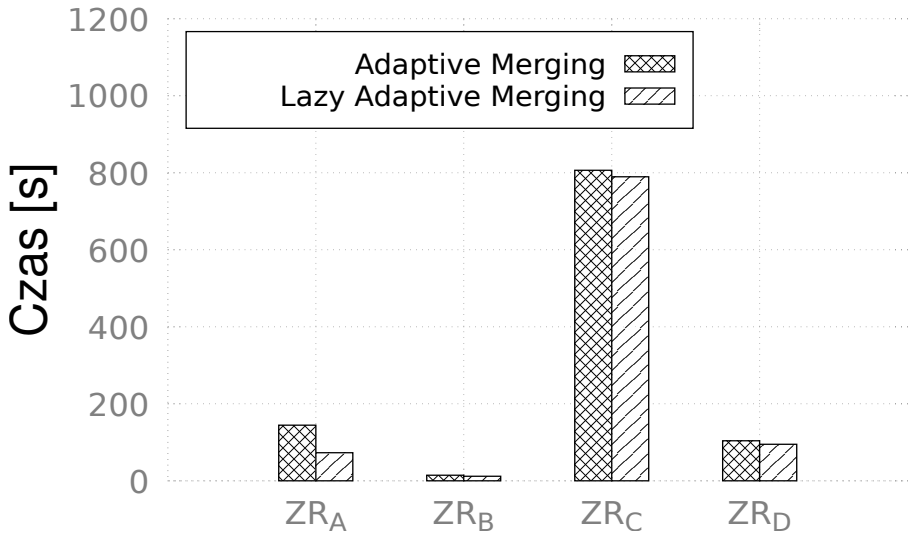
2. ZR_B - 5 serii, każda seria zawiera operacje dodawania 100000 rekordów, 5 wyszukiwań o podanej selektywności oraz usuwanie 100000 rekordów,

3. ZR_C - 10 serii, każda seria zawiera operacje dodawania 10000000 rekordów, 20 wyszukiwań o podanej selektywności oraz usuwanie 1000000 rekordów,

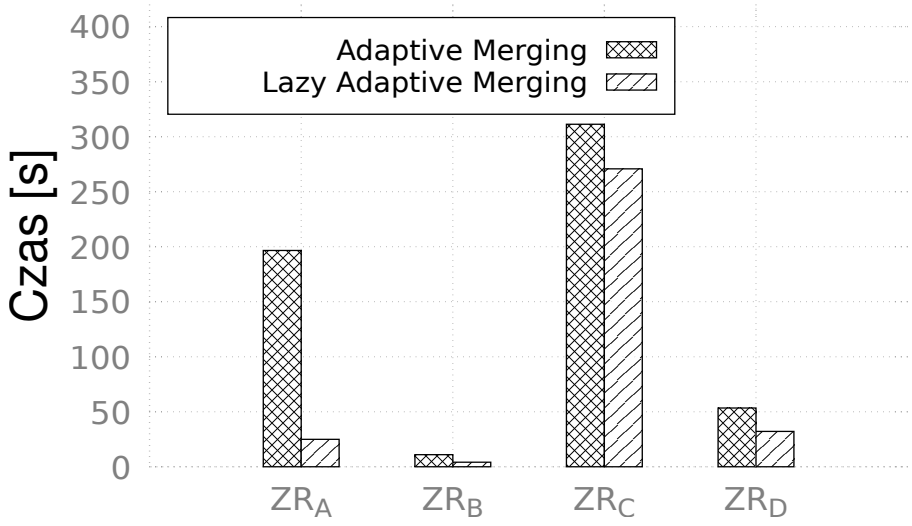
4. ZR_D - 10 serii, każda seria zawiera operacje dodawania 1000000 rekordów, 10 wyszukiwań o podanej selektywności oraz usuwanie 10000 rekordów.

Do testów wybraliśmy ponownie tabele Sklep i Klient. Wykonaliśmy każdy zestaw 2 razy, raz z selektywnością na poziomie 1%, drugi raz na poziomie 5%. Kolejny raz dyski Toshiba i Samsung uzyskały podobne wyniki, o tej samej charakterystyce.

W pierwszej serii (Sklep, 1% selektywności) na obu dyskach system LAM zawsze osiągał czasy lepsze niż klasyczny algorytm AM. Widzimy to na wykresach 6.42 oraz 6.43. Różnice w zestawach ZR_B, ZR_C oraz ZR_D są znikome. Wynika to z kilku przyczyn. Po pierwsze pamiętajmy, że wstawianie danych w systemie LAM odbywa się jako bezpośrednie dodanie nowych danych do indeksu.



Rysunek 6.42: Czas zestawów kwerend
Selektywność 1%
Dysk: Samsung 840
Tabela: Sklep (113B)

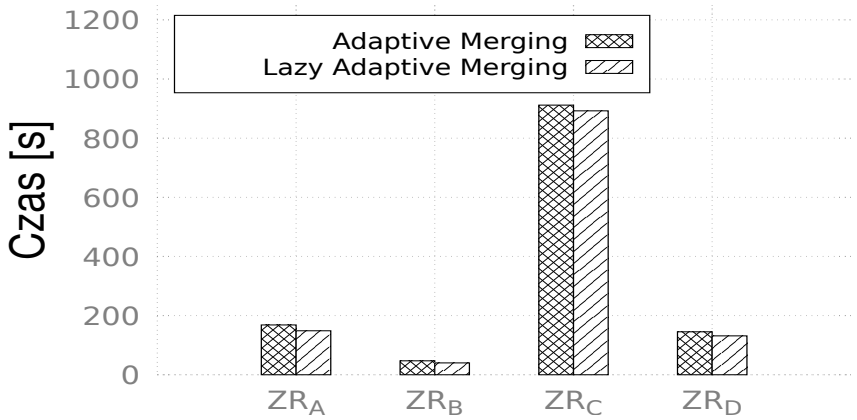


Rysunek 6.43: Czas zestawów kwerend
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Sklep (113B)

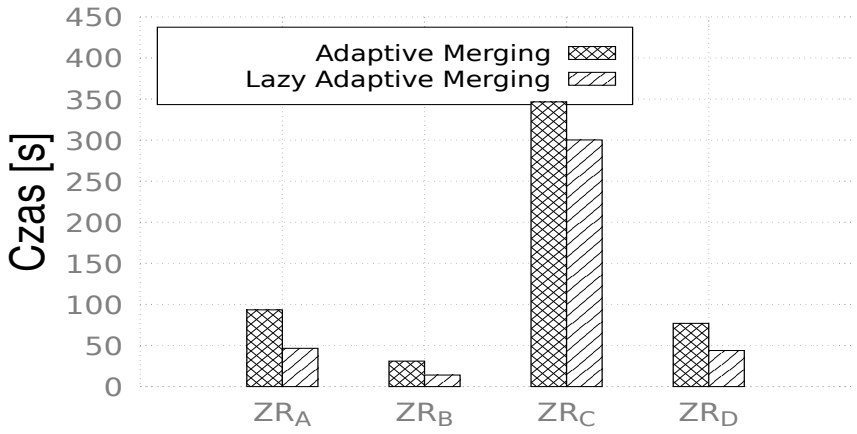
W systemie AM jako buforowanie danych i zapisanie ich bezpośrednio do posortowanej części indeksu jako nowa partycję. Oznacza to, że dodawanie elementów nie różni się zbyt mocno oraz nie wykonuje zmian widocznych w procesie tworzenia indeksu częściowego. Drugą przyczyną jest zbyt mała liczba wyszukiwań, które przeprowadzamy w tych zestawach. Pamiętajmy, że proces zajmuje około 800 kwerend wyszukujących. Tutaj mamy ich odpowiednio (25, 200, 100). Zatem jesteśmy dopiero w początkowych fazach

tworzenia się indeksu i nie możemy liczyć na dużą przewagę LAM nad AM wynikającą jak poprzednio z samego tworzenia indeksu. Po trzecie, oba algorytmy uzyskują bardzo podobne czasy w pierwszej fazie. Dopiero później algorytm LAM mocno wyprzeda klasyczne podejście. Warto jednak zauważyć, że zestaw ZR_A zawierający aż 1000 wyszukiwań, a zatem tworzący pełen indeks pokazuje ogromną przewagę LAM nad AM. Najlepiej widoczne jest to w czasie testów wykorzystując dysk Intela. Algorytm LAM osiągnął wyniki 6 razy lepsze od oryginalnego algorytmu.

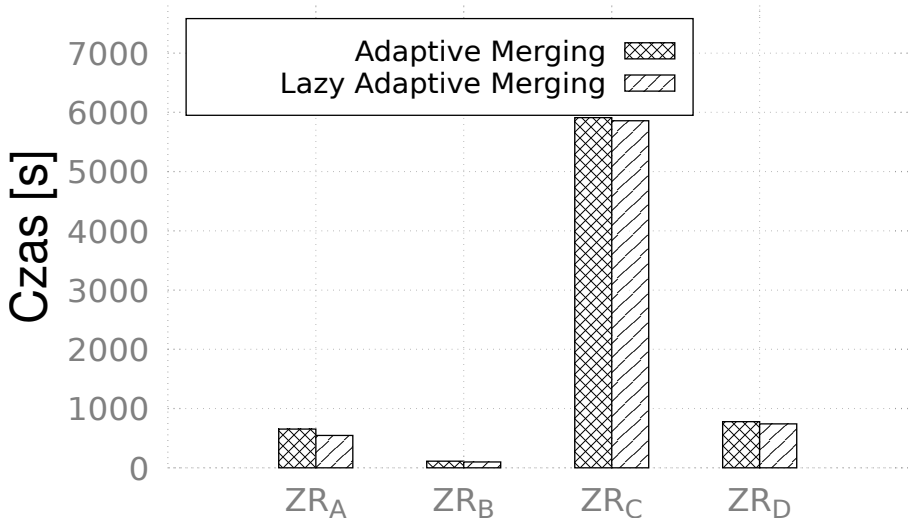
Sytuacja w pozostałych seria się nie zmieniła. Zmiana selektywności z 1% na 5% praktycznie nie wpłynęła na różnice w zestawach nastawionych na modyfikacje tabeli. Dodatkowo wzrost selektywności zmniejszył czas potrzebny algorytmowi AM na wykonanie wszystkich kwerend ZR_A. Jest to oczekiwany wynik, ponieważ jeśli spojrzymy na tabelę 6.13, to widzimy dokładnie, że czas tworzenia częściowego indeksu nie zmienia się dla algorytmu LAM, a dla systemu AM drastycznie się zmniejsza.



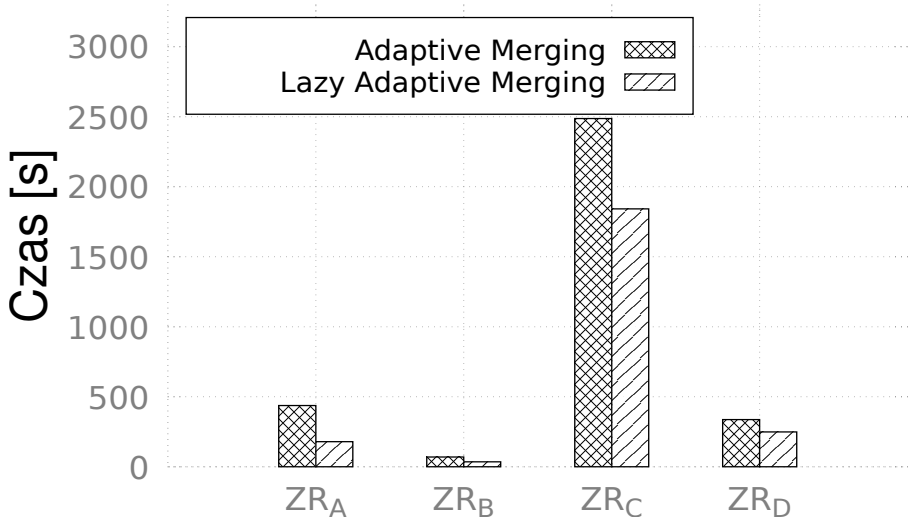
Rysunek 6.44: Czas zestawów kwerend
Selektywność 5%
Dysk: Samsung 840
Tabela: Sklep (113B)



Rysunek 6.45: Czas zestawów kwerend
Selektywność 5%
Dysk: Intel DCP4511
Tabela: Sklep (113B)



Rysunek 6.46: Czas zestawów kwerend
Selektywność 1%
Dysk: Samsung 840
Tabela: Klient (719B)



Rysunek 6.47: Czas zestawów kwerend
Selektywność 1%
Dysk: Intel DCP4511
Tabela: Klient (719B)

Podsumowując, obie serie eksperymentów wykazały ogromny przyrost wydajności obsługi kwerend za pomocą nowego systemu Lazy Adaptive Merging (LAM) względem pierwotnego algorytmu Adaptive Merging. Dzięki wprowadzonej mechanice leniwego usuwania, LAM nie tylko osiąga lepsze czasy obsługi zestawu kwerend, ale także nadpisuje i usuwa o wiele mniej pamięci, co sprawia, że dysk SSD pozostanie dłużej sprawny. Kolejną ważną cechą nowego systemu jest dowolna wręcz amortyzacja czasu potrzebnego

na budowanie indeksu. Zauważmy, że w systemach częściowego indeksowania najwięcej czasu zajmuje samo przepisanie danych do indeksu. Gdy takie systemy działają na dyskach SSD, dochodzi również czas obsługi kasowania bloków, aby przygotować pamięć na nowe dane. Dzięki parametrowi MT oraz dodatkowym pobocznym parametrom takim jak jak minimalne zużycie bloku czy maksymalna liczba bloków reorganizowana jednocześnie, możemy dowolnie sterować mechanizmem łączenia bloków. Dzięki temu możemy rozłożyć w czasie koszt reorganizacji tak, aby użytkownik końcowy nie widział ogromnego spadku na wydajności podczas niektórych zapytań.

Algorytmy indeksowania na pamięci PCM

W tym rozdziale omówimy sposoby indeksowania danych, obecnie używane struktury i algorytmy wykorzystywane w silnikach baz danych na pamięci PCM. Przedstawimy także nowy system do tworzenia częściowego indeksu na tej pamięci [5], który został specjalne dostosowany do charakterystyki pamięci PCM. Ogólny koncept każdego z rodzajów indeksowania przedstawiony został w rozdziale **2**. W tym rozdziale skupimy się na dostosowaniu algorytmów indeksujących do sposobu działania nowej pamięci zmiennofazowej PCM.

Charakterystyka nowej pamięci PCM została przedstawiona w rozdziale **3**. Przypomnijmy sobie jednak główne cechy tej pamięci. Pamięć zmiennofazowa PCM (ang. Phase Change Memory) [123] jest pamięcią nieulotną opartą na nośniku krystalicznym. Obecnie rozwojem tego rodzaju pamięci zajmuje się firma Intel [126]. Podobnie jak pamięć flash, PCM nie traci danych po odłączeniu zasilania. Pamięć ta jest dużo bardziej wydajna. Wartości dotyczące szybkości zapisu i odczytu bitów informacji szacuje się średnio na poziomie kilkadziesiąt razy wyższym niż w przypadku NAND. Podczas zapisu danych nie istnieje konieczność wymazywania całego bloku komórek. Dlatego pamięć ta może zostać wykorzystana nie tylko jako pamięć masowa, ale także jako pamięć operacyjna komputera [127], [128], [129]. Tak jak jest to w przypadku pamięci RAM, pamięć PCM jest również bajtowo adresowalna. Jednak ze względu na jej budowę oraz lepszą współpracę z pamięcią podręczna procesora, została podzielona na strony o wielkości 64B. Zazwyczaj kość zbudowana jest z 4 lub 8 banków pamięci. Obecnie wykorzystuje się głownie 8 banków, ponieważ w czasie jednego odczytu możemy wczytać całą stronę pamięci PCM (64B) odczytując z każdego banku pojedyncze słowo maszynowe architektury 64 bitowej, która jest obecnie najbardziej popularna w osobistych komputerach jak i serwerach. Niestety podobnie jak w przypadku pamięci flash, mimo bardzo szybkiego odczytu, pamięć ta charakteryzuje się wolniejszym (nawet 10 razy) zapisem. Kolejnym problemem jest limit zapisów pojedynczej komórki wynoszący od 10^7 do 10^8. Mimo iż 10mln może się wydawać limitem nie do osiągnięcia, to komputer używający pamięci PCM jako pamięci operacyjnej, mógłby zniszczyć pojedyncze komórki pamięci w zaledwie kilka minut. Jako rozwiązanie tego problemy zaproponowano kilka algorytmów mapujących linie pamięci PCM (64B) oraz logiczne strony (4KB) w podobny sposób jak algorytm FTL mapujący strony dysków SSD [191], [192], [193]. Aby zapobiegając kompletnemu zniszczeniu komórki PCM, system operacyjny pracuje na tych samych adresach logicznych, chociaż adres fizyczny zmienia się w czasie. W ten sposób taki system może wytrzymać kilkadziesiąt lat. Rozwiązanie pełnego mapowania pamięci podzielonej na strony czy inne mniejsze fragmenty nie zostało porzucone po stworzeniu podstawowych systemów. Problem okazał się znacznie większy. Zwykle pamięci są wykorzystywane do jednego celu. Pamięć RAM jest szybką pamięcią operacyjną komputera lub buforem danych w wolniejszych dyskach. Pamięć flash jest wykorzystywana do budowy pamięci masowej w tym dysków

SSD. Pamięć PCM wykorzystywana jest w wielu miejscach w odmienny sposób. Niektóre architektury stosują PCM jako główną pamięć komputera [130], inne jako pamięć masową [131], a jeszcze inne jako bufor danych pomiędzy dyskami SSD a pamięcią RAM [132]. Z tego względu zaproponowano kolejne algorytmy pozwalające wydłużyć życie komórek pamięci. Niektóre skupiły się na dynamicznym wykrywaniu użycia pamięci (np. dużo zapisów do pewnego obszaru pamięci) [194], [195], [196]. Inne tworzyły mapowanie na różnych poziomach systemu operacyjnego, tak aby zmaksymalizować pozytywny efekt mapowania pamięci na żywotność komórek, w zależności od używanej architektury. W [197] zaproponowano system mapujący działający na poziomie sterowników, w [198] na poziomie systemu operacyjnego (głównie systemy plików), a w [199] stworzono moduł, który można użyć w dowolnej aplikacji (ang. user space). Porównanie najważniejszych metod mapowania pamięci można znaleźć w [200]. To właśnie ze względu na asymetrię pomiędzy czasem odczytu a zapisu oraz odmienną charakterystykę pamięci PCM od klasycznych dysków HDD jak i nowych dysków SSD, klasyczne struktury danych muszą być dostosowane do tego rodzaju pamięci.

7.1 Indeksowanie wierszowe

Indeksy wierszowe to najbardziej popularne struktury danych w bazach danych. Zaletą tego typu indeksów jest prostota implementacji oraz szybkość utrzymania porządku na takim indeksie. Cały rekord zapisany jest w ciągłym obszarze pamięci. Dodawanie nowego rekordu, usuwanie starego jak i modyfikacje wykonują pojedyncze operacje na pamięci. Z drugiej strony taki rodzaj indeksowania posiada kilka wad. Zazwyczaj tabela składa się z kilku lub nawet kilkunastu atrybutów. Większość kwerend nie będzie potrzebowała ich wszystkich. Mimo to, ze względu na ułożenie w pamięci, musimy wczytać cały rekord niezależnie od zapytania, co niepotrzebnie wydłuża czas trwania kwerendy.

Najpopularniejszą strukturą danych wykorzystywaną do indeksowania wierszowego jest drzewo B+ [19]. Struktura ta jest bardzo często stosowana w bazach zapisanych na dyskach HDD i w pamięci operacyjnej komputera. To właśnie ze względu na jej popularność podjęto wiele prób dostosowania drzewa B+ do charakterystyki modelu PCM. W [134] porównano 4 modele drzew B+:

- standardowe drzewo B+ (ang. sorted B+), które ma posortowane liście oraz węzły wewnętrzne
- całkowicie nieposortowane drzewo B+ (ang. unsorted B+). Ten rodzaj drzewa utrzymuje główną ideę struktury. Drzewo jako zbiór węzłów tworzy minimalnie częściowy porządek, jednak same węzły są zbiorami nieposortowanymi.
- drzewo z nieposortowanymi liśćmi (ang. unsorted leaf). Ze względu na fakt, iż większość zapisów odbywa się w liściach, które zawierają dane, to właśnie liście są nieposortowane, aby zminimalizować liczbę modyfikacji wewnątrz tego zbioru. Z kolei węzły wewnętrzne, służą głównie do wyszukiwania odpowiedniego liścia. Operacją dominującą na tym typie węzłów jest odczyt. Z tego względu w tym modelu utrzymuje się porządek wewnątrz tych węzłów.
- drzewo z nieposortowanymi liśćmi, posiadające bitmapę (ang. unsorted leaf with bitmap). Jest to ten sam model co zwykłe nieposortowane liście, z tym wyjątkiem, że dodatkowo każdy liść zawiera bitmapę, określającą aktualny stan każdej wartości

(1 - aktualna, 0 - nieaktualna / usunięta).

Z eksperymentów przedstawionych w [134] wynika, że na pamięciach PCM najgorzej radzi sobie standardowe drzewo B+. Mimo że w pełni posortowane drzewo wyszukuje klucze najszybciej, to jednak każda pojedyncza modyfikacja jak dodawanie lub usuwanie rekordów jest bardzo kosztowna. Aby utrzymać posortowane węzły, trzeba znaleźć odpowiednie miejsce na nowy rekord i przeprowadzić reorganizację węzła tak, aby zbiór po dodaniu nowej wartości był dalej posortowany. Wartość oczekiwana takiej operacji to przepisanie aż połowy węzła, co jest bardzo kosztowne patrząc na charakterystykę modelu pamięci PCM. Porównując ze sobą trzy pozostałe modele, drzewo w pełni nieposortowane uzyskało najgorszy wynik ze wszystkich nieposortowanych struktur, mimo iż czas wykonania zestawu kwerend był znacząco mniejszy niż czas uzyskany przez standardowe drzewo. Okazuje się, że w pełni nieposortowane drzewo B+ bardzo wolno wyszukuje rekordy, ponieważ na każdym poziomie zamiast szybkiego wyszukiwania binarnego, musi wczytać cały węzeł. Najlepszym modelem pod względem szybkości wykonywania się kwerend, zużycia pamięci (ang. wearout) oraz poboru prądu okazał się ostatni model - drzewo z nieposortowanymi liśćmi posiadające bitmapę. Podział na nieposortowane liście, w których wykonuje się najwięcej modyfikacji i posortowane węzły wewnętrzne, które służą głównie do wyszukiwania rekordów okazał się najlepszym sposobem spośród wszystkich czterech zaproponowanych. Przede wszystkim dzięki wprowadzeniu bitmapy, usuwanie rekordu przeprowadzane jest jako nadpisanie pojedynczego bitu odpowiadającego usuwanej danej, co znacząco poprawiło szybkość wykonywania tej operacji względem modelu 3 (drzewo z nieposortowanymi liśćmi). Dzieje się tak, gdyż w modelu 3 aby usunąć daną, musimy nadpisać rekord ostatnim rekordem w węźle. Efektem tej optymalizacji było uzyskanie 7 krotnie mniejszego zużycia pamięci, 1,7 krotnie szybszego czasu wykonania zestawu kwerend przy wykorzystaniu dwukrotnie mniej energii. Model ten nazwano drzewem UB+ (unsorted B+ Tree).

Mimo tak dużego wzrostu wydajności, zaproponowano wiele innych modyfikacji drzewa B+. W [27] stworzono strukturę OB+-Tree oraz zmieniono algorytm reorganizacji drzewa w taki sposób, żeby zamiast natychmiastowo modyfikować węzeł rodzica, dodawać kolejny węzeł do listy (ang. overflow node). Autorzy głównie pracowali z modelem drzewa z nieposortowanymi liśćmi posiadającym bitmapę (UB+-Tree). Zauważyli, że nieposortowany liść w chwili braku miejsca na kolejny element jest sortowany i dzielony na 2 podzbiory dokładnie tak, jak to ma miejsce w przypadku oryginalnego drzewa B+. Proces sortowania w zależności od wielkości węzła może być czasochłonny, a sam proces dodawania kolejnego wskaźnika do rodzica, który jest zbiorem posortowanym wykonuje wiele operacji nadpisywania danych, które również są powolne na pamięci PCM. Aby zapobiec obu tym procesom, tworzony jest dodatkowy węzeł, który widoczny jest tylko z poziomu liścia jako lista dodatkowych węzłów (ponieważ każdy liść może mieć takich dodatkowych węzłów więcej niż 1). Podczas przepisywania danych nie sortuje się rekordów, lecz tak jak w przypadku sortowania szybkiego (ang. quicksort) wybiera się element rozdzielający (ang. pivot) i przepisuje się tylko rekordy o kluczu większym od wybranego elementu. Oba węzły dalej nie są posortowane, jednak ze względu na element rozdzielający podczas wyszukiwania wiemy, który węzeł musimy wczytać aby znaleźć pożądany rekord. Dodatkowo, ponieważ nowy węzeł widoczny jest tylko z poziomu liścia, do którego jest przypisany, nie musimy przeprowadzać kosztownej operacji nadpisywania danych w węźle rodzica. Mimo że taki zabieg wydłuża czas wyszukiwania rekordu, to jednak całkowity czas wykonywania zestawu kwerend spadł o 6% względem pierwotnego modelu.

W [29] zaproponowano kolejną modyfikację, tym razem modelu struktury OB+-Tree. Wprowadzenie dodatkowych węzłów, wydłuża czas wyszukiwania. W momencie gdy użytkownik wykonuje większą liczbę wyszukiwań niż modyfikacji, czas uzyskany przez drzewo OB+ jest większy niż czas uzyskany przez drzewo UB+. Autorzy widząc, że wprowadzenie dodatkowych węzłów zależy od wzorca kwerend zaproponowali swoją modyfikację nazwaną drzewem CB+. Struktura wykrywa sposób użycia poszczególnych fragmentów drzewa i dynamicznie włącza i wyłącza funkcjonalność związaną z dodatkowymi węzłami. Oczywiście zmiana nie następuje natychmiastowo, ponieważ algorytm potrzebuje czasu aby stwierdzić zmianę trendu. Jednak pomimo niewielkiego opóźnienia struktura osiągnęła czas o 12% lepszy od drzewa OB+.

W [201] oraz w [202] skupiono się na dostosowaniu rozmiaru węzła drzewa B+ w zależności od wykorzystania pamięci PCM. W przypadku pamięci RAM zazwyczaj rozmiar węzła jest równy rozmiarowi strony logicznej tej pamięci (najczęściej 4KB). W przypadku dysków HDD i SDD rozmiar również dostosowywano do pojemności fizycznych stron dysków (2KB-8KB). Ponieważ pamięć PCM może być wykorzystana jako pamięć operacyjna, bufor lub jako pamięć masowa, a sama pamięć podzielona jest na bardzo małe linie pamięci (64B), to ustalenie odpowiedniego rozmiaru węzła było sporym problemem. W [201] zbadano wpływ wielkości węzła struktury B+ na szybkość wykonywania się kwerend w sytuacji, której pamięć PCM była pamięcią główną komputera. Najlepsze wyniki uzyskano dla węzłów o rozmiarze 4-8 krotnie wyższym niż linia pamięci podręcznej (ang. cacheline), zatem przy takim wykorzystaniu pamięci najlepiej ustalić rozmiar węzła na 256-512B. Z kolei w [202] przeprowadzono eksperymenty z użyciem pamięci PCM jako pamięci masowej (dyskowej). Najlepsze wyniki osiągnięto wtedy, gdy węzeł był wielkości jednej lub dwóch stron pamięci RAM, czyli 4KB-8KB. Wynikało to z braku doczytujących się fragmentów pamięci. Jeden wstępny odczyt i zapis, wystarczał aby wykonywać pełne operacje na węźle (ang. prefetch).

Kolejną modyfikacją drzewa B+ jest wykorzystanie znajomości architektury komputera i pamięci PCM do efektywniejszej pracy na strukturze. W [28] analizowano model wykorzystania PCM jako pamięci dyskowej. Zaproponowano strukturę XB-Tree (ang. eXtendend B+-Tree), która w zależności od wykorzystania węzła umieszczała go w pamięci nieulotnej PCM lub w pamięci szybkiej RAM. Pomysł ten jest bardzo zbliżony do drzewa wędrującego ([203]), które wykorzystywane jest w wielu systemach plików zapisanych na dysku SSD, m.in. JFFS3 ([204]) i UBIFS ([205]). W obu strukturach (XB-Tree oraz Wandering-Tree) użycie węzła jest monitorowane. Gdy dominują odczyty, na stałe buforowane jest w pamięci RAM, aby znacząco przyspieszyć odczyt tego węzła. Gdy główną operacją jest modyfikowanie rekordów (dodawanie i usuwanie), to węzeł jest z powrotem zapisywany na pamięci PCM (ang. flush). Struktura XB-Tree osiągnęła aż o 40-60% lepsze wyniki w zestawach kwerend o dominujących odczytach od drzew OB+ i CB+. Podobną optymalizację wprowadzono w strukturze FP-Tree (ang. Fingering Persistent Tree) [206]. W tym indeksie również większość węzłów buforowanych jest na stałe w pamięci RAM. Dodatkowo każdy z węzłów ma osobny bufor na modyfikacje przeprowadzane na tym węźle. Dzięki temu większość operacji jest agregowana i przeprowadzana jednocześnie. W [207] zaproponowano całkiem nowy indeks, w pełni dostosowany do pamięci PCM oraz systemów wielowątkowych. Indeks DP-Tree składa się z 3 części. Pierwszą z nich jest bufor w pamięci RAM, który jest niewielkim, klasycznym drzewem B+. Służy on do agregowania wszystkich operacji. Do niego dodaje się nowe elementy, oraz zaznacza elementy do usunięcia. Drugą częścią systemu jest sam indeks, zapisany w pamięci PCM.

Jest to struktura podzielona na poziomy, podobnie jak drzewa LSM ([172]) czy FD ([25]). Gdy bufor osiąga maksymalną liczbę wpisów, łączony jest z głównym indeksem według algorytmu przygotowanego przez autorów. Dodatkowo aby uniknąć utraty danych podczas awarii systemu (pamiętajmy, że pamięć RAM jest ulotna, traci dane gdy system jest wyłączony), wprowadzono ostatnią część - dziennik operacji, zapisany na nieulotnej pamięci PCM. Bardzo ważną cechą tego systemu jest wsparcie dla operacji wielowątkowych. Algorytm zapisywania buforowanego drzewa B+ na indeks jest stworzony tak, aby mógł działać z wieloma buforami. Oznacza to, że w zależności od liczby użytkowników (klientów bazy), możemy wprowadzić wiele buforów. Każdy z nich będzie wykonywał operację pojedynczego użytkownika, który realizowany jest jako wątek w systemie. Dodatkowo umiejętne zapisywanie tylko niezbędnych metadanych potrzebnych do odzyskania informacji jak i pełne buforowanie indeksu sprawiło, że indeks ten jest obecnie domyślną strukturą wykorzystywaną przez silniki baz danych pracujących na pamięci PCM.

7.2 Indeksowanie kolumnowe

Indeksowanie kolumnowe w ostatnich latach tak mocno zyskało na popularności, że większość nowoczesnych silników baz danych wspiera ten sposób zapisu tabeli. Wynika to z faktu, że większość zapytań do baz danych nie potrzebuje danych ze wszystkich kolumn. Zatem w wierszowej implementacji marnujemy odczyty z dysku na wczytanie całego wiersza i wyłuskanie tylko potrzebnych atrybutów. W podejściu kolumnowym na jednej stronie w pamięci przetrzymywane są wartości pojedynczej kolumny dla kilku rekordów, dzięki czemu podczas wyszukiwania wczytujemy tylko potrzebne informacje. Niestety kolumnowe podejście jest wolniejsze podczas zapisywania nowych rekordów, ponieważ musimy zapisać podzielony rekord na kolumny w kilku miejscach na dysku [10]. Najpopularniejszą strukturą danych używaną obecnie do indeksowania rekordów zapisanych kolumnowo jest PDT [50] (ang. Positional Delta Tree). Indeks ten do niedawna wykorzystywany był tylko w bazach danych zapisanych w pamięci operacyjnej komputera (ang. in-memory database) oraz na dyskach talerzowych HDD. Obecnie struktura PDT jest domyślnym indeksem dla zapisu kolumnowego w silnikach MonetDB [45] oraz VectorWise [47]. Mimo, iż pamięć zmiennofazowa PCM posiada odmienną charakterystykę od pamięci RAM czy dysków HDD, to struktura PDT również stosowana jest w bazach zapisanych w pamięci PCM. Wynika to głównie z obecnego braku innych struktur lepiej dostosowanych pod model PCM. Możliwe, że w niedalekiej przyszłości zobaczymy próbę zaadoptowania znanych algorytmów pod pamięci PCM takich jak kolumnowe drzewo FD [3], kolumnowe drzewo B+ [208] czy system zapisu danych PAX[51]. Sposób ten zapisuje dane kolumnowo w obrębie jednego fragmentu pamięci. Jako fragment pamięci najczęściej wybiera się stronę na dysku. Zamiast zapisać rekordy po kolei na wybranej stronie, zapisuje się najpierw wszystkie wartości atrybutu pierwszego, następnie drugiego itd. Dzięki temu podczas odczytu danej strony, uzyskujemy większą wydajność odczytu i dekompresji na procesorze, wedle zasady lokalności [209], [210] (ang. full cache utilization / less cache misses). Taki sposób zapisu można by łatwo zaadoptować do pamięci PCM, która w zależności od wykorzystania dzielona jest na mniejsze (64B) lub większe strony (4KB), odpowiadające stronom pamięci RAM czy dysku. W literaturze możemy znaleźć kilka prób implementacji nowego sposobu mapowania danych, tak aby efektywnie były wczytywane nie tylko z pamięci PCM (jak kiedyś z dysku) do pamięci RAM, ale także z PCM do pamięci podręcznej procesora [211], [212], [213], [214].

7.3 Indeksowanie częściowe

Zazwyczaj administrator danych podczas tworzenia bazy danych musi zdecydować jak dobrać indeksy, aby efektywnie obsługiwać kwerendy od użytkownika. Ponieważ jest to bardzo trudne zadanie, zdecydowano się na próbę automatyzacji tego procesu [53]. Coraz częściej stosuje się indeksowanie częściowe, czyli takie, które w wyniku kwerendy wyszukującej dodaje potrzebne dane do indeksu. W ten sposób rekordy zapisane w indeksie są tymi, które były kiedyś potrzebne i możliwe, że będą potrzebne w przyszłości. Dodatkowym atutem tego typu algorytmów jest amortyzacja kosztu tworzenia pełnego indeksu. Zamiast długiego oczekiwania na finalizacje tego procesu, dodajemy małe zbiory rekordów w każdym zapytaniu nieznacznie wydłużając tylko czas jego obsługi. Zwykle jest on niezauważalny przez użytkownika bazy. W [56] [57] zaproponowano Indeks Cracking. Algorytm ten używa metody podziału danych na partycje (ang. partition) względem elementów z zakresu zapytania i działa podobnie do sortowania szybkiego (ang. quicksort) [58]. Kolejna metoda automatycznego tworzenia indeksu podczas kwerend została zaproponowana w [13]. Adaptive Merging w przeciwieństwie do Indeks Cracking korzysta z algorytmu scalania posortowanych list tak jak sortowanie przez scalanie (ang. mergesort) [62]. Ze względu na schemat działania obu algorytmów, Indeks Cracking stosuje się dla baz danych zapisanych w szybkiej pamięci operacyjnej komputera, a Adaptive Merging gdy tabela przechowywana jest na pamięciach blokowych takich jak dyski HDD czy dyski SDD. Pamięć PCM adresowana jest bajtowo, zatem naturalnym wybór algorytmu do częściowego indeksowania padł na Indeks Cracking. W [215], [216] przeprowadzono dogłębną analizę algorytmu i charakterystyki PCM oraz pokazano za pomocą ogromnego zestawu eksperymentów, że Indeks Cracking poprzez częste reorganizacje doprowadza do szybkiego wypalenia się komórek pamięci PCM. Dodatkowo, biorąc pod uwagę asymetrię pomiędzy szybkim czasem odczytu i wolnym zapisu, częste nadpisywanie danych nie jest efektywne na tym rodzaju pamięci. Z tego powodu na pamięciach PCM używa się obecnie oryginalnego algorytmu Adaptive Merging.

Mimo, iż obecnie Adaptive Merging stosowany jest w bazach zapisanych na pamięci PCM, to nie jest on dostosowany do tego rodzaju pamięci i posiada wiele wad. Największą z nich jest reorganizacja partycji podczas przepisywania danych do nowej partycji. Każdy rekord, który jest przepisywany, musi zostać nadpisany innymi rekordami, tak aby partycja nie zawierała wolnych przestrzeni. Dodatkowo AM używa w pełni posortowanego rodzaju wariantu B+ (PB+) [63], a z [134] wiemy, że takie podejście nie jest odpowiednie w pracy z modelem PCM. Z tych powodów postanowiliśmy dostosować algorytm Adaptive Merging dla pamięci PCM. W [5] zaproponowaliśmy modyfikację tego algorytmu nazwaną eAM (extended Adaptive Merging). W eAM partycje dalej są zbiorami posortowanymi, jednak dopuszczamy elementy nieaktualne (usunięte), które dalej zapisane są w partycji. Każda partycja zawiera bitmapę, określającą status każdego rekordu. W chwili gdy przepisujemy element w inne miejsce, zmieniamy status rekordu w bitmapie, nie nadpisując go innymi elementami zbioru. Dzięki temu nie przeprowadzamy niepotrzebnych, kosztownych operacji zapisywania danych. Podczas wczytywania partycji najpierw wczytujemy do pamięci RAM lub pamięci podręcznej procesora (ang. cache) bitmapę, znajdującą się na początku partycji. Dzięki temu możemy określić pozycje rekordów, które nadal są aktualne i wymagają załadowania do pamięci. Drugą modyfikacją jest zmiana tworzonego indeksu. Zamiast posortowanej wersji B+, użyliśmy modyfikacji drzewa - UB+, opisanej w [134]. Wprowadzenie nieposortowanego liścia nie tylko zmniejsza liczbę zapisów

do pamięci PCM podczas bezpośredniego wstawiania nowych elementów do indeksu, ale także redukuje czas potrzebny na posortowanie danych odczytanych z kilku lub nawet kilkunastu partycji. Niestety dalsze modyfikacje algorytmu AM były wręcz niemożliwe do przeprowadzenia. Brak oddzielnych modułów na indeks i zbiór partycji utrudnia proces optymalizacji tego algorytmu (partycje należą do drzewa indeksu). Z tego powodu zaproponowaliśmy PAM (PCM Adaptive Merging) [5] - całkiem nowy system do tworzenia częściowego indeksu.

7.3.1 System PAM (PCM Adaptive Merging)

PCM Adaptive Merging [5] w przeciwieństwie do klasycznego Adaptive Merging nie nadpisuje danych podczas tworzenia nowej partycji. Zamiast tego wpisuje do dziennika parę kluczy odpowiadającą zakresowi przepisanemu z partycji do indeksu. Dzięki temu system unika kosztowej operacji nadpisywania danych w momencie ich przepisywania. System PAM potrafi współpracować z dowolnym indeksem, który wspiera zbiorcze dodawanie danych (ang. bulkload). Należą do nich m.in. DPTree [207] i FPTree [206]. Oba indeksy zostały przystosowane do charakterystyki pamięci PCM. Niestety nie potrafią one w pełni wykorzystać wzorca dodawanych rekordów (zakresy kluczy). Z tego powodu w pracy zaproponowaliśmy również nową strukturę opartą na drzewie B+Tree i DPTree - BB+Tree (Buffered B+Tree).

System PCM Adaptive Merging korzysta z następujących struktur danych:

1. **Partycja** - podstawowa struktura przechowująca zbiór posortowanych danych. Mimo iż pamięć PCM jest bajtowo adresowalna, to dostęp do całej linii pamięci jest o wiele szybszy ze względu na buforowanie kontrolera pamięci. Dlatego wielkość partycji zawsze jest wielokrotnością linii pamięci PCM, aby zminimalizować narzut związany z zapisem i odczytem zbyt małych obszarów pamięci. Podczas pierwszej kwerendy dane wczytywane są do bufora znajdującego się w pamięci RAM i tam sortowane. Następnie zapisywane są na pamięci nieulotnej PCM. Dane są posortowane wewnątrz każdej partycji, zatem możemy wyszukać potrzebny klucz w czasie logarytmicznym używając klasycznego wyszukiwania binarnego. Dodatkowo każda partycja posiada dwa atrybuty, najmniejszą i największą wartość klucza jaka jest w partycji. Atrybuty te zapisane są w pamięci RAM w metadanych zbioru partycji.

2. **Dane partycji** - struktura przechowująca potrzebne metadane o każdej partycji. W zależności od implementacji i wielkości zbioru, może być to lista lub struktura drzewa. Każdy rekord w tej strukturze posiada klucz sztuczny partycji, adres w pamięci oraz wartości minimalne i maksymalne kluczy w danej partycji. Wyszukiwanie danych może odbywać się równoległe na kilku partycjach, ponieważ możemy w łatwy sposób na podstawie zgromadzonych danych stwierdzić, które partycje z podanych adresów należy wczytać, a które nie. Struktura zapisana jest w pamięci RAM, ponieważ można ją łatwo zbudować ze zbioru partycji zapisanych na nieulotnej pamięci.

3. **Bufor do sortowania** - fragment szybkiej pamięci RAM o ustalonym rozmiarze BS (ang. Buffor Size). Używany do sortowania danych przed zapisem do partycji. Im większy bufor, tym więcej danych można posortować za jednym razem. Zatem im większe BS tym mniej będzie partycji w zbiorze.

4. **Indeks** - do indeksowania można użyć dowolnej struktury, która jest dostosowana

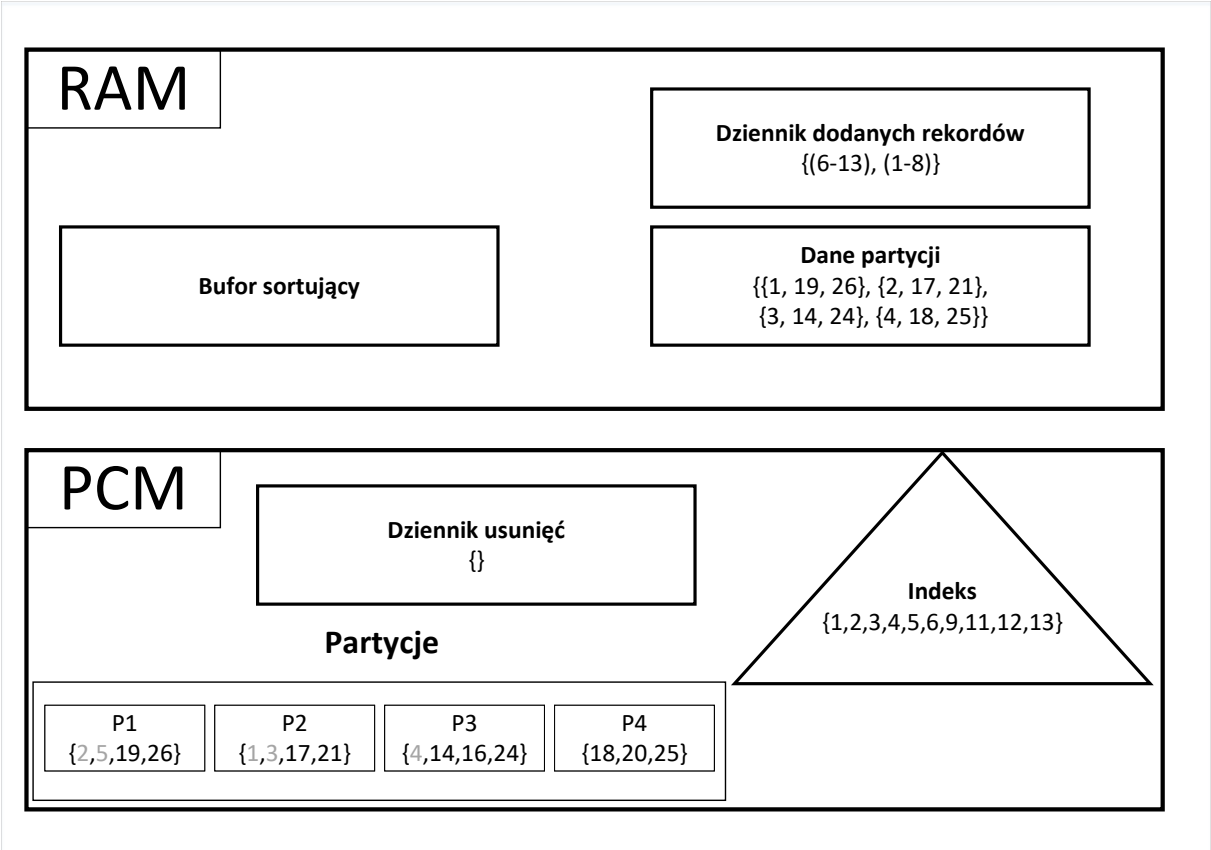
do pamięci PCM oraz posiada wszystkie cechy wymagane nasz system. Domyślnie używamy nowego drzewa BB+.

5. **Dziennik dodanych rekordów (ang. insertion journal)** - dziennik ten zawiera wszystkie zakresy kluczy jakie zostały przepisane z partycji do indeksu. Samo wyszukiwanie w indeksie jest stosunkowo tanie, jednak skan partycji jest kosztowny. Aby zredukować te koszty, podczas wyszukiwania najpierw czytamy dziennik, który przechowywany jest w pamięci RAM. Na jego podstawie możemy stwierdzić, czy rekordy zostały wcześniej przepisane do indeksu czy też nie. Dzięki temu wczytujemy tylko potrzebne partycje. Tej struktury nie trzeba zapisywać w nieulotnej pamięci. Wystarczy podczas startu systemu odczytać rekordy znajdujące się w zbiorze partycji jak i w indeksie. Część wspólna tych zbiorów została przepisana z partycji do indeksu. Zatem należy dodać do dziennika zakresy tych zbiorów. Oczywiście tak utworzony dziennik nie będzie zawierał dokładnie tych samych danych co oryginalny (np. niektóre kwerendy zostaną połączone). Jednak nie wpływa to na poprawność działania systemu, a dokładne dane o kwerendach nie są potrzebne.

6. **Dziennik usuniętych rekordów (ang. deletion journal)** - dziennik ten zawiera wszystkie zakresy kluczy jakie zostały usunięte z partycji. Usuniętych danych nie da się odzyskać w podobny sposób do danych zapisanych w dzienniku dodanych rekordów. Dlatego dziennik ten zapisany jest w nieulotnej pamięci RAM. Dzięki niemu nie musimy fizycznie usuwać danych z partycji. Zamiast tego wpisujemy odpowiedni klucz lub zakres kluczy do dziennika. Podczas wyszukiwania sprawdzamy, czy żądane przez użytkownika rekordy nie znajdują się w dzienniku. Jeśli tak, to odejmujemy je z wynikowego zbioru kwerendy, ponieważ zostały one wcześniej usunięte.

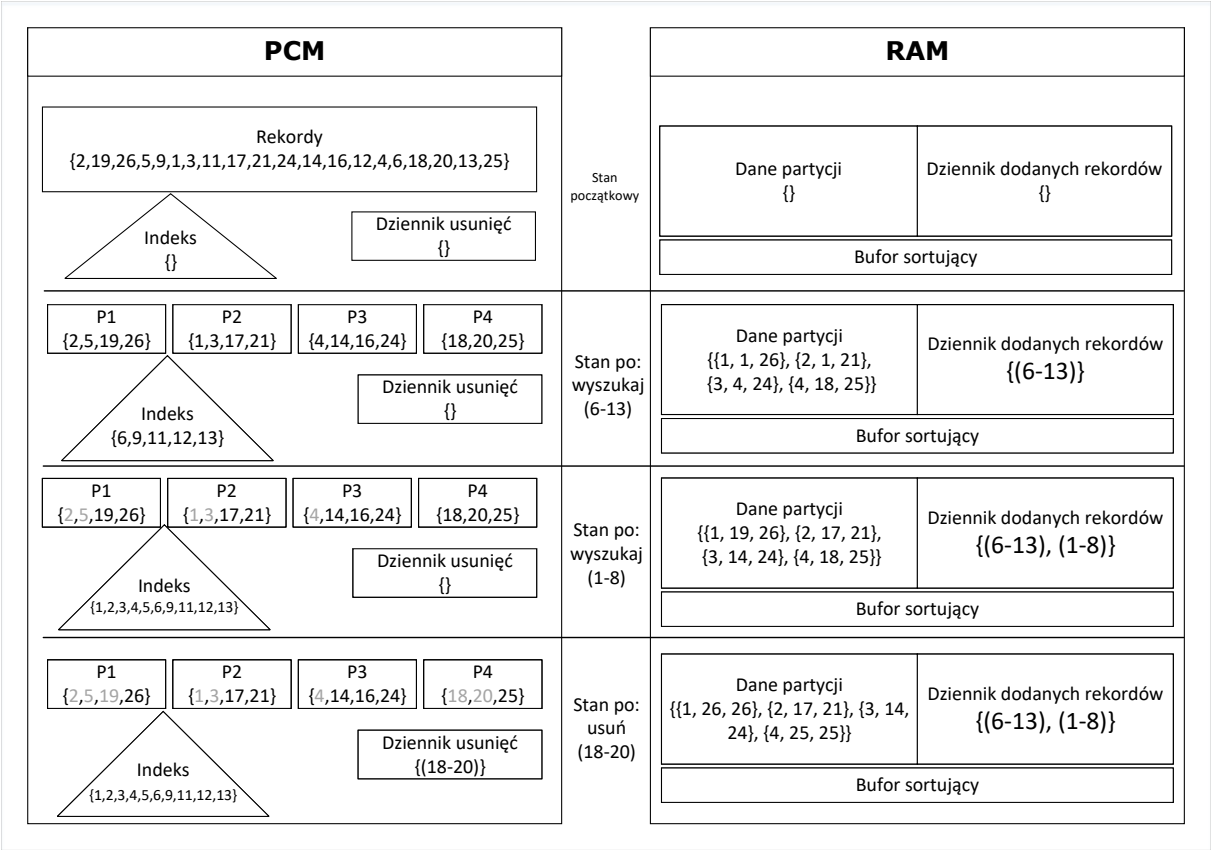
Rysunek 7.1 przedstawia przykładowy stan procesu tworzenia indeksu częściowego przez PAM. Dla uproszczenia załóżmy, że tabela ma 20 rekordów. Początkowo układ danych w pamięci był następujący: {2, 19, 26, 5, 9, 1, 3, 11, 17, 21, 24, 14, 16, 12, 4, 6, 18, 20, 13, 25}. Załóżmy także, że linia pamięci PCM jest w stanie pomieścić maksymalnie 2 rekordy, a bufor sortujący ma pojemność 5 rekordów. Zatem wielkość partycji wyrażona w rekordach musi być mniejsza bądź równa 5 oraz musi być wielokrotnością 2. Pojemność równa czterem rekordom spełnia oba te warunki. Rysunek opisuje sytuację po dwóch kwerendach wyszukujących: (6-13) oraz (1-8), co możemy łatwo odczytać z dziennika dodanych rekordów. W zbiorze partycji składającym się z 4 partycji oznaczonych na rysunku P1, P2, P3 i P4 rekordy przepisane do indeksu oznaczone są kolorem szarym. Jak widzimy, wszystkie dane, te które zostały przepisane jak również te, które nie zostały pobrane przez użytkownika są zapisane w partycjach w pamięci PCM. Mimo że przepisane rekordy nie są już potrzebne, to o wiele lepiej jest zostawić je na miejscu i nie nadpisywać ich danymi aktualnymi. Dzięki temu unikniemy niepotrzebnych operacji zapisywania danych, które są bardzo kosztowne w tym modelu pamięci. Pamięć zostanie zwolniona, gdy wszystkie dane przepisane do indeksu w obrębie jednej partycji zostaną już usunięte na prośbę użytkownika. Warto również zauważyć, że metadane o zbiorze partycji są na bieżąco aktualizowane wraz z każdą zmianą przeprowadzoną w partycji. Przykładowo, mimo iż partycja P1 zawiera dane {2,5,19,26}, to minimalny klucz zapisany w metadanych to 19, a nie 2. Wynika to z faktu, iż wartości 2 i 5 są już nieaktualne i nie powinny być brane pod uwagę podczas kolejnych operacji wykonywanych na tej partycji.

Rysunek 7.2 przedstawia trzy kroki systemu PCM Adaptive Merging podczas tworze-



Rysunek 7.1: Przykład struktury PAM

nia indeksu częściowego. Zakładamy, że początkowo mamy tę samą sytuację co podczas analizy rysunku poprzedniego. Tabela składała się z 20 rekordów: {2, 19, 26, 5, 9, 1, 3, 11, 17, 21, 24, 14, 16, 12, 4, 6, 18, 20, 13, 25}. Ze względu na pojemność linii pamięci PCM (2 rekordy) oraz pojemności bufora sortującego (5 rekordów), rozmiar partycji został ustawiony na wielkość 4 rekordów. Na początku dostaliśmy kwerendę wyszukującą klucze z zakresu (6-13). Dane wczytywane są do bufora i sortowane, następnie zapisywane w partycjach. Sam zakres kwerendy dodawany jest do indeksu. Dlatego w indeksie zapisaliśmy rekordy {6,9,11,12,13}, a także utworzyliśmy pierwszy wpis w dzienniku, parę (6 - 13). Następnie mamy zapytanie o zakresie (1 - 8). Po pierwsze czytamy nasz dziennik, aby sprawdzić czy zakres zapytania pokrywa się z kluczami zapisanymi już w indeksie. Widzimy, że wartości 7 i 8 zostały już przepisane wcześniej. Dlatego zapytanie (1 - 8), możemy traktować jako kwerendę z zakresu (1 - 6). Dzięki atrybutom partycji zapisanych w pamięci RAM stwierdzamy, że partycja P4 nie może zawierać szukanych rekordów, ponieważ klucze zapisane w tej partycji są z zakresu (18 - 25). Pozostałe partycje: P1 (1-26), P2 (2-21) oraz P3(4-24), mogą zawierać szukane dane, dlatego musimy wczytać je z pamięci PCM do bufora. Po odczytaniu danych kopiujemy z partycji P1 wartości 2 i 5, z partycji P2 wartości 1 i 3 oraz wartość 4 z partycji P3. Skopiowane dane dodajemy do naszego indeksu. Zamiast usuwać przepisane dane za pomocą nadpisywania ich danymi aktualnymi, co wywoła kosztowne operacje zapisu, zostawiamy wszystkie dane na miejscu i dodajemy odpowiedni wpis do dziennika dodanych rekordów. Zakres (1-8) w dzienniku oznacza, że dane z tego zakresu zostały przepisane do indeksu i nie trzeba ich ponownie odczytywać z partycji. Ostatnią przedstawioną na rysunku kwerendą jest usuwanie wartości z zakresu (18-20). Jak poprzednio, czytamy najpierw dziennik. Widzimy, że wszystkie



Rysunek 7.2: Przykładowy proces tworzenia częściowego indeksu z użyciem systemu PAM

usuwane dane pochodzą z partycji. Odczytujemy zatem metadane i stwierdzamy, że rekordy, które należy usunąć mogą znajdować się w każdej z partycji. Podczas wczytywania danych z pamięci PCM, okazało się, że z partycji P1 należy usunąć wartość 19, a z partycji P4 wartości 18 i 20. Mimo iż użytkownik kazał nam usunąć rekordy o podanym zakresie kluczy, to system PAM tego nie robi. Zamiast tego dodaje zakres (18-20) do dziennika usunięć oraz aktualizuje metadane partycji.

7.3.2 Procedury systemu PAM

Wstawianie

Dodawanie nowych rekordów do systemu PAM jest bardzo proste. Używamy bowiem gotowej struktury danych do indeksowania rekordów. Wstawianie odbywa się tak jak opisano w używanej strukturze indeksowej. W ten sposób nie zaburzamy porządku posortowanych partycji, co prowadziłoby do reorganizacji całej partycji. Nie dodajemy również nowych partycji, ponieważ obsługa tej struktury w procesie tworzenia indeksu jest zbyt kosztowna. Wiemy, że użytkownik chce skorzystać z rekordu w przyszłości, ponieważ zdecydował dodać się go do bazy. Dlatego od razu wstawiamy go do tworzonego indeksu.

Wyszukiwanie

Wyszukiwanie to najważniejsza operacja w procesie tworzenia indeksu częściowego. To właśnie podczas wyszukiwania dokładamy kolejne rekordy do indeksu zmniejszając tym samym liczbę aktualnych danych w partycjach. Procedura jest prosta. Gdy dostajemy kwerendę od użytkownika, czytamy dziennik dodanych rekordów i próbujemy określić, które dane możemy wczytać z gotowego indeksu, a które z partycji. Ponieważ dziennik to tylko lista zakresów kluczy przepisanych, musimy odczytać całą listę. Oczywiście najlepszym przypadkiem jest zapytanie, które możemy obsłużyć czytając tylko indeks. Gdy jednak tak nie jest, musimy wczytać dane z partycji. Za pomocą metadanych gromadzonych w pamięci RAM o każdej z partycji, odczytujemy z pamięci tylko te partycje, które mogą zawierać potrzebne wartości. Zatem w fazie przygotowania zbieramy wszystkie potrzebne informacje zapisane w pamięci komputera, a następnie wczytujemy równolegle potrzebne obszary pamięci. To że klucz zapisany został w dzienniku nie oznacza, że nadal jest w indeksie. Podczas usuwania lub aktualizacji rekordów w indeksie zmieniamy indeks, jednak nie zmieniamy dziennika. Robimy tak, aby przyspieszyć proces wyszukiwania oraz zachować jego poprawność. Gdybyśmy usunęli dane z dziennika, musielibyśmy odszukać go w zbiorze partycji. Ponieważ nie usuwamy danych natychmiastowo, to możliwe że znajdziemy klucz, który miał być usunięty. Podamy wtedy nieprawdziwą informację użytkownikowi. Nie usuwając wpisu, w czasie logarytmicznym wyszukujemy klucz w indeksie lub dochodzimy do wniosku, że dany klucz został usunięty, co udowadnia poprawność tej operacji. Pamiętamy, że nie wszystkie rekordy zapisane w partycji są aktualne. Niektóre z nich mogły zostać wcześniej usunięte na prośbę użytkownika. Dlatego zanim dodamy rekord do zbioru wynikowego, sprawdzamy czy nie znajduje się w dzienniku usunięć. Jeśli tak, to nie dodajemy go do wyniku kwerendy. Gdy mamy wszystkie potrzebne rekordy, sortujemy je i dodajemy do indeksu. Jednocześnie aktualizujemy metadane partycji oraz dodajemy kolejny wpis w dzienniku dodanych rekordów odpowiadający przepisanym właśnie kluczom. Dopiero po przepisaniu danych, możemy zwrócić zbiór wynikowy użytkownikowi i zakończyć obsługę kwerendy.

Usuwanie

System PAM równoległe obsługuje struktury partycji jak i indeksu, który tworzy. Za pomocą dziennika dodanych rekordów jest w stanie stwierdzić, którą strukturę należy oczytać aby znaleźć rekordy, które należy usunąć. Gdy klucz zapisany jest w indeksie, wystarczy wykonać potrzebną operację na wybranej do indeksowania strukturze. Gdy rekord zapisany jest w partycji, nie należy go usuwać z pamięci PCM. Zamiast tego należy dodać wpis do dziennika usunięć, aby podczas wyszukiwania algorytm zdecydował się na odczyt indeksu.

7.3.3 Opis algorytmów

Pseudokod 7.1 przedstawia dokładny opis kroków algorytmu wyszukującego dane w systemie PAM. Procedura na wejściu przyjmuje posortowany zbiór kluczy, dla którego musimy znaleźć odpowiadające rekordy. W pierwszym kroku próbujemy znaleźć rekord w indeksie (linia 5). Jeśli indeks nie zawiera danej wartości, przechodzimy do kolejnego kroku. Jeśli klucz był zapisany w strukturze, to dodajemy go do zbioru wynikowego (linie 6-7). Kolejnym krokiem jest odczyt partycji. Wiemy, że klucz znajduje się w partycji, jeśli nie został przepisany do indeksu (dziennik dodanych rekordów nie zawiera szukanego

Pseudokod 7.1: pamFind(input: Key *keys*[ ])

```
1  Entry result[ ] := ∅
2  Entry toInsert[ ] := ∅
3
   // na podstawie dziennika określ, które klucze wczytujemy z partycji
4  foreach Key k ∈ keys do
5      Seek Entry e with key k in ind
6      if e exists in index then
7          result := result ∪ e
8
9      if insertionJournal.contains(k) = FALSE and deletionJournal.contains(k) =
         FALSE then
           // Wybór partycji dokonujemy za pomocą danych partrycji w RAM (pid, min, max)
10         Select Partition partitions[] containing k
11         foreach Partition p ∈ partitions do
12             Seek Entry e with key k in p
13             if e exists in p then
14                 toInsert := toInsert ∪ e
15                 result := result ∪ e
16
   // Wstaw nowe dane, aby powiększyć częściowy indeks
17 indexBulkloadInsert(toInsert)
18
   // Aktualizuj dane w RAM
19 Pair p := (min(keys), max(keys))
20 insertionJournalAdd(p)
21 updateParitionsData()
22
23 return result
```

klucza) oraz jeśli nie został usunięty z partycji (dziennik usunięć nie zawiera szukanego klucza). Gdy spełniamy oba te warunki (linia 9), to musimy podjąć próbę znalezienia rekordu w zbiorze partycji. Pamiętajmy, że nie musimy wczytywać wszystkich partycji. W pamięci RAM zapisane mamy aktualne metadane. Na ich podstawie, możemy stwierdzić, czy klucz może znajdować się w partycji czy też nie (linia 10). Gdy wczytujemy dane z partycji, za pomocą wyszukiwania binarnego (linia 12) jesteśmy w bardzo szybkim czasie stwierdzić czy rekord jest zapisany w partycji czy też nie. Jeśli tak, to dodajemy go nie tylko do zbioru wynikowego (linia 15), ale także do zbioru rekordów, które będą przepisane do indeksu w kolejnym kroku (linia 14). Ostatnim krokiem wyszukiwania jest przepisanie znalezionych danych do indeksu. W tym celu najpierw dodajemy rekordy do struktury (linia 17), następnie dodajemy wpis do dziennika dodanych rekordów (linie 19-20) oraz aktualizujemy metadane każdej partycji (linia 21).

7.3.4 Wybór Indeksu dla systemu PAM

System PAM potrafi współpracować z dowolnym indeksem. Operacje, które wykonujemy na strukturze PAM dzielą się na takie, które musimy wykonać na zbiorze partycji oraz na te, które wykonujemy na strukturze indeksu. System nie wymaga żadnej specyficznej dla struktury operacji. Wykorzystuje jedynie podstawowe procedury takie jak dodawanie, usuwanie i wyszukiwanie elementów w indeksie. Z tego powodu możemy użyć dowolnego indeksu w systemie PAM. Jednak nie każdy wybór będzie tak samo dobry. Po pierwsze powinniśmy wybrać taką strukturę, która jest dostosowana do modelu pamięci PCM. Takimi strukturami są m.in. drzewo UB+ [134], OB+ [27] i DP [207]. Po drugie, struktura powinna wspierać operację dodawania zbiorczego (ang. bulkload), ponieważ kopiowanie danych z partycji do indeksu zazwyczaj odbywa się z udziałem dużych zbiorów danych. Po trzecie, ze względu na charakterystykę pamięci PCM (wolny zapis, szybki odczyt), struktura powinna buforować te części drzewa, które są często wczytywane, ale nie modyfikowane, w szybkiej pamięci RAM. Wszystkie trzy kryteria spełnia tylko indeks DP. Niestety DP-Tree nie jest dostosowany do systemów tworzenia indeksów częściowych. Nie jest w stanie w pełni wykorzystać faktu, iż podczas kopiowania dane można podzielić na posortowane podzbiory, które można w szybki sposób dodać do indeksu. W przypadku, gdy podzbiór zapisany jako zakres wartości tworzy część wspólną z wartościami zapisanymi już w drzewie, trzeba dodać rekordy do istniejącego węzła. W przypadku, gdy jest to zbiór składający się z nowych wartości, wystarczy stworzyć nowy węzeł i podpiąć go do struktury drzewa. Zamiast tego, drzewo DP buforuje dane w pamięci RAM, a następnie uniwersalnym i bardzo kosztownym algorytmem dodaje wszystkie rekordy z bufora do głównego drzewa zapisanego na pamięci PCM. Z tego powodu, w pracy poświęconej systemowi PAM [5] zaproponowaliśmy również indeks BB+ (Buffered B+), który nie tylko spełnia wszystkie trzy wcześniej wymienione kryteria, ale także dostosowany jest do systemów tworzenia częściowych indeksów.

7.3.5 Nowy indeks BB+ (buffered B+-Tree)

Struktura BB+ jest modyfikacją popularnego drzewa B+ inspirowaną po części indeksem DP-Tree. W szybkiej pamięci RAM zapisujemy dane, które powinny zostać dodane oraz te, które powinne zostać usunięte z głównego indeksu. Dodatkowo węzły wewnętrzne, które głównie służą do wyznaczenia ścieżki z korzenia do liścia również są buforowane, podobnie jak to ma miejsce w strukturze DP. Liście drzewa zapisane są zawsze w pamięci PCM i nigdy nie są buforowane, ponieważ zawierają rekordy dodane przez użytkownika. Utrata tych danych podczas awarii systemu jest niedopuszczalna. Wszystkie pozostałe struktury można łatwo odzyskać. Węzły wewnętrzne odzyskujemy budując drzewo na nowo na podstawie liści. Bufor dodanych i usuniętych rekordów możemy odzyskać z systemowego dziennika transakcji, dostępnego w popularnych systemach operacyjnych takich jak Windows i Linux oraz w silnikach baz danych np. MySql czy Postgress. Podczas analizy kwerend wykonywanych w systemie PAM zauważyliśmy, że raz zbudowany węzeł nie jest tak często modyfikowany jak w przypadku zwykłego indeksu. Dzieje się tak, ponieważ dane kopiowane z partycji do indeksu są rozłączne z rekordami już skopiowanymi. Zatem zbudowany węzeł bardzo często jest odczytywany i jednocześnie rzadko modyfikowany. Z tego względu zdecydowaliśmy się na podział węzła w drzewie BB+ na dwie sekcje: posortowaną i nieposortowaną. Początkowo gdy tworzymy węzeł, czy to przez kopiowanie danych z partycji, czy przez reorganizację drzewa BB+, podobną do znanej z indeksu B+, zapisujemy dane do części posortowanej. Kolejne odczyty tego węzła będą

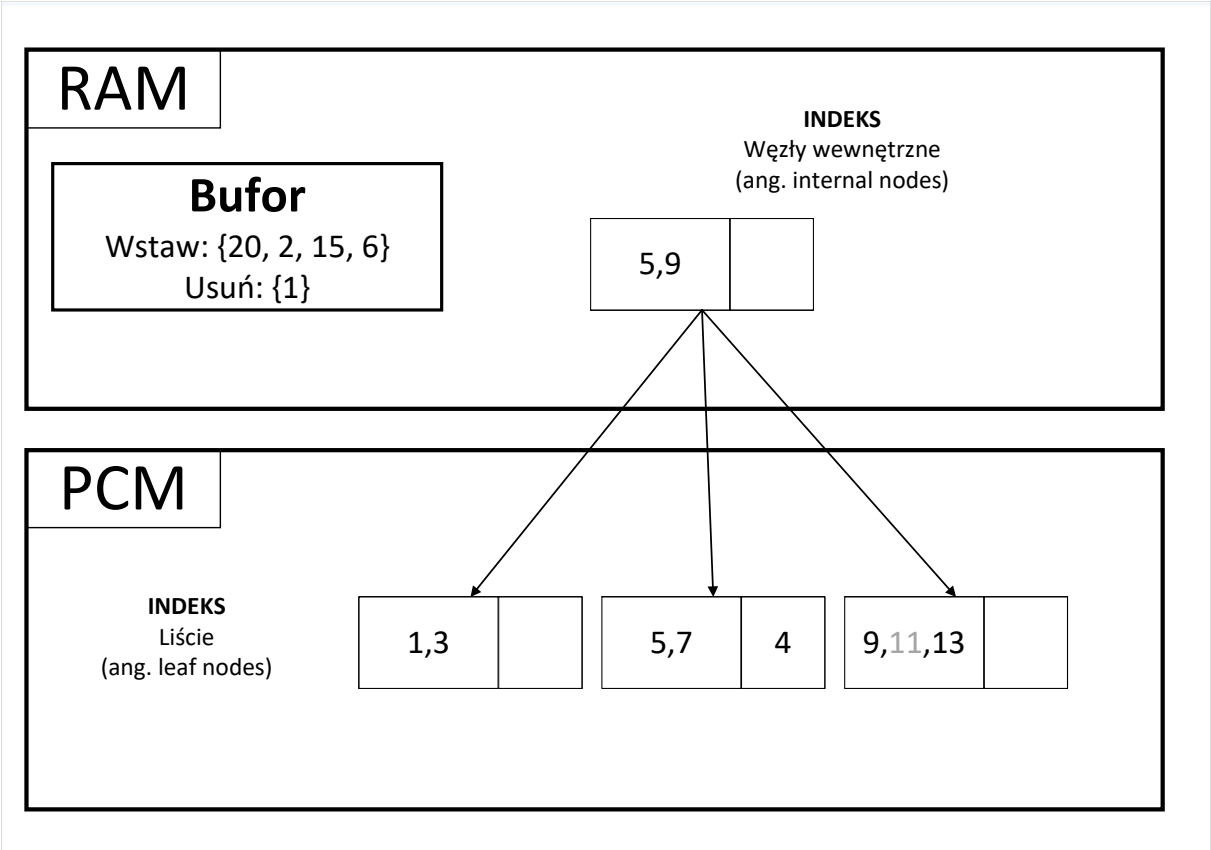
bardzo szybkie, ponieważ przejście po węźle odbywać się będzie za pomocą wyszukiwania binarnego. Podczas modyfikacji istniejącego węzła, nowe dane dodawane są do sekcji nieposortowanej tak jak w przypadku drzewa UB+. Tak samo jak w drzewie UB+, indeks BB+ posiada bitmapę dla każdego węzła mówiącą które dane są aktualne. Dzięki temu zamiast reorganizować węzeł podczas każdego usuwania, zaznaczamy fakt usunięcia w bitmapie i kończymy procedurę usuwania. Jako efekt uboczny, możliwe że powstanie puste miejsce w środku jakieś sekcji. Wypełnianie sekcji nieposortowanej jest tak samo tanie jak dodawanie rekordu na koniec węzła. Zatem traktujemy wypełnienie pustych miejsc w tej sekcji jako priorytet. Niestety, jeśli dodamy rekord w środek sekcji posortowanej, utracimy porządek tej sekcji. Wtedy przesuwamy wskaźniki obu sekcji zmniejszając sekcje posortowaną i zwiększając nieposortowaną, tak aby nowy element oraz wszystkie elementy na prawo od niego znalazły się w sekcji nieposortowanej.

Podsumowując nowa struktura BB+ posiada następujące cechy:

- Drzewo BB+ jest modyfikacją drzewa B+ w pełni dostosowaną do systemu PAM i pamięci PCM,
- W pamięci RAM buforuje elementy, które trzeba dodać i usunąć z głównego indeksu,
- Węzły wewnętrzne (ang. internal nodes) buforowane są w pamięci RAM
- Liście zapisane są w nieulotnej pamięci PCM
- Każdy węzeł podzielony jest na 2 sekcje: posortowaną i nieposortowaną,
- Rozmiar węzła dostosowany jest do architektury wykorzystania pamięci PCM. Jest wielokrotnością linii pamięci podręcznej procesora (ang. cacheline) lub jest równa stronie w pamięci RAM

Rysunek 7.3 przedstawia przykładowe drzewo BB+. Obecnie w buforze przetrzymujemy cztery rekordy, które należy dodać do drzewa oraz jeden rekord, który należy usunąć. Sama struktura podzielona jest na warstwy węzłów wewnętrznych, która również buforowana jest w szybkiej pamięci RAM oraz warstwę liści, która zapisana jest w pamięci PCM. Każdy węzeł podzielony jest na dwie sekcje. Sekcja posortowana znajduję się w lewej części węzła, a nieposortowana w prawej części. Przykładowo wartości 5 i 7 należą do części posortowanej liścia środkowego, a klucz o wartości 4 jest zapisany w sekcji nieposortowanej. Widzimy również, że wartość 11 w ostatnim liściu oznaczona jest kolorem szarym a nie czarnym. Oznacza to, że rekord 11 został usunięty przez użytkownika, co zostało zaznaczone w bitmapie, nie wykonując przy tym fizycznego usuwania w pamięci. W następnych operacjach dodawania rekordów do tego węzła puste miejsce zostanie zapełnione przez nowe rekordy.

Pseudokod 18 opisuje procedurę dodawania całego zbioru elementów do drzewa BB+. Algorytm wykonujemy rekurencyjnie, a więc na początku musimy wyznaczyć warunki końca. Pierwszym warunkiem jest liczebność zbioru wejściowego rekordów. Jeśli zbiór ten jest pusty (linia 1), nie mamy czego dodać do obecnego poddrzewa i możemy skończyć ciąg wykonań rekurencyjnych dla tej części drzewa. Drugim warunkiem jest rodzaj węzła. Gdy w końcu osiągniemy poziom liści (linie 4-6), nie szukamy poddrzewa dalej, ponieważ właśnie znaleźliśmy właściwy węzeł. Zatem należy wykonać dodawanie elementów do wyznaczonego liścia, za pomocą funkcji *bbpLeafInsert* (algorytm 19). Od linii 8 mamy właściwe ciało funkcji *bbpBulkload*. Celem każdego wywołania procedury jest wyznaczenie podzbioru elementów na podstawie wartości kluczy oraz wywołanie rekurencyjne



Rysunek 7.3: Przykładowy indeks BB+

algorytmu na poddrzewie i wyznaczonym podzbiorze. Wiemy, że w drzewach B+ każdy wskaźnik wskazuje na węzły (poddrzewa) o wartościach mniejszych niż klucz wskaźnika. Dokładnie z tej własności drzewa korzystamy. Celem jest dodanie elementów do drzewa zachowując wszystkie własności struktury. Dlatego podzbiory elementów, które powinny być dodane do drzewa wyznaczane są w takim sam sposób jak ścieżka podczas wyszukiwania. Algorytm jest następujący. Najpierw ustalamy wartość ostatniego maksimum (potrzebny do wyznaczania następnego minimum) jako $-\infty$ (linia 8). Następnie przechodzimy po wszystkich wskaźnikach danego węzła (linia 10). Za każdym wywołaniem pętli ustalamy minimum zbioru jako poprzednie maksimum, a maksimum zbioru jako wartość obecnego wskaźnika. Pamiętamy jednak, że wartość wskaźnika nie znajduje się w poddrzewie, na który wskazuje. Dlatego też tworzymy podzbiór z zakresu $< min,max)$ lewostronnie domkniętego (linia 16). Gdy mamy stworzony podzbiór, wykonujemy procedurę rekurencyjnie dla poddrzewa obecnego wskaźnika i utworzonego podzbioru.

Aby zminimalizować liczbę zapisów na pamięci PCM oraz zmaksymalizować zyski wynikające ze wstawiania dużych posortowanych zbiorów danych do drzewa, struktura BB+ składa się z węzłów podzielonych na 2 sekcje. Sekcja posortowana tworzona jest na początku życia węzła. Gdy alokujemy pamięć na nowy węzeł i zapisujemy do niego pierwsze posortowane elementy, wszystkie rekordy trafiają do sekcji posortowanej. Dzięki temu, gdy dominują kwerendy wyszukujące dane, możemy bardzo szybko przeprowadzić wyszukiwanie szukanego klucza. Jednocześnie, tak jak wspomnieliśmy, aby zminimalizować liczbę zapisów, nowe rekordy dodawane są głównie do sekcji nieposortowanej. Pseudokod 19 zawiera listę kroków algorytmu dodającego nowe klucze do węzła. Jako argument wejściowy procedura przyjmuje węzeł, na którym musi pracować, oraz zbiór danych, który trzeba

Pseudokod 7.2: bbpBulkload(input: Node *node*, Entry *entries*[ ])

```
1  if length(entries) = 0 then
2      return
3
4  if isLeaf(node) = TRUE then
5      bbpLeafInsert(node, entries)
6      return
7
8  Key lastMax := −∞
9
   // Idziemy z góry na dół do odpowiednich węzłów z podzbiorem o wartościach z zakresu
      < minKey,node.keys[i])
10 for i ← 1 to FANOUT do
11     Key minKey := lastMax
12
       // node.keys[FANOUT] ustalamy jako ∞
13     Key maxKey := node.keys[i]
14     lastMax := maxKey
15
       // Stwórz podzbiór slice z entries zawierający dane z zakresu < minKey, maxKey)
16     Entry slice[ ] := sliceMake(entries, minKey, maxKey)
17
       // Wykonaj rekurencyjnie
18     bulkload(node.child[i], slice)
```

dodać lub usunąć (zbiór rekordów może zawierać dane do usunięcia). Mimo, iż algorytm modyfikuje węzeł kilkukrotnie (w każdej iteracji pętli), to pamiętajmy że w rzeczywistości węzeł jest buforowany. Z pamięci PCM odczytamy potrzebne fragmenty węzła do pamięci RAM, tam przeprowadzimy wszystkie modyfikacje, a następnie zaaplikujemy zmiany na pamięci PCM. Początkowo w każdej iteracji pętli sprawdzamy czy modyfikacje węzła nie zaburzyły jego podstawowych własności. Jeśli węzeł jest zbyt mały, to należy go scalić z innymi węzłami (linie 3-5). Analogicznie, jeśli węzeł nie jest w stanie zmieścić kolejnego rekordu, jest on dzielony na dwa węzły (linie 7-9). Gdy jesteśmy pewni, że możemy przeprowadzić operację na węźle, sprawdzamy czy rekord, który obecnie odczytaliśmy ze zbioru wejściowego jest do usunięcia czy do dodania. Jeśli musimy usunąć rekord (linie 11-12), to wyłączamy odpowiedni bit w bitmapie ale nie usuwamy rekordu z pamięci. Dzięki bitmapie wiemy, że usunięty rekord jest nieaktualny i nie będziemy go brali pod uwagę podczas wyszukiwania czy następnego usuwania. Gdy dodajemy rekord, podążamy wedle ustalonych priorytetów. Po pierwsze, jeśli istnieje przestrzeń do wypełnienia w sekcji nieposortowanej (linie 14-15), to należy ją wypełnić. Po drugie, jeśli sekcja nieposortowana nie jest pełna to dodajemy kolejny rekord na koniec tej sekcji (linie 16-17). W momencie gdy sekcja nieposortowana nie jest w stanie pomieścić kolejnego rekordu (linie 18 - 19), musimy dodać element do przestrzeni w sekcji posortowanej (przypadek, gdy sekcja posortowana jest również pełna nie jest obsługiwany, ponieważ to oznacza, że węzeł należało podzielić). Niestety, po dodaniu elementu do części posortowanej, zaburzyliśmy porządek tej sekcji. Należy zatem zmniejszyć część posortowaną i jednocześnie zwiększyć nieposor-

Pseudokod 7.3: bbpLeafInsert(input: Node *node*, Entry *entries*[ ])

```
1  Node n = node
2  foreach Entry e ∈ entries do
       // Dodanie danych do usunięcia może doprowadzić do posiadania zbyt małego węzła. W
          tym przypadku scalamy węzły i aktualizujemy węzeł rodzica
3      if numValidEntries(n) < FANOUT/2 then
4          n = MergeNodes()
5          UpdateInnerNodes()
6
       // Dodanie nowych danych, może doprowadzić do przepełniani węzła. Wtedy dzielimy
          węzeł i aktualizujemy wskaźniki w węźle rodzica
7      if numValidEntries(n) > FANOUT then
8          n = SplitNode()
9          UpdateInnerNodes()
10
11     if e.ToDelete = TRUE then
12         delete(n,e)
13     else
           // Wstawianie danych do posortowanej części węzła jest kosztowne. Dlatego najpierw
              sprawdzamy, czy istnieje miejsce w części nieposortowanej
14         if existsGapInUnsorted(n) = TRUE then
15             insertIntoUnsortedGap(n,e)
16         else if unsortedNotFull(n) = TRUE then
17             insertIntoUnsorted(n,e)
18         else if existsGapInSorted(n) = TRUE then
19             insertIntoSortedGap(n,e)
```

towaną. Opisane kroki wykonujemy dla każdego elementu znajdującego się zbiorze danych wejściowych.

7.3.6 Eksperymenty

W tej części rozdziału przedstawimy i przeanalizujemy eksperymenty wykonane za pomocą symulatora SIPS, który został dokładnie opisany w rozdziale 4. Do eksperymenty wykorzystano model referencyjny pamięci PCM przedstawiony w [167]. Ten sam model został użyty do testów w [134], [161], [217]. Jest to zatem bardzo popularny model, który używa się do przedstawienia wyników w sytuacji, gdy nie mamy możliwości użycia realnej kości pamięci, która wymaga zakupu dość drogiego procesora (intel xeon gold lub platinium). Referencyjny model przedstawiony w tabeli 7.1 charakteryzuje się pojemnością

Model	Pojemność strony	Szybkość odczytu	Szybkość zapisu
Testowy Model PCM	64B	6,4GB/s	1,6GB/s

Tabela 7.1: Parametry testowego modelu pamięci PCM [167]

strony (linii pamięci) równej pojemności strony pamięci podręcznej procesora (ang. cache-line). Ponieważ procesor wczytuje pełne strony pamięci podręcznej, to równa ich wielkość wpływa korzystnie na szybkość wczytywania danych z pamięci PCM. Dodatkowo model ten posiada szybkość zapisu na poziomie 1,6GB/s oraz szybkość odczytu równą 6,4GB/s. Jak widzimy, model posiada również bardzo dużą asymetrię pomiędzy wolną modyfikacją pamięci a bardzo szybkim odczytem. Jest to charakterystyka dobrze nam znana z pamięci flash i dysków SSD z niej zbudowanych. Wiemy zatem, iż algorytmy minimalizujące liczbę zapisów osiągną największe korzyści z redukcji tej operacji. W testowanej bazie danych rola pamięci PCM jest podobna do zwykłego dysku. Jest to pamięć nieulotna, w której zapisujemy wszystkie dane trwałe. Nie służy ona do buforowania, gdyż do tego celu w naszej architekturze wykorzystujemy pamięć RAM. Z tego powodu, rozmiar węzłów drzew B+ został ustalony na wielkość 4KB, czyli na rozmiar strony w RAM. Dzięki temu uzyskaliśmy mapowanie 1:1 z pamięcią PCM. Oczywiście pamięć ta jest bajtowo adresowalna. Oznacza to, że możemy w teorii odczytać dowolny pojedynczy bajt. W praktyce kontroler pamięci pozwala na odczyt dowolnej strony (64B). Zatem, mimo iż wielkość węzła w pamięci PCM jest równa 4KB, to dalej procesor będzie odczytywał potrzebne fragmenty o wielkości strony pamięci podręcznej 64B. Eksperymenty zostały przeprowadzone na dwóch charakterystycznych tabelach popularnego zestawu kwerend TPC-C ([137]): Sklep oraz Klient. Charakterystyka tabel została dokładnie opisana w rozdziale **4**.

Tworzenie pełnego indeksu

	AM	**eAM**	**PAM**
SERIA I (Sklep, 1%)	157s	72s	37s
SERIA II (Sklep, 5%)	156s	69s	37s
SERIA III (Klient, 1%)	905s	313s	223s
SERIA IV (Klient, 5%)	902s	309s	221s

Tabela 7.2: Całkowity czas wykonania zestawu kwerend

	AM	**eAM**	**PAM**
SERIA I (Sklep, 1%)	15,2GB	4,8GB	3GB
SERIA II (Sklep, 5%)	15,2GB	4,8GB	3GB
SERIA III (Klient, 1%)	68GB	34GB	22GB
SERIA IV (Klient, 5%)	68GB	34GB	22GB

Tabela 7.3: Zużycie pamięci (liczba nadpisanych i usuniętych gigabajtów)

Z rozdziałów **4** oraz **6** wiemy, że tworzenie pełnego indeksu to skomplikowany i często długi proces. Porównanie sumarycznych czasów nie jest wystarczające, aby zrozumieć

charakterystykę systemów oraz ich słabe i mocne strony. Z tego względu porównujemy również czas każdej kwerendy. Ta seria eksperymentów ma na celu analizę systemów Adaptive Merging (AM), extendend Adaptive Merging (eAM) oraz PCM Adaptive Merging (PAM) podczas procesu tworzenia indeksu. Każdy z tych systemów został dokładnie opisany w tym rozdziale. AM jest to oryginalne podejście do tworzenia częściowego indeksu na pamięciach blokowych. eAM jest to stworzona przez nas modyfikacja oryginalnego algorytmu AM dostosowana do pamięci PCM na tyle, na ile pozwalała na to architektura algorytmu AM. PAM jest to całkiem nowe podejście do tworzenia indeksu, w pełni dostosowane do modelu PCM, które dodatkowo potrafi współpracować z wieloma indeksami. Domyślnie jak i w eksperymentach w systemie PAM został użyty nowy indeks BB+.

Pierwsza seria eksperymentów polega na wykonaniu samych kwerend wyszukujących o ustalonej selektywności do momentu, gdy indeks nie zostanie w pełni utworzony. Dzięki temu będziemy mogli przeanalizować w pełni różnice i podobieństwa porównywanych systemów. Każdy eksperyment rozpoczyna swoje działanie na bazie danych zawierającej 10mln rekordów, które nie są w żaden sposób posortowane. Wykonujemy kolejne kwerendy z wyszukiwaniem losowym o podanej selektywności, w czasie których dodajemy odczytane rekordy do indeksu. Eksperyment kończy się gdy indeks będzie w pełni utworzony.

W [139] pokazano bardzo dobry sposób na prowadzenie obserwacji takiego procesu. Sposób obserwacji został dokładnie opisany w rozdziale **4**. Przypomnijmy jednak główne zasady. Autorzy wyznaczyli 4 fazy, które można wyróżnić w każdym algorytmie indeksowania.

1. **sadzenie (ang. planting)** - Pierwsza faza sadzenia polega na inicjalizacji całego procesu. Indeks zawiera na tyle mało elementów, że prawdopodobieństwo jego użycia jest bliskie 0. Dlatego musimy wykonać drogie skanowanie oraz dodać kilka operacji potrzebnych na powiększenie indeksu.

2. **pielęgnacja (ang. nursing)** - Druga faza zaczyna się, gdy czas jest mniejszy niż prosty skan, ale nadal większy niż w przypadku użycia pełnego indeksowania.

3. **wzrastanie (ang. growing)** - Trzecia faza zaczyna się, gdy natrafimy na zapytanie, które możemy obsłużyć w minimalnym czasie, czyli takim samym co pełny indeks.

4. **zbiory (ang. harvesting)** - Ostatnia faza zbiorów zaczyna się, gdy indeks jest w pełni stworzony, a zatem nie mamy już czasowych narzutów na jego tworzenie. Czas na obsługę zapytania jest taki sam jak przy pełnym indeksie.

Aby zbadać wszystkie zmienne, które mogą wpłynąć na charakterystykę wyników, przeprowadziliśmy trzy serie testów. Każda z nich wykonywała wyszukiwania o wzorcu losowym, który najlepiej oddaje naturę hurtowni danych. Taki serwer ma tak wielu użytkowników, że mimo iż każdy z nich tworzy zapytania według jakiegoś wzorca, to łącząc wszystkie kwerendy w ciąg, zapytania wyglądają jak przypadkowe (losowe).

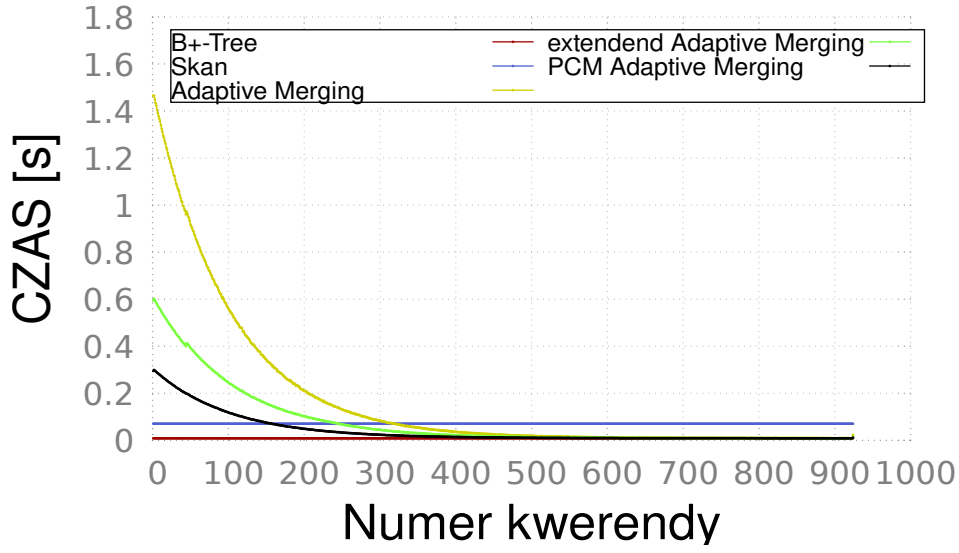
1. SERIA I - Tworzymy pełny indeks na tabeli Sklep, używamy do tego wyszukiwań z zakresu kluczy o selektywności 1%.

2. SERIA II - Tworzymy pełny indeks na tabeli Sklep, używamy do tego wyszukiwań z zakresu kluczy o selektywności 5%.

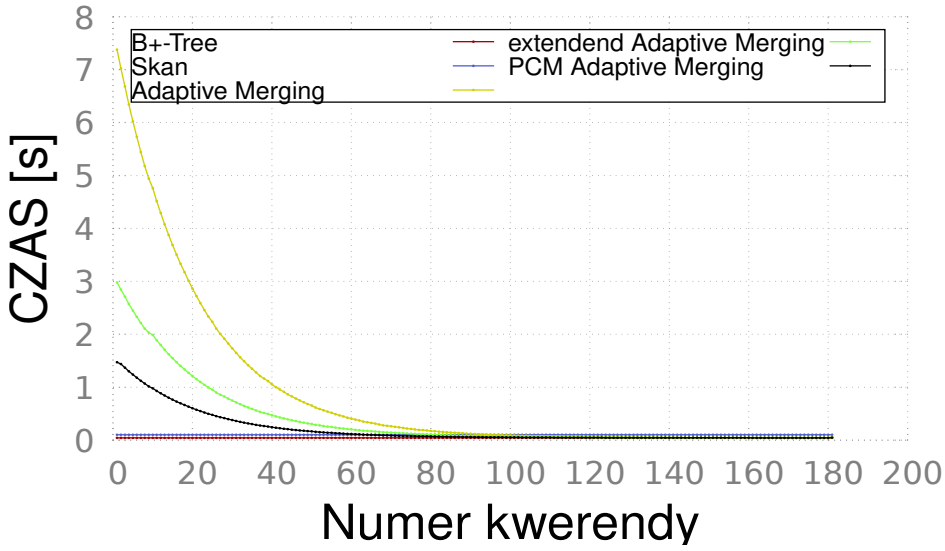
3. SERIA III - Tworzymy pełny indeks na tabeli Klient, używamy do tego wyszukiwań z zakresu kluczy o selektywności 1%.

4. SERIA IV - Tworzymy pełny indeks na tabeli Klient, używamy do tego wyszukiwań z zakresu kluczy o selektywności 5%.

Tabela 7.2 zawiera dokładny sumaryczny czas jaki uzyskały algorytmy częściowego indeksowania na testowym modelu PCM. Dodatkowo tabela 7.3 zawiera całkowite zużycie pamięci PCM, czyli liczbę gigabajtów nadpisaną i usuniętą podczas procesu tworzenia pełnego indeksu. Widzimy, że selektywność nie wpływa znacząco na czas jak i zużycie pamięci. Sytuacja ma miejsce w przypadku użycia każdego z porównywanych systemów. Dzieje się tak, ponieważ pamięć PCM nie jest blokowa tak jak dysk SSD. Oznacza to, że jednoczesny odczyt ogromnych zbiorów danych jest porównywalnie szybki tak jak kilkaset mniejszych odczytów. Analogiczna sytuacja jest podczas zapisywania. Dlatego zużycie pamięci jest dokładnie takie same w przypadku obu selektywności (1% i 5%). Oczywistym faktem jest, iż tabela o większym rekordzie zajmuje więcej miejsca na dysku. Zatem praca na tabeli Klient zajmuje więcej czasu oraz zużywa więcej zasobów niż praca na tabeli Sklep. Można zaobserwować, że czas jak i zużycie pamięci rośnie liniowo wraz z rosnącym rozmiarem rekordu. Warto nadmienić, że system eAM uzyskał dwukrotnie lepszy czas od oryginalnego systemu AM podczas pracy na tabeli Sklep oraz trzy razy lepszy czas podczas pracy na tabeli Klient. Dodatkowo nowy system PAM uzyskał średnio 4 razy lepszy czas przy wykorzystaniu obu tabel. Znacząco też poprawiło się zużycie pamięci. Algorytm PAM wykonał 5 razy mniej modyfikacji w przypadku tabeli Sklep oraz 3 razy mniej w przypadku tabeli Klient. W każdej serii czas oraz zużycie pamięci przez algorytm eAM było zawsze lepsze od oryginalnego AM, a wyniki uzyskane przez system PAM zawsze było najlepsze w przeprowadzonych testach.

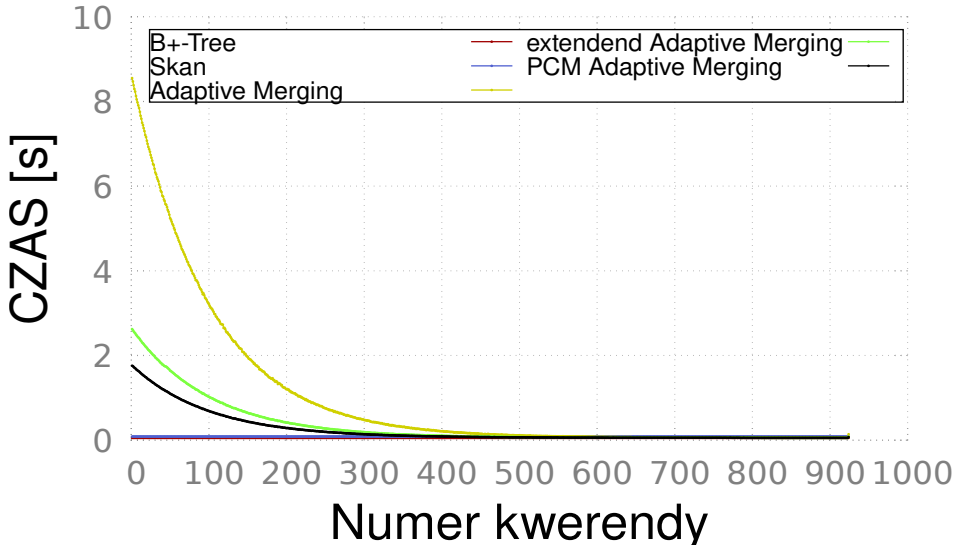


Rysunek 7.4: Czas tworzenia pełnego indeksu
Selektywność 1%
Tabela: Sklep (113B)

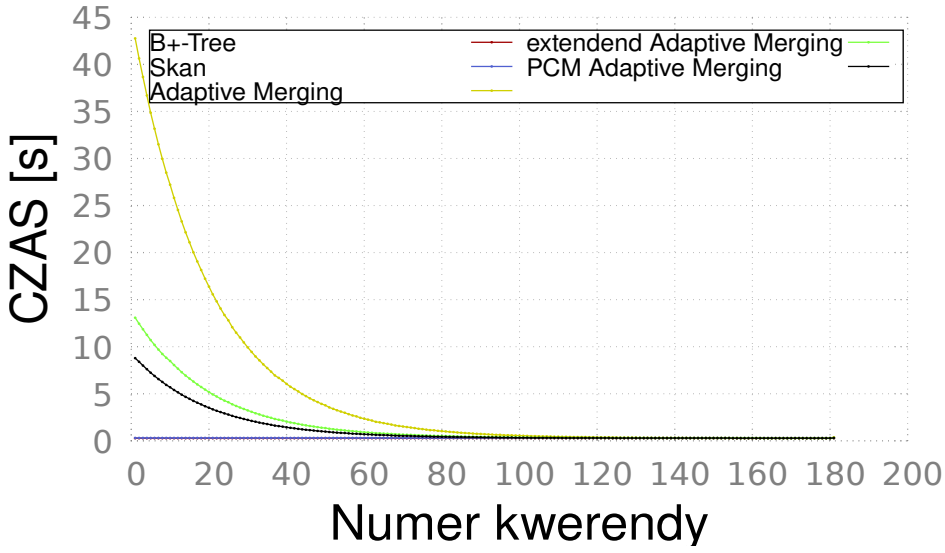


Rysunek 7.5: Czas tworzenia pełnego indeksu
Selektywność 5%
Tabela: Sklep (113B)

Przeanalizujmy teraz wyniki serii I dla tabeli Sklep. Przy badaniu każdej kwerendy użyliśmy również pełnego indeksu B+ oraz pełnego skanowania tabeli jako punktów odniesienia. Rysunki 7.4, 7.5 przedstawiają czas uzyskany przez wszystkie użyte algorytmy dla każdej z kwerend dla odpowiednio wyszukiwań z selektywnością na poziomie 1% oraz 5%. Na wstępie możemy zauważyć, że mimo iż selektywność nie wpływała na całkowity czas wykonania zestawu kwerend, to ma ona wpływ na liczbę kwerend potrzebną na zbudowanie pełnego indeksu. Jest to związane z tym, iż przy większej selektywności, odczytujemy więcej danych jednocześnie, a zatem więcej kopiujemy do indeksu. Dlatego proces tworzenia pełnego indeksu przy wykorzystaniu kwerend wyszukujący o selektywności 1% trwał 920 kwerend, a przy 5% tylko 178. Selektywność nie miała wpływu na trend widoczny na obu wykresach. Zawsze system PAM osiągał najlepszy czas obsługi kwerendy, a oryginalny AM najgorszy. Dodatkowo, algorytm PAM najszybciej osiągnął kolejne fazy budowania częściowego indeksu. Przy selektywności 1% fazę pielęgnacji osiągnął już w 134 kwerendzie (AM dopiero w 328), a fazę wzrastania już w 402 kwerendzie (AM dopiero w 605). Analizując wykresy 7.6 i 7.7 widzimy, że tak samo jak selektywność, również rozmiar rekordu nie wpływa na charakterystykę trendu. Ponownie system PAM był najszybszy, a oryginalny AM najwolniejszy. Warto jednak zauważyć, że rozmiar rekordu tabeli wpływa na czas uzyskania kolejnych faz przez wszystkie algorytmy. Poprzednio przy pracy na tabeli Sklep, system PAM osiągnął fazę II przy 134 i 65 kwerendzie odpowiednio dla serii I (selektywność 1%) i II (selektywność 2%). Tym razem dla tych samych selektywności algorytm PAM osiągnął fazę pielęgnacji w 298 i 72 kwerendzie. Oznacza to, iż im większy rekord, tym dłużej należy czekać, aż systemy tworzenia częściowego indeksu będą w stanie obsłużyć kwerendę szybciej od procedury pełnego skanowania tabeli. Wynika to z dużej asymetrii pomiędzy czasem operacji zapisu i odczytu. Różnica w referencyjnym modelu PCM jest tak duża, że o wiele szybciej jest liniowo odczytać całą bazę danych, niż dodawać nowe rekordy do indeksu. Oczywiście, taka sytuacja kończy się wraz z końcem fazy sadzenia, kiedy to mamy tyle danych w indeksie, że system jest szybszy niż proste skanowanie bazy.



Rysunek 7.6: Czas tworzenia pełnego indeksu
Selektywność 1%
Tabela: Klient (719B)



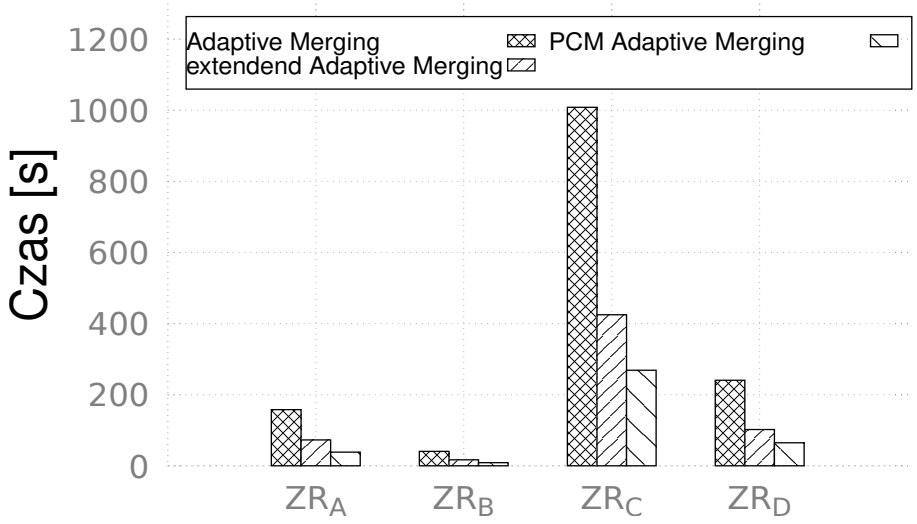
Rysunek 7.7: Czas tworzenia pełnego indeksu
Selektywność 5%
Tabela: Klient (719B)

Rozszerzony zestaw kwerend

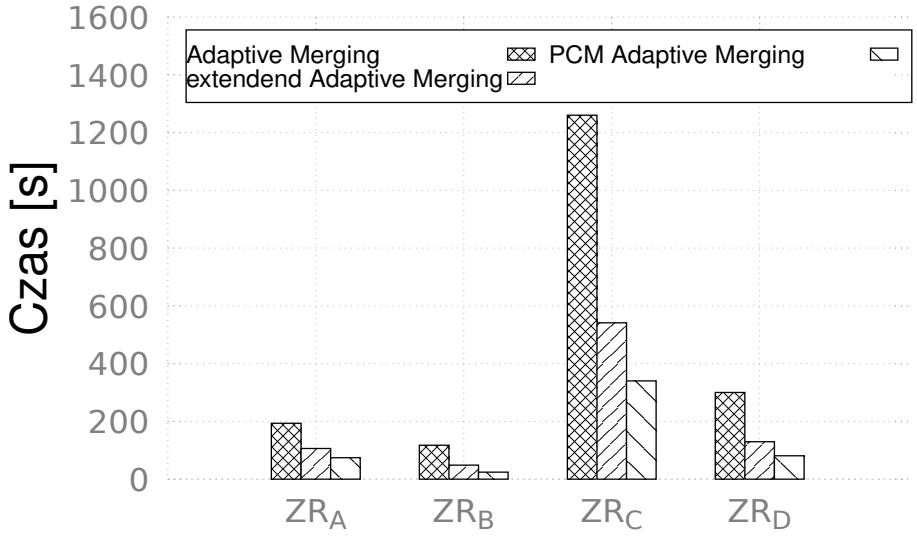
W poprzedniej sekcji zbadaliśmy sam proces tworzenia częściowego indeksu. Zazwyczaj do bazy trafiają różne kwerendy, a nie tylko zapytania wyszukujące rekordy po kluczu. Dlatego w tej części rozdziału przeanalizujemy wyniki otrzymane podczas wykonywania kwerend z zestawu rozszerzonego. Charakterystyka wszystkich 4 zestawów opisana w rozdziale 4 jest następująca:

1. ZR_A - 100 serii, każda seria zawiera operacje dodawania 5 rekordów, 10 wyszukiwań o podanej selektywności oraz usuwanie 5 rekordów,

2. ZR_B - 5 serii, każda seria zawiera operacje dodawania 100000 rekordów, 5 wyszukiwań o podanej selektywności oraz usuwanie 100000 rekordów,
3. ZR_C - 10 serii, każda seria zawiera operacje dodawania 10000000 rekordów, 20 wyszukiwań o podanej selektywności oraz usuwanie 1000000 rekordów,
4. ZR_D - 10 serii, każda seria zawiera operacje dodawania 1000000 rekordów, 10 wyszukiwań o podanej selektywności oraz usuwanie 10000 rekordów.



Rysunek 7.8: Czas zestawów kwerend
Selektywność 1%
Tabela: Sklep (113B)

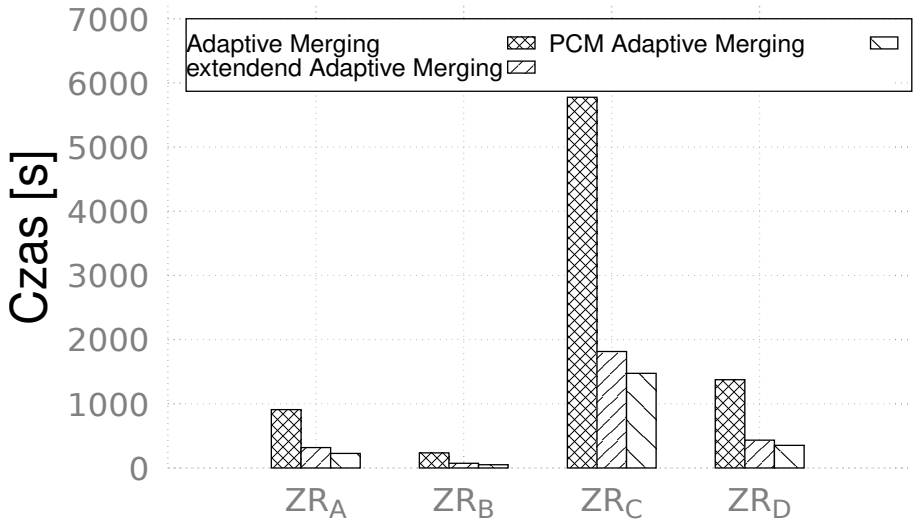


Rysunek 7.9: Czas zestawów kwerend
Selektywność 5%
Tabela: Sklep (113B)

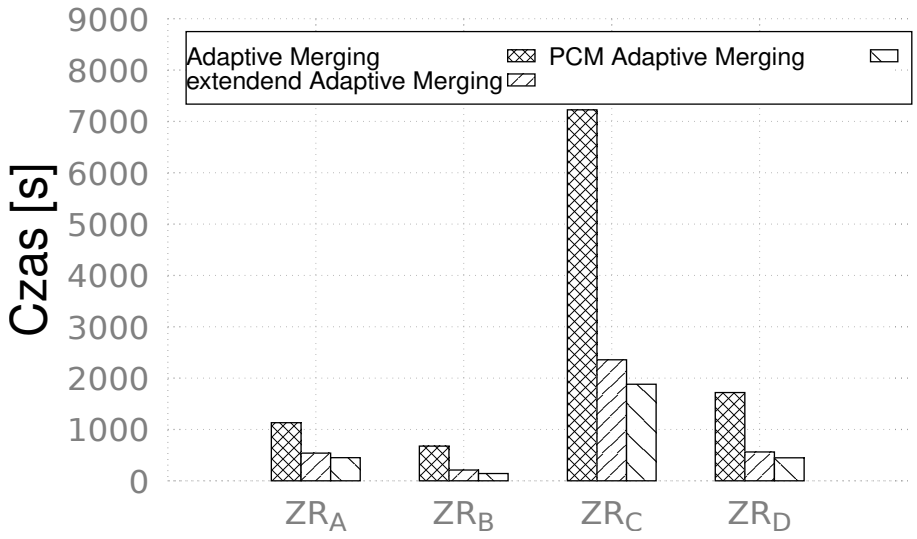
Przeanalizujmy zatem wyniki eksperymentów przeprowadzonych z użyciem zestawu

rozszerzonego. Wykresy 7.8 i 7.9 przedstawiają wyniki testów przeprowadzonych na tabeli Sklep z kwerendami wyszukującymi o selektywności odpowiednio 1% i 5%. Widzimy, że selektywność nie ma dużego wpływu na rezultaty. Pamięć PCM charakteryzuje się bardzo szybkim odczytem i o wiele wolniejszym zapisem. Zatem w zestawie o dominującej operacji dodawania lub usuwania rekordu, czas potrzebny na wyszukiwanie jest znikomy w porównaniu do czasu modyfikacji bazy. Również oczywistym faktem jest to, że zestaw ZR_C, który zawiera najwięcej operacji dodawania i usuwania, wykonywał się najwolniej ze wszystkich zestawów. Analogicznie zestawy ZR_A i ZR_B, które skupiają się głównie na wyszukiwaniach wykonywały się najszybciej. W każdym zestawie, niezależnie od selektywności, najwolniejszy był system Adaptive Merging, a najszybszy PCM Adaptive Merging. W zestawach o dominujących odczytach PAM był 4 krotnie szybszy od AM, a zestawie ZR_C nawet 6 krotnie szybszy od oryginalnego algorytmu.

Rysunki 7.10, 7.11 przedstawiają wyniki tego samego zestawu kwerend, przeprowadzonego na tabeli Klient, również z selektywnościami ustawionymi na odpowiednio 1% i 5%. Analizując wyniki testów możemy dojść do tych samych wniosków, co przy testach na tabeli Sklep. Ponownie selektywność nie ma dużego wpływu na czas obsługi całego zestawu. Kolejny raz system AM był najwolniejszy, a system PAM najszybszy. Tym razem w zestawach ZR_A i ZR_B PAM uzyskał 3 krotną przewagę (wcześniej była 4 krotna), a w zestawach ZR_C i ZR_D algorytm PAM był 4 krotnie szybszy (wcześniej nawet 6 krotnie) od oryginalnego algorytmu AM. Możemy również zauważyć, że czas obsługi każdego z zestawów rośnie bardzo podobnie wraz ze wzrostem rozmiaru rekordu. Przykładowo, dla rekordu z tabeli Sklep system PAM potrzebował 1050s aby wykonać zestaw ZR_C przy ustalonej selektywności na 1%. Ten sam zestaw algorytm wykonywał 5.62 razy dłużej dla rekordu z tabeli Klient. Rekord z tej tabeli jest 6.3 razy większy niż rekord tabeli Sklep.



Rysunek 7.10: Czas zestawów kwerend
Selektywność 1%
Tabela: Klient (719B)



Rysunek 7.11: Czas zestawów kwerend
Selektywność 5%
Tabela: Klient (719B)

Podsumowując, system PAM radzi sobie świetnie nie tylko w przypadku samego wyszukiwania, ale także wtedy, gdy baza danych jest modyfikowana w momencie kiedy indeks nie jest jeszcze w pełni stworzony. Wprowadzenie dzienników, które minimalizują liczbę modyfikacji pamięci, bardzo mocno przyspieszyło operacje oznaczania danych jako usunięte. Dodatkowo indeks BB+ jest w pełni przystosowany do kwerend, które zazwyczaj obsługują systemy częściowego indeksowania. Oba główne usprawnienia powodują, że system PAM nie tylko osiąga nawet 4 krotnie lepsze wyniki od oryginalnego algorytmu Adaptive Merging, ale także cechuje się 2 większą wydajnością od dostosowanej do PCM wersji eAM.

Zakończenie

8.1 Podsumowanie

Praca poświęcona jest indeksowaniu baz danych na różnych nowoczesnych typach pamięci, takich jak pojedyncze kości flash, dyski SSD oraz pamięć zmiennofazowa PCM. W dzisiejszych czasach indeks to nie tylko prosta struktura danych wyposażona w podstawowe procedury, to często duży, rozbudowany system, który uwzględnia wiele czynników m.in. charakterystykę pamięci. Dotyczy to w szczególności ograniczeń liczby możliwych usunięć danych z komórek pamięci, jak również asymetrii pomiędzy kosztami zapisu i odczytu. Efektywny sposób indeksowania rekordów nie tylko powinien minimalizować czas potrzebny na wykonanie kwerendy, ale także powinien wydłużać czas życia nowych nośników pamięci. Niestety szybkie pamięci jak dyski SSD oraz PCM posiadają ograniczenia wynikające z fizycznych limitów surowca użytego do budowy dysku. W hurtowniach danych codziennie zapisuje się terabajty informacji. W tym kontekście, minimalizacja zużycia pamięci jest bardzo istotna nie tylko z punktu widzenia ekonomicznego, ale także z ekologicznego. Algorytm indeksowania musi być bezpieczny. Oznacza to, że wszystkie dane, które są w nim zapisane muszą zostać nienaruszone podczas awarii systemu. Mimo to, algorytmy indeksujące bardzo często używają różnych technik buforowania, aby przyspieszyć wykonywanie zapytań. Informacje te zapisywane są w ulotnej pamięci RAM, a w chwili awarii systemu tracone. Zatem bardzo ważnym aspektem takich algorytmów jest zdolność przywrócenia stanu przed awarią. Z reguły jest to realizowane poprzez odtworzenie wszelkich utraconych danych na podstawie innych informacji zapisanych na dysku.

Pierwszym głównym celem pracy jest analiza popularnych algorytmów i systemów wykorzystywanych w silnikach baz danych do indeksowania rekordów. Drugim celem jest opracowanie nowych metod indeksowania oraz implementacja zaprojektowanych struktur danych. W pracy zaproponowano kilka nowych algorytmów i systemów porządkowania danych, realizujących indeksowanie rekordów. Każdy z algorytmów przeanalizowano, dołączając analizę teoretyczną złożoności obliczeniowej oraz dowód poprawności. W rozdziale **5** omówiono algorytmy indeksujące rekordy na pamięci flash. Przedstawiono także nowy algorytm indeksowania FA-Tree (ang. Flash Aware Tree) [1]. Zaproponowana struktura danych FA-Tree osiągała nawet 20 krotnie lepszy czas obsługi kwerend zorientowanych na modyfikacje bazy od klasycznego indeksu opartego na drzewie B+. Jednocześnie indeks FA-Tree zużył 6 krotnie mniej pamięci od drzewa B+.

W rozdziale **6** omówiono sposoby indeksowania danych zapisanych na dyskach SSD. Przedstawiono nowe algorytmy indeksowania danych takie jak wierszowy indeks FALSM-Tree (ang. Flash Aware LSM-Tree) [2], strukturę w pełni dostosowaną do dysków SSD jak i wspierającą operację dodawania zbiorczego rekordów (ang. bulkload). Dzięki wprowadzeniu nowej metody dodawania rekordów, czas obsługi takiej kwerendy został zredukowany aż 6 krotnie względem oryginalnego indeksu LSM. Dodatkowo warto również

zauważyć, że eksperymenty przeprowadzone na zestawach TPC-C wykazały, że nowy algorytm nadpisywał nawet 6 krotnie mniej pamięci. Zatem znacząco wydłużył czas życia dysków SSD. W tym samym rozdziale omówiono także całkiem nowy kolumnowy indeks CFT (ang. Columned FD-Tree) [3]. Indeks ten jest zbiorem struktur danych typu FD-Tree połączonych ze sobą specjalnie zaprojektowanymi metadanymi. Mimo kolumnowego ułożenia, indeks CFT osiągał czas modyfikacji struktury mocno zbliżony do czasu wykonania modyfikacji przez oryginalne wierszowe podejście z wykorzystaniem drzewa FD. Maksymalne odchylenie na niekorzyść kolumnowego indeksu wynosiło 6%. W tym samym rozdziale również przedstawiono nowy mechanizm indeksowania częściowego LAM (ang. Lazy Adaptive Merging) [4]. System ten jest w pełni przystosowany do charakterystyki dysków SSD. Nie tylko osiągał 2 krotnie lepsze wyniki od algorytmu AM podczas tworzenia samego indeksu, ale także wykonywał kwerendy modyfikujące tabelę średnio o 15% szybciej.

W rozdziale **7** omówiono indeksowanie baz danych używających pamięć zmiennofazową PCM do przechowywania danych. Przedstawiono także nową strukturę danych BB+-Tree (ang. Buffered B+-Tree) oraz nowy sposób częściowego indeksowania dostosowanego do pamięci PCM - PAM (ang. PCM Adaptive Merging) [5]. Wyniki pokazują ogromną przewagę nowego przedstawionego systemu nad oryginalnym algorytmem Adaptive Merging. Algorytm PAM stworzył indeks nawet 5 krotnie szybciej, modyfikując także 5 krotnie mniej pamięci, co znacząco wydłużyło żywotność tego nośnika pamięci.

8.2 Dalsze kierunki badań

Technologia rozwija się w bardzo szybkim tempie. Jeszcze 20 lat temu, dyski HDD o pojemności 10GB wystarczały na zapisanie wszystkich potrzebnych plików i danych. 10 lat temu dyski o pojemności 256GB były powszechnie używane i uznawane za dostatecznie pojemne. Obecnie każdy w komputerze osobistym posiada dyski o pojemności 1-4TB, co i tak nie wystarcza, aby pomieścić wszystkie potrzebne pliki,takie jak zdjęcia i filmy o wysokiej rozdzielczości. Prawo Moore'a ([218]) dokładnie opisuje to zjawisko. Wedle tej teorii, liczba danych potrzebnych ludzkości rośnie wykładniczo z roku na rok, a co za tym idzie liczba tranzystorów i komórek pamięci również musi rosnąć wykładniczo aby zaspokoić potrzeby rynkowe [219].

Szybko rosnąca potrzeba na przechowywanie co raz to większej ilości danych, powoduje nie tylko zwiększanie pojemności dysków w hurtowniach danych, np. poprzez użycie pamięci NAND QLC [220], która pozwala na zapis większej ilości danych przy podobnym koszcie modelu pamięci, ale także wymusza na dostawcach oprogramowania ulepszanie algorytmów procesujących te dane. Jednym z podstawowych technik optymalizacji jest zrównoleglenie niektórych operacji: zbiorczego dodawania (ang. bulkload) [221], odczytywania tabeli (ang. scan) [222], czy łączenia danych z tabeli (ang. join) [223]. Zazwyczaj silniki baz danych, które wykorzystują pełny potencjał procesora, poprzez użycie wielu wątków osiągają o wiele osiągi od silników jednowątkowych [224], [225].

Potrzeba polepszenia wydajności w procesie przetwarzania danych, spowodowała adopcję nowych komponentów sprzętowych w hurtowniach danych. Wprowadzone nowe modele dysków SSD z interfejsem PCIe 4.0 osiągają szybkość odczytu na poziomie aż 7GB/s, modele z interfejsem PCIe 5.0 posiadają prędkość odczytu sięgającą nawet do 10GB/s. Najnowsze modele dysków SSD borykają się z problemem nadmiernego nagrzewania się,

przy ciągłym użyciu dysku na maksymalnym poziomie. Gdy taka sytuacja ma miejsce, kontroler obniża wydajność dysku aby uchronić go przed nadmiernym przegrzaniem (ang. termal throttling) [226]. Aby tego uniknąć silnik baz danych powinien odpowiednio rozdzielać zadania tak, aby przetwarzać dane podzielone na zbiory w taki sposób, żeby dysk miał czas na schłodzenie, a procesor w tym czasie wykonał swoje zadanie. Dzięki temu kontroler dysku dostosuje pobieraną moc przez model dysku co skutkuje mniejszymi temperaturami, przekładającymi się na większą sumaryczną wydajność. [227].

Kolejnym ciekawym nurtem jest użycie kart graficznych (ang. GPU) do przetwarzania niektórych danych. Karta graficzna posiada kompletnie inną architekturę od zwykłych procesorów: o wiele wolniej wykonuje operacje decyzyjne, jednocześnie posiada bardzo wysoką wydajność operacji arytmetycznych jak i prostej obróbki danych [228], [229]. Z tego względu w ciągu kilku minionych lat zostało zaproponowanych wiele algorytmów wykorzystujących GPU w hurtowniach danych [230], [231], [232], [233].

Każde z wcześniej wymienionych problemów i trendów wymaga wprowadzania co raz to nowych algorytmów i usprawnień, aby nadążyć za problemami rynku. Uważam, że wymienione przeze mnie problemy mogą zostać głębiej przestudiowane w ramach rozszerzenia tej pracy.

Bibliografia

[1] W. Macyna and M. Kukowski, "Flash-aware clustered index for mobile databases," in *2018 International Conference on Industrial Enterprise and System Engineering (ICoIESE 2018)*, pp. 25–30, Atlantis Press, 2019.

[2] W. Macyna and M. Kukowski, "Bulk loading of the secondary index in lsm-based stores for flash memory," in *European Conference on Advances in Databases and Information Systems*, pp. 133–143, Springer, 2022.

[3] W. Macyna and M. Kukowski, "Flash-aware storage of the column oriented databases," *Fundamenta Informaticae*, vol. 173, no. 1, pp. 47–72, 2020.

[4] W. Macyna and M. Kukowski, "Partially indexing on flash memory," in *International Conference on Database and Expert Systems Applications*, pp. 95–105, Springer, 2019.

[5] M. Kukowski and W. Macyna, "Adaptive merging on phase change memory," *Fundamenta Informaticae*, vol. 188, 2023.

[6] A. Bhattacharya, *Fundamentals of database indexing and searching*. CRC Press, 2014.

[7] E. Bertino, B. C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and D. Andronico, *Indexing techniques for advanced database systems*, vol. 8. Springer Science & Business Media, 2012.

[8] Y. Manolopoulos, Y. Theodoridis, and V. Tsotras, *Advanced database indexing*, vol. 17. Springer Science & Business Media, 2012.

[9] S. S. Lightstone, T. J. Teorey, and T. Nadeau, *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann, 2010.

[10] V. Bhagat and A. Gopal, "Comparative study of row and column oriented database," in *2012 Fifth International Conference on Emerging Trends in Engineering and Technology*, pp. 196–201, IEEE, 2012.

[11] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 967–980, 2008.

[12] A. S. Kanade and A. Gopal, "Choosing right database system: Row or column-store," in *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pp. 16–20, IEEE, 2013.

[13] G. Graefe and H. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, (New York, NY, USA), pp. 371–381, ACM, 2010.

[14] C. J. Date, *An introduction to database systems*. Pearson Education India, 1975.

[15] H. Garcia-Molina, *Database systems: the complete book*. Pearson Education India, 2008.

[16] M. Athanassoulis and S. Idreos, "Design tradeoffs of data access methods," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 2195–2200, 2016.

[17] S. P. Ghosh and M. E. Senko, "File organization: on the selection of random access index points for sequential files," *Journal of the ACM (JACM)*, vol. 16, no. 4, pp. 569–579, 1969.

[18] R. J. Srodawa and L. A. Bates, "An efficient virtual machine implementation," in *Proceedings of the workshop on virtual computer systems*, pp. 43–73, 1973.

[19] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, pp. 173–189, Sep 1972.

[20] B. Bloom, "Space/time trade-offs, in, hash, coding, with," *Communications, of, A CM„ pages*, 1970.

[21] L. Qiaoyu, L. Jianwei, and X. Yubin, "Performance analysis of data organization of the real-time memory database based on red-black tree," in *2010 International Conference on Computing, Control and Industrial Engineering*, vol. 1, pp. 428–430, IEEE, 2010.

[22] C. C. Foster, "A generalization of avl trees," *Communications of the ACM*, vol. 16, no. 8, pp. 513–517, 1973.

[23] J.-I. Aoe, K. Morimoto, and T. Sato, "An efficient implementation of trie structures," *Software: Practice and Experience*, vol. 22, no. 9, pp. 695–721, 1992.

[24] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[25] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1195–1206, 2010.

[26] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, "Lazy-adaptive tree: An optimized index structure for flash devices," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 361–372, 2009.

[27] P. Chi, W.-C. Lee, and Y. Xie, "Making b+-tree efficient in pcm-based main memory," in *Proceedings of the 2014 international symposium on Low power electronics and design*, pp. 69–74, 2014.

[28] L. Li, P. Jin, C. Yang, S. Wan, and L. Yue, "Xb+-tree: A novel index for pcm/dram-based hybrid memory," in *Australasian Database Conference*, pp. 357–368, Springer, 2016.

[29] L. Li, P. Jin, C. Yang, Z. Wu, and L. Yue, "Optimizing b+-tree for pcm-based hybrid memory.," in *EDBT*, pp. 662–663, 2016.

[30] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," *Acm Sigmod Record*, vol. 14, no. 4, pp. 268–279, 1985.

[31] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a column-oriented dbms," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pp. 491–518, 2018.

[32] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, *et al.*, "The design and implementation of modern column-oriented database systems," *Foundations and Trends® in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[33] F. Yu, T. J. Matacic, B. J. Latronica, and W.-C. Hou, "Ob-tree: a new write optimisation index on out-of-core column-store databases," *International Journal of Intelligent Information and Database Systems*, vol. 11, no. 1, pp. 46–66, 2018.

[34] M. Boissier, T. Djürken, R. Schlosser, and M. Faust, "A cost-aware and workload-based index advisor for columnar in-memory databases," in *International Conference on Information and Software Technologies*, pp. 285–299, Springer, 2016.

[35] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query processing techniques for solid state drives," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 59–72, 2009.

[36] E. V. Ivanova and L. B. Sokolinsky, "Parallel processing of very large databases using distributed column indexes," *Programming and Computer Software*, vol. 43, no. 3, pp. 131–144, 2017.

[37] A. Floratou, J. Patel, E. Shekita, and S. Tata, "Column-oriented storage techniques for mapreduce," *arXiv preprint arXiv:1105.4252*, 2011.

[38] K.-C. Kim and C.-S. Kim, "Parallel processing of sensor network data using column-oriented databases," *AASRI Procedia*, vol. 5, pp. 2–8, 2013.

[39] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 671–682, 2006.

[40] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 283–296, 2009.

[41] T. Waage, *A Framework for Property-preserving Encryption in Wide Column Store Databases*. PhD thesis, Dissertation, Göttingen, Georg-August Universität, 2017, 2017.

[42] J. Hildebrandt, D. Habich, and W. Lehner, "Model-driven integration of compression algorithms in column-store database systems.," in *LWDA*, pp. 30–41, Citeseer, 2016.

[43] M. Boissier and M. Jendruk, "Workload-driven and robust selection of compression schemes for column stores.," in *EDBT*, pp. 674–677, 2019.

[44] K. Sridhar and J. Johnson, "Entropy aware adaptive compression for sql column stores," in *International Conference: Beyond Databases, Architectures and Structures*, pp. 90–104, Springer, 2018.

[45] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution.," in *Cidr*, vol. 5, pp. 225–237, Citeseer, 2005.

[46] M. Stonebraker, L. Rowe, and M. Hirohama, "The implementation of postgres," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 2, pp. 125 – 142, 04 1990.

[47] M. Zukowski and P. A. Boncz, "Vectorwise: Beyond column stores," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 21–27, 2012.

[48] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 157–168, 2014.

[49] T. T. F. S. B. SPECIFICATIONS, "Stac-m3™ benchmarks," 2013.

[50] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz, "Positional update handling in column stores," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 543–554, 2010.

[51] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe, "Fast scans and joins using flash drives," in *Proceedings of the 4th international workshop on Data management on new hardware*, pp. 17–24, 2008.

[52] Q. Cao, Z. Liang, Y. Fan, and X. Meng, "A flash-based decomposition storage model," in *International Conference on Database Systems for Advanced Applications*, pp. 73–80, Springer, 2012.

[53] M. Zaman, J. Surabattula, and L. Gruenwald, "An auto-indexing technique for databases based on clustering," in *Proceedings. 15th International Workshop on Database and Expert Systems Applications, 2004.*, pp. 776–780, IEEE, 2004.

[54] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri, "Automatically indexing millions of databases in microsoft azure sql database," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 666–679, 2019.

[55] W. G. Pedrozo and M. S. M. G. Vaz, "A tool for automatic index selection in database management systems," in *2014 International Symposium on Computer, Consumer and Control*, pp. 1061–1064, IEEE, 2014.

[56] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007.

[57] M. L. Kersten, S. Manegold, M. Kersten, and S. Manegold, "Cracking the database store," in *In CIDR*, 2005.

[58] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, pp. 321–, July 1961.

[59] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, (New York, NY, USA), pp. 297–308, ACM, 2009.

[60] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 5, pp. 502–513, Feb. 2012.

[61] S. Idreos, M. L. Kersten, and S. Manegold, "Updating a cracked database," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, (New York, NY, USA), pp. 413–424, ACM, 2007.

[62] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[63] G. Graefe, "Sorting and indexing with partitioned b-trees," in *CIDR*, 2003.

[64] J. Lee, H. Roh, and S. Park, "External mergesort for flash-based solid state drives," *IEEE Transactions on Computers*, vol. 65, pp. 1518–1527, May 2016.

[65] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter, "Main memory adaptive indexing for multi-core systems," in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, (New York, NY, USA), pp. 3:1–3:10, ACM, 2014.

[66] S. Idreos, S. Manegold, H. Kuno, and G. Graefe, "Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 4, pp. 586–597, June 2011.

[67] Z. Xue, X. Qin, X. Zhou, S. Wang, and A. Yu, "Optimized adaptive hybrid indexing for in-memory column stores," in *Proceedings of the 18th International Conference on Database Systems for Advanced Applications - Volume 7827*, (New York, NY, USA), pp. 101–111, Springer-Verlag New York, Inc., 2013.

[68] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, 1979.

[69] Y. Li, B. He, Q. Luo, and K. Yi, "Tree indexing on flash disks," in *2009 IEEE 25th International Conference on Data Engineering*, pp. 1303–1306, IEEE, 2009.

[70] "Oficjalne źródła jądra systemu operacyjnego linux." https://github.com/torvalds/linux.

[71] "Oficjalna dokumentacja systemu operacyjnego linux." https://linux.die.net/man/.

[72] M. Snir and J. Yu, "On the theory of spatial and temporal locality," 01 2005.

[73] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, oct 2021.

[74] "Oficjalna dokumentacja systemu mysql." https://dev.mysql.com/doc/.

[75] P. Hong-Jin, "A study of the innodb storage engine in the mysql 5.6," 01 2017.

[76] S. Mueller, *Upgrading and Repairing PCs (17th Edition)*. USA: Que Corp., 2006.

[77] G. Graefe, "Write-optimized b-trees," pp. 672–683, VLDB Endowment, 2004.

[78] B.Buvaneswari, "A survey on multi gate mosfets," *International Journal of Innovative Research in Science, Engineering and Technology*, vol. 3, 2014.

[79] S. Electronics, *Nand Flash Memory & Smartmedia Data*. 2007.

[80] K. Naruke, S. Yamada, E. Obi, S. Taguchi, and M. Wada, "A new flash-erase eeprom cell with a sidewall select-gate on its source side," in *International Technical Digest on Electron Devices Meeting*, pp. 603–606, 1989.

[81] G. Samachisa, C.-S. Su, Y.-S. Kao, G. Smarandoiu, C.-Y. Wang, T. Wong, and C. Hu, "A 128 k flash eeprom using double-polysilicon technology," *IEEE Journal of Solid-State Circuits*, vol. 22, no. 5, pp. 676–683, 1987.

[82] A. Amin, "Design, selection and implementation of flash erase eeprom memory cells," *IEE Proceedings G (Circuits, Devices and Systems)*, vol. 139, no. 3, pp. 370–376, 1992.

[83] S. Skorobogatov, "Data remanence in flash memory devices," in *Cryptographic Hardware and Embedded Systems – CHES 2005* (J. R. Rao and B. Sunar, eds.), (Berlin, Heidelberg), pp. 339–353, Springer Berlin Heidelberg, 2005.

[84] J. Hu, H. Jiang, L. Tian, and L. Xu, "Pud-lru: An erase-efficient write buffer management algorithm for flash memory ssd," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 69–78, 2010.

[85] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, p. 138–163, jun 2005.

[86] Y. Deng and J. Zhou, "Architectures and optimization methods of flash memory based storage systems," *Journal of Systems Architecture*, vol. 57, no. 2, pp. 214–227, 2011.

[87] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 234–241, 2006.

[88] A. Ban, "Flash file system," United States Patent No. 5,404,485, 1995.

[89] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 332–343, 2009.

[90] B. I. Petro Estakhri, "Moving sequential sectors within a block of information in a flash memory mass storage architecture," United States Patent No. 5,930,815, 1999.

[91] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[92] S. J. Kwon and T.-S. Chung, "An efficient and advanced space-management technique for flash memory using reallocation blocks," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 2, pp. 631–638, 2008.

[93] B.-s. Kim and G.-y. Lee, "Method of driving remapping in flash memory and flash memory architecture suitable therefor," Apr. 30 2002. US Patent 6,381,176.

[94] T.-S. Chung and H.-S. Park, "Staff: A flash driver algorithm minimizing block erasures," *Journal of Systems Architecture*, vol. 53, no. 12, pp. 889–901, 2007.

[95] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system.," in *USENIX*, pp. 155–164, 1995.

[96] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif, "Abstract specification of the ubifs file system for flash memory," in *International Symposium on Formal Methods*, pp. 190–206, Springer, 2009.

[97] X. Cai and S. Shao, "An optimization algorithm for ubifs wear-leveling," in *2010 2nd International Workshop on Intelligent Systems and Applications*, pp. 1–4, IEEE, 2010.

[98] J. Cichon and W. Macyna, "Approximate counters for flash memory," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, vol. 1, pp. 185–189, IEEE, 2011.

[99] H. Prodinger, "Approximate counting with m counters: a detailed analysis," *Theoretical Computer Science*, vol. 439, pp. 58–68, 2012.

[100] F. Geier, "The differences between ssd and hdd technology regarding forensic investigations," 2015.

[101] V. Kasavajhala, "Solid state drive vs. hard disk drive price and performance study," *Proc. Dell Tech. White Paper*, pp. 8–9, 2011.

[102] L. Jeevitha, B. Umadevi, and M. Hemavathy, "Sata protocol implementation on fpga for write protection of hard disk drive/solid state device," in *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 614–617, IEEE, 2019.

[103] M. Knowles, "Survey of the storage evolution," in *2003 User Group Conference. Proceedings*, pp. 362–367, 2003.

[104] E. C. Lee, J. Rho, B. J. Lee, and H. Kang, "Heat dissipation analysis of m.2 nvme solid-state drive in vacuum," in *2019 International Vacuum Electronics Conference (IVEC)*, pp. 1–2, 2019.

[105] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssd," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[106] H. Riggs, S. Tufail, I. Parvez, and A. Sarwat, "Survey of solid state drives, characteristics, technology, and applications," in *2020 SoutheastCon*, pp. 1–6, IEEE, 2020.

[107] H. Cho, D. Shin, and Y. I. Eom, "Kast: K-associative sector translation for nand flash memory in real-time systems," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, pp. 507–512, IEEE, 2009.

[108] M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 203–216, 2013.

[109] Y. Du, J. Gu, Z. Xiao, and M. Huang, "Ssw: A strictly sequential writing method for open-channel ssd," *Journal of Systems Architecture*, vol. 109, p. 101828, 2020.

[110] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1075–1086, 2008.

[111] Y. Liu, J. Huang, C. Xie, and Q. Cao, "Raf: A random access first cache management to improve ssd-based disk cache," in *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, pp. 492–500, IEEE, 2010.

[112] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "Ssd bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[113] K. Smith, "Garbage collection," *SandForce, Flash Memory Summit, Santa Clara, CA*, pp. 1–9, 2011.

[114] P. Yang, N. Xue, Y. Zhang, Y. Zhou, L. Sun, W. Chen, Z. Chen, W. Xia, J. Li, and K. Kwon, "Reducing garbage collection overhead in {SSD} based on workload prediction," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[115] M. Jung, R. Prabhakar, and M. T. Kandemir, "Taking garbage collection overheads off the critical path in ssds," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 164–186, Springer, 2012.

[116] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds," *ACM Transactions on Storage (TOS)*, vol. 13, no. 3, pp. 1–26, 2017.

[117] N. Shahidi, M. T. Kandemir, M. Arjomand, C. R. Das, M. Jung, and A. Sivasubramaniam, "Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 561–572, IEEE, 2016.

[118] J. Guo, Y. Hu, B. Mao, and S. Wu, "Parallelism and garbage collection aware i/o scheduler with improved ssd performance," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1184–1193, IEEE, 2017.

[119] S. Ahn and S. Park, "An analytical approach to evaluation of ssd effects under mapreduce workloads," *JSTS: Journal of Semiconductor Technology and Science*, vol. 15, no. 5, pp. 511–518, 2015.

[120] G. Kim and D. Shin, "Performance analysis of ssd write using trim in ntfs and ext4," in *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pp. 422–423, IEEE, 2011.

[121] J. Kim, H. Kim, S. Lee, and Y. Won, "Ftl design for trim command," in *The Fifth International Workshop on Software Support for Portable Storage*, pp. 7–12, 2010.

[122] T. Frankie, G. Hughes, and K. Kreutz-Delgado, "Ssd trim commands considerably improve overprovisioning," *Flash Memory Summit*, 2011.

[123] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[124] G. W. Burr, M. J. Brightsky, A. Sebastian, H.-Y. Cheng, J.-Y. Wu, S. Kim, N. E. Sosa, N. Papandreou, H.-L. Lung, H. Pozidis, *et al.*, "Recent progress in phase-change memory technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 2, pp. 146–162, 2016.

[125] L. Bo, S. Zhi-Tang, Z. Ting, F. Song-Lin, and G. Fu-Xi, "Novel material for nonvolatile ovonic unified memory (oum)–ag11in12te26sb51 phase change semiconductor," *Chinese Physics*, vol. 13, no. 7, p. 1167, 2004.

[126] B. Beeler, "Intel optane dc persistent memory module (pmm)," *Retrieved March*, vol. 11, p. 2021, 2019.

[127] O. Zilberberg, S. Weiss, and S. Toledo, "Phase-change memory: An architectural perspective," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, pp. 1–33, 2013.

[128] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[129] G. Servalli, "A 45nm generation phase change memory technology," in *2009 IEEE International Electron Devices Meeting (IEDM)*, pp. 1–4, IEEE, 2009.

[130] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 133–146, 2009.

[131] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 2–13, 2009.

[132] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 24–33, 2009.

[133] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *ACM SIGARCH computer architecture news*, vol. 37, no. 3, pp. 14–23, 2009.

[134] S. Chen, P. B. Gibbons, S. Nath, *et al.*, "Rethinking database algorithms for phase change memory.," in *Cidr*, vol. 11, pp. 9–12, 2011.

[135] P. P. Kalua and E. L. Robertson, *Benchmark queries for temporal databases.* Citeseer, 1993.

[136] M. H. Dunham, R. Elmasri, M. A. Nascimento, and M. Sobol, *Benchmarking temporal databases: A research agenda.* Citeseer, 1995.

[137] S. T. Leutenegger and D. Dias, "A modeling study of the tpc-c benchmark," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, (New York, NY, USA), p. 22–31, Association for Computing Machinery, 1993.

[138] "Tpc-h – transaction performance council." http://www.tpc.org.

[139] G. Graefe, S. Idreos, H. Kuno, and S. Manegold, "Benchmarking adaptive indexing," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 169–184, Springer, 2010.

[140] H. Kim, E. H. Nam, K. S. Choi, Y. J. Seong, J.-y. Choi, and S. L. Min, "Development platforms for flash memory solid state disks," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 527–528, IEEE, 2008.

[141] "Projekt openssd." http://www.openssd-project.org.

[142] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, J. Kim, *et al.*, "Bluessd: An open platform for cross-layer experiments for nand flash-based ssds," in *WARP-5th Annual Workshop on Architectural Research Prototyping*, 2010.

[143] Y. Cai, E. F. Haratsch, M. McCartney, and K. Mai, "Fpga-based solid-state drive prototyping platform," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 101–104, IEEE, 2011.

[144] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, vol. 41, pp. 10–5555, Califor-nia, USA, 2005.

[145] "Projekt qemu." https://github.com/qemu/qemu.

[146] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pp. 83–90, 2018.

[147] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choi, S. Yoon, and J. Cha, "Vssim: Virtual machine based ssd simulator," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, IEEE, 2013.

[148] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "Simplessd: Modeling solid state drives for holistic system simulation," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 37–41, 2017.

[149] D. Lee, D. Hong, W. Choi, and J. Kim, "Mqsim-e: An enterprise ssd simulator," *IEEE Computer Architecture Letters*, 2022.

[150] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for {SSD} performance," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[151] G. Ganger, B. Worthington, and Y. Patt, "The disksim simulation environment (v4. 0)," *Parallel Data Lab, http://www. pdl. cmu. edu/DiskSim/Online-document*, 2009.

[152] V. Prabhakaran and T. Wobber, "Ssd extension for disksim simulation environment," *Microsoft Reseach*, 2009.

[153] J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh, "Cps-sim: configurable and accurate clock precision solid state drive simulator," in *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 318–325, 2009.

[154] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *2009 First International Conference on Advances in System Simulation*, pp. 125–131, IEEE, 2009.

[155] P. Jin, X. Su, Z. Li, and L. Yue, "A flexible simulation environment for flash-aware algorithms," in *Proceedings of the 18th ACM conference on information and knowledge management*, pp. 2093–2094, 2009.

[156] M. Jung, E. H. Wilson, D. Donofrio, J. Shalf, and M. T. Kandemir, "Nandflashsim: Intrinsic latency variation aware nand flash memory system modeling and simulation at microarchitecture level," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, IEEE, 2012.

[157] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The unwritten contract of solid state drives," in *Proceedings of the twelfth European conference on computer systems*, pp. 127–144, 2017.

[158] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the international conference on Supercomputing*, pp. 96–107, 2011.

[159] J. Yoo, Y. Won, S. Kang, J. Choi, S. Yoon, and J. Cha, "Analytical model of ssd parallelism," in *2014 4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications (SIMULTECH)*, pp. 551–559, IEEE, 2014.

[160] D. Zhang, P. Jin, X. Wang, C. Yang, and L. Yue, "Dphsim: A flexible simulator for dram/pcm-based hybrid memory," in *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*, pp. 319–323, Springer, 2017.

[161] J.-G. Kim, D.-Y. Kim, and Y.-H. Choe, "Pcm simulator," *The Magazine of the IEIE*, vol. 3, no. 2, pp. 46–55, 1976.

[162] X. Li, K. Lu, and X. Zhou, "Sim-pcm: a pcm simulator based on simics," in *2012 Fourth International Conference on Computational and Information Sciences*, pp. 1236–1239, IEEE, 2012.

[163] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–481, IEEE, 2018.

[164] Z. Shao, N. Chang, and N. Dutt, "Ptl: Pcm translation layer," in *2012 IEEE Computer Society Annual Symposium on VLSI*, pp. 380–385, IEEE, 2012.

[165] R. Micheloni and L. Crippa, *Solid-State-Drives (SSDs) Modeling*. Springer, 2017.

[166] J. Zhang, P. Li, B. Liu, T. G. Marbach, X. Liu, and G. Wang, "Performance analysis of 3d xpoint ssds in virtualized and non-virtualized environments," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 1–10, 2018.

[167] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[168] C. Delimitrou, S. Sankar, B. Khessib, K. Vaid, and C. Kozyrakis, "Time and cost-efficient modeling and generation of large-scale tpcc/tpce/tpch workloads," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 146–162, Springer, 2011.

[169] R. N. Avula and C. Zou, "Performance evaluation of tpc-c benchmark on various cloud providers," in *2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 0226–0233, IEEE, 2020.

[170] M. A. Kandaswamy and R. L. Knighten, "I/o phase characterization of tpc-h query operations," in *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, pp. 81–90, IEEE, 2000.

[171] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "The uncracked pieces in database cracking," *Proceedings of the VLDB Endowment*, vol. 7, no. 2, pp. 97–108, 2013.

[172] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[173] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An efficient b-tree layer implementation for flash-memory storage systems," vol. 6, p. 19–es, jul 2007.

[174] T. Gleixner, F. Haverkamp, and A. Bityutskiy, "Ubi-unsorted block images," *Rapport technique, adresse: http://linux-mtd. infradead. org/doc/ubidesign/ubidesign. pdf*, 2006.

[175] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 4, pp. 560–595, 1984.

[176] G. M. Sacco and M. Schkolnick, "Buffer management in relational database systems," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 4, pp. 473–498, 1986.

[177] Y. Ou, T. Härder, and P. Jin, "Cfdc: a flash-aware replacement policy for database buffer management," in *Proceedings of the fifth international workshop on data management on new hardware*, pp. 15–20, 2009.

[178] J. Bercken and B. Seeger, "An evaluation of generic bulk loading techniques," VLDB, 2001.

[179] G. Roumelis, A. Fevgas, M. Vassilakopoulos, A. Corral, P. Bozanis, and Y. Manolopoulos, "Bulk-loading and bulk-insertion algorithms for xBR$^+$-trees in Solid State Drives," *Computing*, vol. 101, no. 10, pp. 1539–1563, 2019.

[180] G. Roumelis, M. Vassilakopoulos, A. Corral, and Y. Manolopoulos, "An efficient algorithm for bulk-loading xbr+-trees," *Computer Standards & Interfaces*, vol. 57, pp. 83–100, 2018.

[181] Y. Zhu, Z. Zhang, P. Cai, W. Qian, and A. Zhou, "An efficient bulk loading approach of secondary index in distributed log-structured data stores," in *International Conference on Database Systems for Advanced Applications*, pp. 87–102, Springer, 2017.

[182] C. Luo and M. J. Carey, "Lsm-based storage techniques: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[183] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 505–520, 2018.

[184] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[185] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "Kvssd: Close integration of lsm trees and flash translation layer for write-efficient kv store," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 563–568, IEEE, 2018.

[186] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[187] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 209–223, 2020.

[188] H. Lee, M. Lee, and Y. I. Eom, "Sfm: Mitigating read/write amplification problem of lsm-tree-based key-value stores," *IEEE Access*, vol. 9, pp. 103153–103166, 2021.

[189] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "Complexity of fragmentable object bin packing and an application," *Computers & Mathematics with Applications*, vol. 35, no. 11, pp. 91–97, 1998.

[190] S. Martello and P. Toth, "Bin-packing problem," *Knapsack problems: Algorithms and computer implementations*, pp. 221–245, 1990.

[191] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pp. 14–23, IEEE, 2009.

[192] Y. Ou, L. Chen, J. Xu, and T. Härder, "Wear-aware algorithms for pcm-based database buffer pools," in *International Conference on Web-Age Information Management*, pp. 165–176, Springer, 2014.

[193] Z. Wu, P. Jin, and L. Yue, "Efficient space management and wear leveling for pcm-based storage systems," in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 784–798, Springer, 2015.

[194] S. D. Viglas, "Write-limited sorts and joins for persistent memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 5, pp. 413–424, 2014.

[195] X. Liu and W. Golab, "Brief announcement: Towards a theory of wear leveling in persistent data structures," in *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pp. 220–223, 2022.

[196] K. Zeng, Y. Lu, H. Wan, and J. Shu, "Efficient storage management for aged file systems on persistent memory," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1769–1774, IEEE, 2017.

[197] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *ACM SIGARCH computer architecture news*, vol. 37, no. 3, pp. 14–23, 2009.

[198] C. Pan, M. Xie, J. Hu, M. Qiu, and Q. Zhuge, "Wear-leveling for pcm main memory on embedded system via page management and process scheduling," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 1–9, IEEE, 2014.

[199] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha, "Curling-pcm: Application-specific wear leveling for phase change memory based embedded systems," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 279–284, IEEE, 2013.

[200] S. Rashidi, M. Jalili, and H. Sarbazi-Azad, "A survey on pcm lifetime enhancement schemes," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–38, 2019.

[201] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious b+-trees," in *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 283–294, 2003.

[202] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 235–246, 2001.

[203] V. Dubeyko, "Method of resolving wandering tree problem of log-structured file systems by means of segment-based extent approach,"

[204] A. B. Bityuckiy, "Jffs3 design issues," *Memory technology device (MTD) subsystem for Linux*, 2005.

[205] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif, "Abstract specification of the ubifs file system for flash memory," in *International Symposium on Formal Methods*, pp. 190–206, Springer, 2009.

[206] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 371–386, 2016.

[207] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "Dptree: differential indexing for persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.

[208] G. Graefe, "Efficient columnar storage in b-trees," *ACM SIGMOD Record*, vol. 36, no. 1, pp. 3–6, 2007.

[209] P. Krishnan, D. Raz, and Y. Shavitt, "The cache location problem," *IEEE/ACM transactions on networking*, vol. 8, no. 5, pp. 568–582, 2000.

[210] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 381–392, IEEE, 2014.

[211] K. Wu, K. Tu, Y. Patel, R. Sen, K. Park, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Nyxcache: Flexible and efficient multi-tenant persistent memory caching," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pp. 1–16, 2022.

[212] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for olap workloads," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 339–351, 2021.

[213] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci, "Rethinking file mapping for persistent memory," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 97–111, 2021.

[214] Y. Yang, Q. Cao, J. Yao, Y. Dong, and W. Kong, "Spmfs: A scalable persistent memory file system on optane persistent memory," in *50th International Conference on Parallel Processing*, pp. 1–10, 2021.

[215] G. Graefe and H. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *Proceedings of the 13th International Conference on Extending Database Technology*, pp. 371–381, 2010.

[216] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "An experimental evaluation and analysis of database cracking," *The VLDB Journal*, vol. 25, no. 1, pp. 27–52, 2016.

[217] L. Li, P. Jin, C. Yang, Z. Wu, and L. Yue, "Optimizing b+-tree for pcm-based hybrid memory.," in *EDBT*, pp. 662–663, 2016.

[218] R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[219] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 2, pp. 202–207, 2011.

[220] S. Liang, Z. Qiao, S. Tang, J. Hochstetler, S. Fu, W. Shi, and H.-B. Chen, "An empirical study of quad-level cell (qlc) nand flash ssds for big data applications," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 3676–3685, 2019.

[221] B. W. Low, B. Y. Ooi, and C. S. Wong, "Scalability of database bulk insertion with multi-threading," in *Software Engineering and Computer Systems* (J. M. Zain, W. M. b. Wan Mohd, and E. El-Qawasmeh, eds.), (Berlin, Heidelberg), pp. 151–162, Springer Berlin Heidelberg, 2011.

[222] D. Mrozek, B. Socha, S. Kozielski, and B. Małysiak-Mrozek, "An efficient and flexible scanning of databases of protein secondary structures: With the segment index and multithreaded alignment," *Journal of Intelligent Information Systems*, vol. 46, pp. 213–233, 2016.

[223] P. Garcia and H. F. Korth, "Database hash-join algorithms on multithreaded computer architectures," CF '06, (New York, NY, USA), p. 241–252, Association for Computing Machinery, 2006.

[224] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah, "Improving database performance on simultaneous multithreading processors," 2005.

[225] J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross, "Realizing parallelism in database operations: Insights from a massively multithreaded architecture," DaMoN '06, (New York, NY, USA), p. 4–es, Association for Computing Machinery, 2006.

[226] H. Zhang, E. Thompson, N. Ye, D. Nissim, S. Chi, and H. Takiar, "Ssd thermal throttling prediction using improved fast prediction model," in *2019 18th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pp. 1016–1019, 2019.

[227] R. Khamamkar and A. Keswani, "Analyzing power management in non-volatile memory express (nvme) solid state drives," in *2019 IEEE Pune Section International Conference (PuneCon)*, pp. 1–4, 2019.

[228] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[229] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 93–96, IEEE, 2018.

[230] T.-Y. Wang, C.-F. Wu, C.-W. Tsao, Y.-H. Chang, and T.-W. Kuo, "Scheduling-aware prefetching: Enabling the pcie ssd to extend the global memory of gpu device," in *2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, 2021.

[231] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of gpus and cpus for database analytics," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, (New York, NY, USA), p. 1617–1632, Association for Computing Machinery, 2020.

[232] R. Lee, M. Zhou, C. Li, S. Hu, J. Teng, D. Li, and X. Zhang, "The art of balance: A rateupdb™ experience of building a cpu/gpu hybrid database product," *Proc. VLDB Endow.*, vol. 14, p. 2999–3013, jul 2021.

[233] H. K. H. Subramanian, B. Gurumurthy, G. C. Durand, D. Broneske, and G. Saake, "Analysis of gpu-libraries for rapid prototyping database operations : A look into library support for database operations," in *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pp. 36–41, 2021.