Przechowywanie oraz manipulowanie danymi z uwzględnieniem nowoczesnych rodzajów pamięci

Autoreferat

Dr inż. Wojciech Macyna
Wydział Informatyki i Telekomunikacji
Politechnika Wrocławska

Wrocław 2025

Spis treści

1 IMIĘ I NAZWISKO

Wojciech Macyna

2 POSIADANE DYPLOMY I STOPNIE NAUKOWE

- **Doktor nauk technicznych** w zakresie informatyki (dyscyplina informatyka); Wydział Informatyki i Zarządzania, Politechnika Wrocławska, 2004. Tytuł rozprawy: *Weryfikacja dynamicznych więzów integralności specyfikowanych w logice temporalnej z operatorami metrycznymi.* **Promotor: dr hab. inż. Zygmunt Mazur.**
- **Magister inżynier** (informatyka, specjalność: systemy informacji naukowo-technicznej); Wydział Informatyki i Zarządzania, Politechnika Wrocławska, 1997. Tytuł pracy magisterskiej: *Obiektowo-temporalne bazy danych.* **Promotor: prof. dr hab. inż Ngoc Thanh Nguyen.**

3 DOTYCHCZASOWE ZATRUDNIENIE W JEDNOSTKACH NAUKOWYCH

- 2005 - 2009 Asystent, Instytut Matematyki i Informatyki, Politechnika Wrocławska
- 2009 - 2013 Adiunkt, Instytut Matematyki i Informatyki, Politechnika Wrocławska
- 2013 - 2015 Starszy Wykładowca, Instytut Matematyki i Informatyki, Politechnika Wrocławska
- 2015 - 2020 Starszy Wykładowca, Katedra Informatyki, Wydział Podstawowych Problemów Techniki, Politechnika Wrocławska
- 2020 - 2021 Adiunkt, Katedra Informatyki, Wydział Podstawowych Problemów Techniki, Politechnika Wrocławska
- 2021 - Adiunkt, Katedra Podstaw Informatyki, Wydział Informatyki i Telekomunikacji, Politechnika Wrocławska

4 OMÓWIENIE OSIĄGNIĘĆ, O KTÓRYCH MOWA W ART. 219 UST. 1 PKT. 2 USTAWY Z DNIA 20 LIPCA 2018 R. PRAWO O SZKOLNICTWIE WYŻSZYM I NAUCE

Osiągnięciem jest jednotematyczny cykl publikacji zatytułowany: **"Przechowywanie oraz manipulowanie danymi z uwzględnieniem nowoczesnych rodzajów pamięci".**

Osiągnięcie naukowe zostało podzielone na trzy części. Pierwsza część (rozdział 4.2) opisuje badania związane z przechowywaniem danych na pamięci flash (prace: [A1],[A2], [A3],[A4],[A5],[A6],[A7]). W drugiej części (rozdział 4.3) znajdują się wyniki badań dotyczących indeksowania na drzewie LSM (prace: [A8],[A9],[A10]). Natomiast w części trzeciej (rozdział 4.4) opisane są badania związanie z indeksowaniem na pamięci PCM (prace: [A11], [A12]). Każda z tych części posiada podrozdziały, które zawierają opis poszczególnych publikacji oraz wkład do dyscypliny.

4.1 Publikacje wchodzące w skład osiągnięcia

[A1] Jacek Cichoń, Wojciech Macyna. Approximate counters for flash memory. *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2011, Toyama, Japan, August 28-31, 2011, Volume 1*, strony 185–189. IEEE Computer Society, 2011.

[A2] Maciej Pawlik, Wojciech Macyna. Implementation of the aggregated r-tree over flash memory. Hwanjo Yu, Ge Yu, Wynne Hsu, Yang-Sae Moon, Rainer Unland, Jaesoo Yoo, redaktorzy, *Database Systems for Advanced Applications - 17th International Conference, DASFAA 2012, International Workshops: FlashDB, ITEMS, SNSM, SIM3, DQDI, Busan, South Korea, April 15-19, 2012. Proceedings*, wolumen 7240 serii *Lecture Notes in Computer Science*, strony 65–72. Springer, 2012.

[A3] Wojciech Macyna, Krzysztof Majcher. Cost-based storage of the r-tree aggregated values over flash memory. *2018 International Conference on Industrial Enterprise and System Engineering (ICoIESE 2018)*, strony 97–102. Atlantis Press, 2019.

[A4] Macyna Wojciech, Kukowski Michal. Flash-aware clustered index for mobile databases. *2018 International Conference on Industrial Enterprise and System Engineering (ICoIESE 2018)*, strony 25–30. Atlantis Press, 2019.

[A5] Wojciech Macyna, Michal Kukowski. Partially indexing on flash memory. Sven Hartmann, Josef Küng, Sharma Chakravarthy, Gabriele Anderst-Kotsis, A Min Tjoa, Ismail Khalil, redaktorzy, *Database and Expert Systems Applications - 30th International Conference, DEXA 2019, Linz, Austria, August 26-29, 2019, Proceedings, Part I*, wolumen 11706 serii *Lecture Notes in Computer Science*, strony 95–105. Springer, 2019.

[A6] Krzysztof Kwiatkowski, Wojciech Macyna. CFTL - flash translation layer for column oriented databases. Ali Selamat, Ngoc Thanh Nguyen, Habibollah Haron, redaktorzy, *Intelligent Information and Database Systems - 5th Asian Conference, ACIIDS 2013, Kuala Lumpur, Malaysia, March 18-20, 2013, Proceedings, Part I*, wolumen 7802 serii *Lecture Notes in Computer Science*, strony 146–155. Springer, 2013.

[A7] Wojciech Macyna, Michal Kukowski. Flash-aware storage of the column oriented databases. *Fundam. Informaticae*, 173(1):47–72, 2020.

[A8] Wojciech Macyna, Michal Kukowski. Bulk loading of the secondary index in lsm-based stores for flash memory. Silvia Chiusano, Tania Cerquitelli, Robert Wrembel, Kjetil Nørvåg, Barbara Catania, Genoveva Vargas-Solar, Ester Zumpano, redaktorzy, *New Trends in Database and Information Systems - ADBIS 2022 Short Papers, Doctoral Consortium and Workshops: DOING, K-GALS, MADEISD, MegaData, SWODCH, Turin, Italy, September 5-8, 2022, Proceedings*, wolumen 1652 serii *Communications in Computer and Information Science*, strony 133–143. Springer, 2022.

[A9] Wojciech Macyna, Michal Kukowski, Michal Zwarzko. Multi-core adaptive merging of the secondary index for lsm-based stores. Christine Strauss, Toshiyuki Amagasa, Gabriele Kotsis, A Min Tjoa, Ismail Khalil, redaktorzy, *Database and Expert Systems Applications - 34th International Conference, DEXA 2023, Penang, Malaysia, August 28-30, 2023, Proceedings, Part II*, wolumen 14147 serii *Lecture Notes in Computer Science*, strony 245–257. Springer, 2023.

[A10] Wojciech Macyna, Carlos Ordonez. A bloom filter hierarchy for non-key search in key-value stores. *2024 IEEE International Conference on Big Data (BigData)*, strony 3712–3719, 2024.

[A11] Maciej Jurga, Wojciech Macyna. Implementation of the aggregated r-tree for phase change memory. Sven Hartmann, Hui Ma, Abdelkader Hameurlain, Günther Pernul, Roland R. Wagner, redaktorzy, *Database and Expert Systems Applications - 29th International Conference, DEXA 2018, Regensburg, Germany, September 3-6, 2018, Proceedings, Part II*, wolumen 11030 serii *Lecture Notes in Computer Science*, strony 301–309. Springer, 2018.

[A12] Wojciech Macyna, Michal Kukowski. Adaptive merging on phase change memory. *Fundam. Informaticae*, 188(2):103–126, 2022.

4.2 Przechowywanie danych na pamięci flash

Przez ostatnich kilka dekad systemy bazodanowe wykorzystywały twarde dyski do przechowywania danych. Podstawowym elementem twardego dysku jest znajdujący się w obudowie wirujący talerz lub zespół talerzy. Zapis i odczyt danych zapewniają głowice, które ustawiane są w odpowiedniej pozycji względem obracających się talerzy. Taka budowa powoduje stosunkowo wolny czas działania tych urządzeń oraz sprawia, że są one delikatne i wrażliwe na bodźce zewnętrzne. W odpowiedzi na niedoskonałości twardych dysków pojawiły się nowe nośniki pamięci. Jednym z nich jest pamięć flash.

Pamięć flash jest pamięcią blokową. Oznacza to, że operacje odczytu i zapisu można wykonać tylko na ciągłych obszarach pamięci zwanych stronami (ang. pages). Zazwyczaj strona ma pojemność 2KB-16KB i składa się z wielu komórek pamięci (ang. memory cell). Pamięć flash posiada również inne ograniczenia wynikające ze swojej architektury. Aby nadpisać dane na stronie pamięci należy najpierw wykonać operację usuwania wszystkich danych, które się tam znajdują. Operacja usuwania musi jednak zostać wykonana na całym bloku pamięci (ang. memory block), który zawiera zwykle

od 32 do 128 stron. Konsekwencją takiej architektury jest duża asymetria pomiędzy szybkością odczytu i zapisu. Dodatkowo, koszt zużycia energii jest również znacznie większy podczas zapisu niż podczas odczytu z pamięci flash. Ponadto, każda strona pamięci flash ma ograniczoną żywotność. Zbyt częste nadpisywanie tych samych stron pamięci może doprowadzić do ich szybkiego zniszczenia. Z powyższych faktów wynika, że w celu uzyskania efektywnych systemów opartych o pamięć flash należy redukować liczbę operacji usuwania i zapisu.

Najpopularniejszym i najczęściej wykorzystywanym rodzajem pamięci flash jest pamięć typu NAND. Występuje ona w wielu urządzeniach elektronicznych, takich jak dyski SSD, pamięci USB, karty pamięci oraz smartfony. Charakteryzuje się dużą pojemnością, stosunkowo niskim kosztem oraz trwałością zapisu danych nawet po odłączeniu zasilania. Dyski SSD są najbardziej typowym zastosowaniem pamięci NAND. Jako że składają się one z wielu kości pamięci flash, to ich charakterystyka oraz ograniczenia są identyczne jak w przypadku pamięci flash. Obecne dyski SSD używają dwóch typów interfejsów: wolniejszego SATA (ang. Serial Advanced Technology Attachment) [38], [51] o przepustowości maksymalnej 750MB/s oraz szybszego M.2 (ang. Next Generation Form Factor) [55] wykorzystującego złącza PCIe5 (ang. Peripheral Component Interconnect Express) o maksymalnej przepustowości wynoszącej aż 4GB/s. Interfejs M.2 może zostać użyty w protokole NVMe (ang. Non-Volatile Memory Express) [47], [76], 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.

Badania nad pamięcią NAND są bardzo aktywnie prowadzone w kilku kierunkach. Głównym trendem jest tworzenie coraz efektywniejszej architektury takiej pamięci. Przykładem takich badań mogą być prace: [46], [49], [15], [50], [79], [48]. Kolejny kierunek badawczy to tworzenie nowych metod oprogramowania i przechowywania danych na tych pamięciach, a także nowe możliwości ich zastosowania: [72], [20], [27], [56]. Więcej na temat problemów oraz wyzwań stojących przez pamięcią NAND można znaleźć w pracach: [35] i [88].

Dalsza część rozdziału opisuje mój wkład w następujące zagadnienia związane z pamięcią flash. Są to: liczniki probabilistyczne do przechowywania liczby usunięć bloków pamięci flash, indeksowanie danych oraz przechowywanie danych kolumnowych baz danych na pamięci flash.

4.2.1 Licznik probabilistyczny

Celem podrozdziału jest przedstawienie licznika probabilistycznego stosowanego do mierzenia stopnia zużycia bloków pamięci flash.

Jak to już zostało wcześniej wspomniane, nadmierne wymazywanie danych z poszczególnych bloków pamięci flash może doprowadzić do ich zniszczenia a w konsekwencji całej pamięci flash. Blok pamięci flash może być usuwany pomiędzy 10^5 a 10^6 razy. Tak więc rozkład zapisu do pamięci powinien być równomierny. Aby to zrobić, każdy blok powinien posiadać swój licznik usunięć. Naiwne podejście polega na przechowywaniu dokładnych liczników związanych z każdym blokiem pamięci (zobacz prace: [62],[63],[66]). Okazuje się jednak, że aż taka dokładność nie jest potrzebna. Żywotność bloku pamięci jest określona w sposób przybliżony, więc zastosowanie licznika przybliżonego jest wystarczające w tym przypadku.

Zaproponowany w pracy [69] licznik probabilistyczny działa w sposób następujący. Na początku licznik (C) przyjmuje wartość 0. Następnie wybierana jest losowa wartość z przedziału [0,1]. Gdy wylosowana wartość jest mniejsza niż 2^{-C}, wówczas licznik jest inkrementowany. Łatwo za-

uważyć, że im większa wartość licznika C, tym inkrementacja ta będzie rzadziej występować. W konsekwencji, jedynie $\log_2 \log_2 n$ bitów jest potrzebnych, żeby przechowywać wartość licznika po n inkrementacjach. Warto tutaj zauważyć, że w przypadku licznika standardowego potrzebnych jest $\lfloor \log_2 n \rfloor + 1$ bitów. Tak więc w celu przechowania wartość 1000000, licznik standardowy potrzebuje 21 bitów a licznik probabilistyczny tylko 5 bitów.

W pracy [A1] wprowadziliśmy system liczników probabilistycznych do mierzenia liczby operacji usuwania danych z bloków pamięci flash. Każdy blok pamięci flash jest wyposażony w jeden licznik probabilistyczny. W pracy udowodniliśmy formalnie, że efektywność użycia systemu liczników probabilistycznych oraz standardowych jest podobna dla pamięci flash mimo tego, że liczniki probabilistyczne zajmują mniej pamięci. Ponadto pokazaliśmy, że po wymazaniu n bloków liczba zmian w pamięci flash związanych z modyfikacją systemu liczników probabilistycznych wynosi $O(\log n)$. W przypadku liczników standardowych jest to $O(n)$. Uzyskaliśmy w ten sposób podwójny efekt: zmniejszyliśmy znacznie rozmiar licznika związanego z każdym blokiem pamięci flash z $O(\log n)$ na $O(\log \log n)$ oraz ograniczyliśmy liczbę zmian w pamięci flash z liniowej na logarytmiczną, co ma duży wpływ na trwałość pamięci flash, gdzie liczba wymazań poszczególnych bloków jest ograniczona. Aby potwierdzić eksperymentalnie efektywność zaproponowanej metody, zaimplementowaliśmy symulator zarządzania pamięcią flash zaopatrzony w system liczników probabilistycznych i porównaliśmy jego efektywność z takim samym symulatorem, który ma system liczników standardowych. Zaproponowany przez nas system liczników probabilistycznych wzbudził zainteresowanie środowiska algorytmicznego (zobacz praca: [73]).

4.2.2 Indeksowanie na pamięci flash

Indeksy to struktury używane w celu przyspieszenia dostępu do danych zapamiętanych w plikach bazy danych. Baza danych posiada zazwyczaj jeden indeks podstawowy (ang. primary index) oraz może posiadać wiele różnych indeksów wtórnych (ang. secondary index). Z reguły indeksy oparte są na strukturze zwanej B+-drzewem (ang. B+-tree) [4]. Ponadto istnieje wiele innych typów indeksów. Są to między innymi indeksy przestrzenne (ang. spatial index), indeksy bitmapowe (ang. bitmap index), indeksy prefiksowe (ang. trie index), itd. Warto nadmienić, że do każdego z tych typów indeksów należy wiele różnych odmian. Przykładowo, jedną z odmian indeksu przestrzennego jest R-drzewo (ang. R-tree) [26].

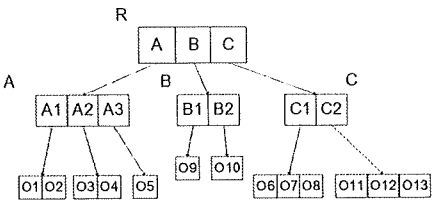
Indeksy są zaimplementowane efektywnie dla pamięci masowych opartych o tradycyjne twarde dyski. Niemniej jednak, jak to już zostało wcześniej wspomniane, pamięć flash posiada odmienną charakterystykę niż twarde dyski, co powoduje konieczność ponownego zaimplementowania tego rodzaju indeksów. Nowe koncepcje B+-drzewa dla pamięci flash zostały zaproponowane w następujących pracach: [61], [2], [30], [40], [41], [42], [39], [43]. Z kolei implementacje indeksów przestrzennych dla pamięci flash zostały przedstawione w następujących pracach: [8], [7], [39], [58], [78], [85], [77], [9], [19].

Niniejszy podrozdział porusza trzy problemy. Pierwsza część (podrozdział 4.2.2.1) przedstawia metody przechowywania wartości zagregowanych związanych z R-drzewem w sposób zoptymalizowany dla pamięci flash. W części drugiej (podrozdział 4.2.2.2) opisano nową strukturę (FA-drzewo), dzięki której można w efektywny sposób przechowywać indeks klastrowy z uwzględnieniem charakterystyki pamięci flash. W ostatniej części (podrozdział 4.2.2.3) zaproponowano technikę leniwego indeksowania częściowego (ang. lazy adaptive merging) dla indeksu wtórnego (ang. secondary index)

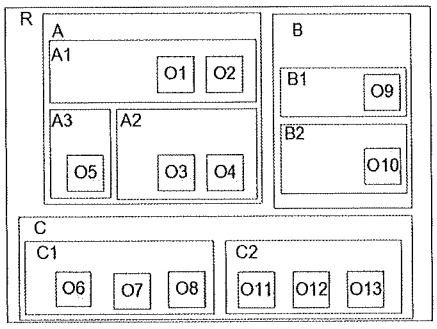
na pamięci flash.

4.2.2.1 Przechowywanie wartości zagregowanych w R-drzewie.

R-drzewa są popularnym indeksem służącym do przechowywania danych przestrzennych oraz wspomagającym wyszukiwanie obiektów w przestrzeni wielowymiarowej (zobacz [85]). Struktura R-drzewa jest podobna do struktury B-drzewa. Rysunek 1 przedstawia R-drzewo, które jest indeksem dla obszaru R podzielonego na prostokąty A, B, C (zobacz rysunek 2). Prostokąty te z kolei dzielą się na mniejsze obszary, które tworzą liście drzewa: $A1$, $A2$, $A3$, $B1$, $B2$, $C1$, $C2$. Obszary tworzące liście R-drzewa mogą zawierać różne obiekty. Taka struktura pozwala łatwo zidentyfikować specyficzne obiekty znajdujące się na danym obszarze. Z każdym obszarem R-drzewa mogą być związane wartości zagregowane. Przykładem może być liczba samochodów znajdujących się na danym obszarze w danej chwili lub średnia wysokość budynków mieszkalnych. Łatwo zauważyć, że o ile samo R-drzewo dla danego obszaru będzie się zmieniało bardzo rzadko, to wartości zagregowane mogą się zmieniać częściej. Tradycyjne metody zakładają przechowywanie R-drzewa oraz wartości zagregowanych w tym samym miejscu w pamięci. Takie podejście jest efektywne w przypadku twardych dysków, gdzie dane mogą być nadpisywane w tym samym sektorze pamięci bez straty efektywności odczytu i zapisu oraz bez narażania się na szybkie zużycie poszczególnych sektorów pamięci ([23]).



Rysunek 1: R-drzewo dla zaindeksowanego obszaru



Rysunek 2: Zaindeksowany obszar

W pracy [A2] zaproponowaliśmy metodę, która zakłada przechowywanie wartości zagregowanych w osobnych blokach pamięci flash. Dzięki takiemu podejściu każda zmiana wartości zagregowanej nie spowoduje niepotrzebnego uaktualniania danych znajdujących się na blokach pamięci zajmowanych przez dane związane z samym R-drzewem. Zaproponowana metoda polega na rozszerzeniu koncepcji opracowanej w pracy [85], która zakłada użycie tablicy indeksów mapujących wierzchołki drzewa do odpowiednich bloków pamięci flash. Dzięki temu odczytywanie danych wierzchołków R-drzewa polega na bezpośrednim odczycie wpisanych w tablicy indeksów bloków pamięci flash. Z kolei modyfikacja wierzchołków polega na wpisywaniu nowych danych do osobnych bloków pamięci flash i uaktualnianiu tego w tablicy indeksów. W pracy [A2] dodaliśmy tablicę agregującą, które odpowiada za mapowanie wartości zagregowanych do bloków pamięci flash. Problemem jest takie rozmieszczenie wartości zagregowanych do odpowiednich bloków pamięci, aby liczba zapisanych bloków była minimalna. Problem ten jest problemem NP-trudnym, który można sprowadzić

do problemu pakowania (ang. Bin-Packing problem). Do rozwiązania tego problemu wybraliśmy algorytm aproksymacyjny FIRST-FIT (zobacz [84]). Z obliczeń wynika, że do zapisania k wartości agregujących potrzeba $R_1 = O(2 \cdot k \cdot H \cdot (C+1))$ odczytów oraz $W_1 = O(2 \cdot (\frac{k \cdot H}{r}))$ zapisów do pamięci flash, gdzie H jest wysokością drzewa, C jest minimalną liczbą bloków pamięci flash powiązanych z jednym wierzchołkiem drzewa, a r jest liczbą wartości zagregowanych, które można wpisać do pojedynczego bloku pamięci flash. Z kolei liczba zapisów i odczytów z użyciem metody przedstawionej w [85] wynosi odpowiednio: $R_2 = O(k \cdot H \cdot (C+1))$ oraz $W_2 = O(2 \cdot (\frac{k \cdot H}{r}))$, gdzie f oznacza liczbę wpisów indeksowych (ang. index items) w bloku pamięci. Biorąc pod uwagę zależność: $f \ll r$ łatwo zauważyć, że zaproponowana metoda potrzebuje wykonywania znacznie mniej operacji zapisu w porównaniu z metodą tradycyjną. Jako że prędkość oraz koszt energetyczny operacji zapisu jest kilkakrotnie większy od prędkości i kosztu energetycznego operacji odczytu, zaproponowana metoda jest znacznie bardziej wydajna niż metoda tradycyjna przedstawiona w pracy [85]. Efektywność zaproponowanej metody oraz przedstawione wyżej oszacowania zostały potwierdzone eksperymentalnie.

Praca [A3] poświęcona jest kontynuacji badań zapoczątkowywanych w pracy [A2]. Okazuje się bowiem, że nie wszystkie wartości zagregowane muszą być zapisywane na pamięci flash. Niektóre z nich mogą być łatwo obliczone na podstawie innych, zapisanych już wcześniej wartości zagregowanych. Na przykład możemy łatwo obliczyć liczbę obiektów w obszarze A, jeśli mamy zapisaną liczbę obiektów w obszarach $A1$, $A2$ oraz $A3$. Wówczas zapisywanie liczby obiektów w obszarze A może być bardziej kosztowne niż ich obliczanie na podstawie zapisanych wcześniej wartości związanych z $A1$, $A2$ oraz $A3$. W pracy [A3] zaproponowaliśmy metodę wybierania wartości zagregowanych do zapisu opartą na kosztach zapisu i odczytu z pamięci flash. Metoda ta umożliwia przechowywanie najbardziej kosztowo korzystny podzbiór wartości zagregowanych. Koszt jest ściśle związany z kosztem dostępu do pamięci flash i jest zdefiniowany w następujący sposób. Niech AP oznacza zbiór wartości zagregowanych I zapamiętanych w buforze na pamięci flash. Niech wartości r_I, w_I oraz d_I oznaczają odpowiednio koszt odczytu, zapisu oraz usunięcia pojedynczej wartości zagregowanej z bufora. Ponadto r_L oznacza koszt odczytu wartości zagregowanej związanej z liściem R-drzewa ($R_{leafnode}$). Koszt ewaluacji zapytania dla węzła $c(R)$ jest zdefiniowany następująco:

$$c(R) = \begin{cases} 0 \text{ jeśli } I_R \in RAM \\ r_I \text{ jeśli } I_R \in AP \\ r_L \text{ jeśli } R \in R_{leafnodes} \\ \sum_{R_j \in R_{children}} c(R_j) \text{ w przeciwnym przypadku} \end{cases}$$

W pracy przedstawiliśmy algorytm polegający na usuwaniu najmniej korzystnych kosztowo wartości zagregowanych oraz zastępowaniu ich przez inne, ostatnio obliczane. Aby metoda była efektywna dla pamięci flash, zaproponowaliśmy warstwę translacji umieszczoną na pamięci flash (ang. flash translation layer) składającą się z osobnych bloków pamięci przechowujących dane niezmodyfikowane (D-blocks) oraz zmodyfikowane i usunięte (U-blocks). Dzięki temu w przypadku usuwania i modyfikacji wartości zagregowanech, wartości te nie są od razu usuwane z pamięci, ale są oznaczane jako "do usunięcia" (ang. obsolete). Gdy bloków typu "U-blocks" jest zbyt dużo, wówczas następuje ich usuwanie przy jednoczesnym przenoszeniu danych tam przechowywanych do bloków typu "D-blocks". W pracy przeprowadziliśmy szereg eksperymentów z których wynika, że metoda jest średnio o 50% lepsza niż tradycyjne R-drzewo, które nie zapisuje wartości zagregowanych w

osobnym buforze oraz R-drzewo, które zapisuje wartości zagregowane w sposób nie uwzględniający kosztów zapisu i odczytu pamięci flash.

4.2.2.2 Indeks klastrowy dla pamięci flash.

W wielu systemach zarządzania bazami danych (np. SQL Server) tabele są wyposażone w indeks klastrowy, który determinuje fizyczny układ danych w tabeli. W tabeli może być tylko jeden indeks klastrowy, ponieważ dane mogą być posortowane tylko według jednej kolumny (lub kombinacji kolumn). Wartości w kolumnie (lub kolumnach) indeksu klastrowego określają w jakiej kolejności są przechowywane wiersze w tabeli. Z reguły indeks klastrowy jest oparty na strukturze B+-drzewa (ang. B+-tree). Z uwagi jednak na fakt, że B+-drzewo nie jest efektywną strukturą w przypadku pamięci flash, w pracy [A4] zaproponowaliśmy nową strukturę zwaną FA-drzewem (ang. FA-tree). FA-drzewo składa się z poziomów, przy czym każdy poziom jest przechowywany na przylegających blokach pamięci flash. FA-drzewo ma określony parametr k, który określa stosunek liczby rekordów na sąsiadujących poziomach. Tak więc poziom L_{i+1} przechowuje k razy więcej rekordów niż poziom L_i. Ponadto, poszczególne poziomy mają zdefiniowany parametr t określający liczbę rekordów, które mogą przechowywać. Jeśli parametr ten zostanie przekroczony dla poziomu L_i, wówczas poziom ten łączy się z poziomem L_{i+1}. Łączenie w FA-drzewie jest więc głównym kosztem, na który składa się odczyt danych z poziomów L_i i L_{i+1} oraz zapis odczytanych danych do nowego poziomu L'_{i+1}.

Na tak zdefiniowanej strukturze oparliśmy indeks klastrowy dla relacyjnych baz danych przechowywanych na pamięci flash. Jako że rekordy są przechowywane w strukturze drzewiastej, to zachowany jest logarytmiczny czas wyszukiwania rekordów. Z drugiej strony, rekordy mogą być przechowywane nie tylko w liściach, ale także na pośrednich poziomach drzewa. W rezultacie unikamy reorganizacji całego drzewa podczas wstawiania, a dzięki zastosowaniu łączenia poziomów oraz możliwości wstawiania wielu rekordów na raz (ang. bulk loading) zmniejszamy liczbę kosztownych operacji usuwania na pamięci flash.

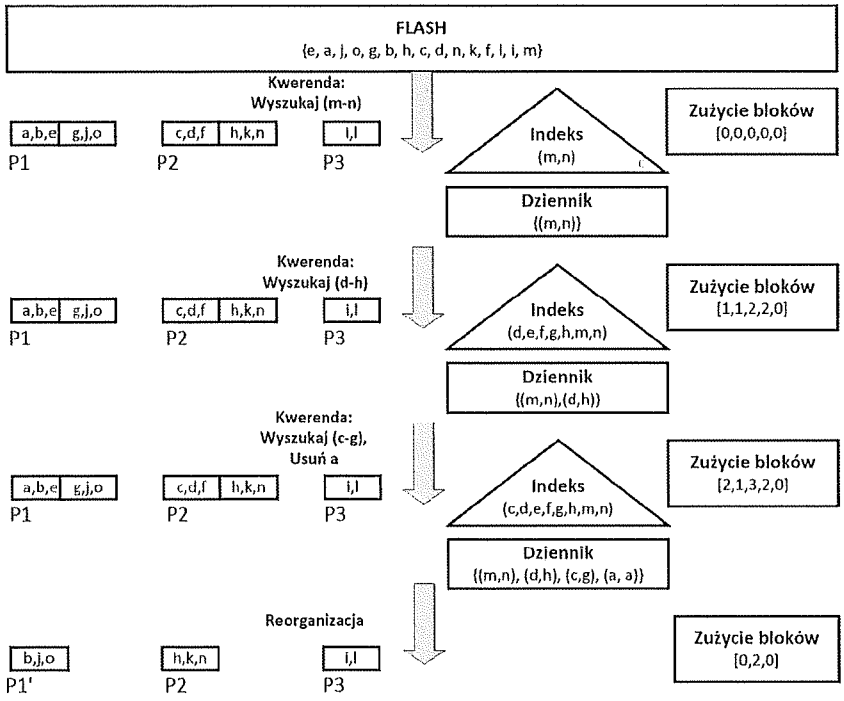
W pracy pokazaliśmy estymację kosztu odczytu i zapisu (rozumianego jako liczba stron sekwencyjnie odczytanych i zapisanych do pamięci flash). Koszt wyszukiwania elementu w FA-drzewie zawierającym n rekordów wynosi $O(\log_k n)$. Z kolei koszt zapisu i odczytu podczas wstawiania elementu do FA-drzewa zawierającego n rekordów wynosi: $O(\frac{k}{b-k} \cdot \log_k n)$, gdzie b oznacza liczbę rekordów, który może pomieścić jedna strona pamięci flash, a k oznacza stosunek ilości elementów zapisanych w sąsiadujących poziomach. Dla porównania koszt zapisu i odczytu podczas wstawiania elementu indeksu klastrowego opartego o B+-drzewa zawierającego n rekordów wynosi: $O(\log_b n)$. Dodatkowo mamy tutaj do czynienia z zapisem sekwencyjnym, gdzie czas takiego zapisu jest kilkakrotnie niższy niż czas zapisu losowego (ang. random write), który występuje w przypadku indeksu klastrowego opartego na B+-drzewie. Z obliczeń wynika, że indeks oparty o taką strukturę jest lepszy niż tradycyjny indeks klastrowy. W pracy została przeprowadzona seria eksperymentów dla różnych zestawów danych, które potwierdzają tę tezę.

4.2.2.3 Indeksowanie częściowe na pamięci flash.

W pracy [A5] zaproponowano technikę leniwego częściowego indeksowania (ang. lazy adaptive merging) dla indeksu wtórnego na pamięci flash.

Z reguły wszystkie rekordy w tabeli bazy danych są objęte indeksem wtórnym (ang. secondary

index). Jednak takie podejście nie jest skuteczne szczególnie w dużych bazach danych, gdzie niektóre rekordy są często odczytywane, podczas gdy inne są rzadko lub w ogóle nie pobierane przez zapytania. Załóżmy, że tabela bazy danych *Sprzedaż* przechowuje historię transakcji sprzedaży. Jeżeli większość zapytań dotyczy danych z ostatniego roku, nie warto indeksować całej tabeli po kolumnie *Data*. W takim przypadku bardziej opłaca się indeksować tylko dane z ostatniego roku, gdyż dane z innych okresów czasowych są odpytywane bardzo rzadko. Problem ten można rozwiązać poprzez częściowe indeksowanie. Kluczową ideą jest tworzenie indeksu wtórnego częściowego jako produktu ubocznego wysyłanych zapytań do bazy danych. W rezultacie tabela bazy danych jest indeksowana częściowo w zależności od obciążenia zapytaniami. Problem częściowego indeksowania jest szeroko rozpatrywany w literaturze: [33], [32],[34], [45]), [24], [53], [90], [86], [54]. Nie ma jednak pracy, która rozpatrywałaby ten problem na pamięci flash.



Rysunek 3: Proces częściowego indeksowania.

Zaproponowana w pracy [A5] metoda częściowego indeksowania działa w następujący sposób. Załóżmy, że dane są zapisane w sposób nieposortowany i nie są w żadnym indeksie (zobacz rysunek 3). Gdy pojawia się zapytanie do bazy o dane z zakresu $m-n$, wówczas wszystkie elementy indeksu będące wynikiem zapytania (w tym przypadku m i n) są wpisywanie do indeksu częściowego. Dodatkowo zakres zapytania jest wprowadzany do specjalnej struktury zwanej dziennikiem. Wszystkie pozostałe dane są kopiowane do partycji na pamięci flash ($P1$, $P2$ oraz $P3$). Każda partycja składa się z bloków pamięci flash, a dane są posortowane wewnątrz partycji. Z każdą partycją związana jest minimalna i maksymalna wartość danych, które się tam znajdują. Informacja ta jest zapisywana w

RAM. Kiedy pojawia się kolejne zapytanie (np. o dane z zakresu $d-h$), wówczas metoda sprawdza, czy dane z zakresu d i h są w już indeksie. Jeśli tak, to są one zwracane bezpośrednio z indeksu. Jeśli nie, wówczas przeszukiwane są poszczególne partycje w celu ich znalezienia. Następnie, znalezione w wyniku zapytania elementy są zapisywane do indeksu, a zakres zapytania do dziennika. Elementy wyciągnięte z partycji i zapisane do indeksu (zaznaczone na czerwono) nie są natychmiast usuwane z partycji, lecz są usuwane dopiero podczas procesu łączenia partycji. Dodatkowo, metoda używa struktury zwanej tablicą zużycia bloków, która zapisuje, ile danych zostało pobranych z poszczególnych partycji. Dopiero gdy danych w partycji zostanie mało (mniej niż wynosi zadana wartość t), wówczas partycje są łączone. Metoda działa do momentu, aż wszystkie dane z partycji zostaną zapisane w indeksie. Widać więc wyraźnie, że do indeksu są wpisywane tylko takie elementy, które są używane w zapytaniach. Dzięki zastosowaniu technik optymalizacyjnych specyficznych dla pamięci flash, zaproponowana metoda zmniejsza liczbę operacji zapisu i usuwania na pamięci flash podczas tworzenia indeksu. W wyniku tego czas odpowiedzi na zapytania jest skrócony dwukrotnie w porównaniu do tradycyjnych metod. Poprawność stosowanej metody udowodniliśmy w sposób eksperymentalny. Do porównania użyliśmy dysków o różnych parametrach oraz różne wzorce zapytań. Ponadto porównaliśmy naszą metodę ze zwykłym przeszukiwaniem (bez indeksowania), z bazą danych całkowicie poindeksowaną (ang. full index) oraz z metodą wykorzystującą częściowe indeksowanie bez włączonej optymalizacji dla pamięci flash. Warto zaznaczyć, że zaproponowane podejście może być zastosowane niezależnie od modelu danych. Również zastosowany indeks może być dowolnym indeksem zoptymalizowanym dla pamięci flash.

4.2.3 Przechowywanie danych kolumnowych baz danych na pamięci flash

Celem tego podrozdziału jest przedstawienie metody przechowywania kolumnowych baz danych na pamięci flash. Efektem prac jest opracowanie struktury zwanej CF-drzewem (ang. CF-tree), która ogranicza liczbę operacji zapisu na pamięci flash a jednocześnie zachowuje logarytmiczny czas wyszukiwania w kolumnowej bazie danych po atrybucie należącym do klucza głównego tabeli.

W tradycyjnym systemie relacyjnych baz danych dane są przechowywane wierszowo jako zbiór rekordów, przy czym każdy rekord składa się z zestawu atrybutów (kolumn) przechowywanych z reguły na przylegających do siebie blokach pamięci. Z kolei kolumnowa baza danych przechowuje dane w sposób zorientowany kolumnowo. Tak więc dane z poszczególnych kolumn przechowywane są fizycznie obok siebie. Dzięki takiej architekturze zapytania, które potrzebują dane tylko z niektórych kolumn, mogą pobierać te dane tylko z tych obszarów pamięci, w których dane tych kolumn są przechowywane, co radykalnie zmniejsza liczba operacji dyskowych. Poza tym, składowanie danych tego samego typu umożliwia wydajną kompresję tych danych. Przechowywanie danych w kolumnowych bazach danych jest aktywnym polem badań (zobacz prace: [81], [29], [3], [25]).

W pracy [A6] zaproponowano kolumnowy poziom translacji CFTL (ang. Column Flash Translation Layer), który umożliwia integrację kolumnowego formatu baz danych z tradycyjnym FTL (ang. flash translation layer). Struktura CFTL przechowuje dane poszczególnych kolumn z wykorzystaniem mapowania S-Map i P-Map. S-Map jest listą bloków pamięci flash, natomiast P-Map jest listą stron w poszczególnych blokach. W przypadku operacji *delete* oraz *update* na kolumnowej bazie danych dane z niektórych stron pamięci flash muszą być uaktualnione i przeniesione w inne miejsce (do innych bloków). Kolumnowy format przechowywania danych musi posiadać mechanizm, dzięki któremu dane z poszczególnych kolumn mogą być połączone w rekord. Struktura P-Map umożliwia

to dzięki łatwemu manipulowaniu listy stron pamięci flash, na których znajdują się dane z poszczególnych kolumn. W pracy obliczyliśmy liczbę stron pamięci, które należy pobrać, w zależności od rodzaju zapytania (tj. rozmiaru tabeli, ilości zwracanych rekordów oraz liczby zapytywanych atrybutów) dla kolumnowej oraz wierszowej bazy danych.

Niech x, y oznaczają odpowiednio liczbę kolumn i rekordów w tabeli. Załóżmy także, że r to rozmiar strony pamięci flash, a $z(i)$ rozmiar kolumny i, przy czym $i \in \{1, 2, .., x\}$. Wówczas: $E_r = \left\lfloor \frac{r}{\sum_{i=1}^{x} z(i)} \right\rfloor$ określa liczbę wierszy na jednej stronie pamięci w wierszowym modelu danych. Z kolei: $E_c(z) = \lfloor \frac{r}{z} \rfloor$ to liczba elementów kolumny o rozmiarze z umieszczonych na jednej stronie. Na tej podstawie możemy oszacować całkowitą liczbę stron pamięci pobieranych dla zapytania dla modelu wierszowego i kolumnowego jako: $S_r \leqslant \left\lceil \frac{y}{E_r} \right\rceil$ oraz $S_c = \sum_{i=1}^{x} \left\lceil \frac{y}{E_c(z(i))} \right\rceil$. Jeśli więc zapytanie zwraca wszystkie kolumny tabeli, wówczas mamy $S_c = S_r$. Z kolei jeśli nie wszystkie kolumny muszą być pobrane, wówczas: $S_c \leqslant S_r$. Jako że czas procesowania zapytań jest ściśle związany z ilością pobieranych stron pamięci, zastosowanie naszej metody dla kolumnowych baz danych podnosi efektywność systemu. W pracy potwierdzono doświadczalnie efektywność zaproponowanej metody dla różnych zestawów danych.

Kolejna praca [A7] jest kontynuacją badań przedstawionych w [A6]. Tabele w bazie danych posiadają zwykle klucz główny (ang. primary key) nałożony na atrybut (kolumnę) bądź zestaw atrybutów (kolumn). Wówczas rekordy są posortowane według klucza głównego, dzięki czemu wyszukiwanie po kluczu głównym odbywa się w czasie logarytmicznym. Rozpatrując jednak kolumnowe bazy danych można zauważyć, że fizyczne wstawianie nowych rekordów w taki sposób, aby posortowanie po kluczu głównym było zachowane nie jest trywialne. Z reguły nowe dane są wpisywane na koniec tabeli, przez co posortowanie po kluczu nie jest zachowane. Z kolei ewentualne późniejsze sortowanie całej tabeli po kluczu jest zbyt kosztowne, aby można go było zastosować w praktyce. Celem pracy [A7] jest uzyskanie logarytmicznego czasu modyfikowania kolumnowej bazy danych na pamięci flash przy jednoczesnym zachowaniu posortowania po kluczu głównym, a więc z zachowaniem logarytmicznej wydajności wyszukiwania. Zaproponowaliśmy w niej nowy sposób przechowywania danych kolumnowych dla pamięci flash oraz dysków SSD, które taką pamięć wykorzystują.

Metoda działa w sposób następujący. Zakładamy, że dane każdej kolumny tabeli są przechowywane w osobnej strukturze drzewiastej zwanej CF-drzewem (ang. CF-tree). Cechą charakterystyczną takiego drzewa jest to, że każdy jego poziom jest zbiorem przylegających do siebie stron pamięci (ang. sorted run), które przechowują dane posortowane po kluczu głównym. Drzewo składa się z kilku poziomów przy czy poziom najwyższy jest przechowywany w pamięci RAM. Operacje modyfikacji (*insert*, *update* i *delete*) są przeprowadzane najpierw na najwyższym poziome drzewa. Następnie, gdy ilość danych na tym poziomie przekroczy ustaloną wartość, zostaje on połączony z poziomem niżej. W ten sposób unikamy reorganizacji całego drzewa. Dodatkowo, struktura jest modyfikowana za pomocą zbioru operacji (ang. bulk loading) przy użyciu operacji łączenia poziomów (ang. level merging). Operacja ta wpływa bardzo korzystnie na trwałość pamięci flash, gdyż jest wykonywana za pomocą zapisów sekwencyjnych. Ponadto, zastosowanie wskaźników pomiędzy poziomami (ang. fence pointers) powoduje, że drzewo zachowuje logarytmiczny czas wyszukiwania. W pracy oszacowaliśmy czas operacji odczytu i operacji modyfikacji jako $O(C \cdot \log_k n)$ oraz $O(\frac{C \cdot b}{b-k} \cdot \log_k n)$, gdzie b oznacza liczbę rekordów, który może pomieścić jedna strona pamięci flash, k jest wysokością drzewa, a C jest liczbą kolumn w tabeli. Ponadto, udowodniliśmy formalnie po-

prawność łączenia elementów z poszczególnych kolumn w rekord. Przeprowadziliśmy również szereg eksperymentów potwierdzających przydatność naszej metody. Między innymi porównaliśmy naszą koncepcję do koncepcji bazy danych zorientowanych wierszowo ([1]), z podejściem PAX, gdzie rekordy są przechowywane na ministronach na pamięci flash ([81]). Oprócz tego, zaimplementowaliśmy koncepcję PDT (Positional Delta Tree) wprowadzoną w pracy [29], która wówczas była w naszym odczuciu najlepszą z zaproponowanych metod. Dodatkowo, rozszerzyliśmy implementację PDT o indeksy rzadkie (ang. sparse index), które znacznie poprawiały jej wydajność. Mimo tego, nasze podejście okazało się lepsze od wszystkich opisanych powyżej koncepcji dla różnych zestawów danych. Warto też wspomnieć, że zaproponowana przez nas struktura może być dostosowana do różnych wzorców zapytań dzięki możliwości zmiany struktury drzewa. W przeciwieństwie do metod przedstawionych w [6] zastosowanie naszej metody nie ogranicza się do baz danych zorientowanych na odczyt (ang. read intensive), ale jest efektywne także dla baz danych, w których często występują operacje zapisu (ang. write intensive).

4.2.4 Wkład do dyscypliny.

Na podstawie wyżej przedstawionych prac można zidentyfikować następujący wkład do dyscypliny:

- Zaproponowanie systemu liczników probabilistycznych do przechowywania stopnia zużycia bloków pamięci flash (praca [A1]). W wyniku prac nad tym problemem zmniejszyliśmy rozmiar licznika związanego z każdym blokiem pamięci flash z $O(\log n)$ na $O(\log \log n)$ oraz ograniczyliśmy liczbę zmian w pamięci flash związaną z modyfikacją liczników z $O(n)$ na $O(\log n)$. Wyniki zostały potwierdzone formalnie i eksperymentalnie.

- Opracowanie koncepcji przechowywania danych zagregowanych dla R-drzewa z uwzględnieniem charakterystyki pamięci flash (prace: [A2], [A3]). Przestawiona w pracy [A2] koncepcja jest pierwszym podejściem do tego problemu na pamięci flash. Potwierdziliśmy formalnie i eksperymentalnie wyższość naszej metody porównując ją z jedyną w tym czasie koncepcją przedstawioną w pracy [85]. Dodatkowo w pracy [A3] opracowaliśmy koncepcję selekcjonowania wartości zagregowanych związanych z R-drzewem z uwzględnieniem kosztów zapisu i odczyt na pamięci flash.

- Zaproponowanie indeksu klastrowego (FA-drzewo) dla baz danych działających na pamięci flash. W pracy [A4] potwierdziliśmy formalnie i eksperymentalnie korzyści wynikające z zastosowania indeksu FA-drzewo w stosunku indeksu klastrowego opartego o B+-drzewo na pamięci flash.

- Opracowanie koncepcji indeksowania częściowego na pamięci flash (praca [A5]). Poprawność stosowanej metody udowodniono w sposób eksperymentalny. Do porównania użyliśmy dysków o różnych parametrach oraz różne wzorce zapytań. Ponadto porównaliśmy naszą metodę ze zwykłym przeszukiwaniem (bez indeksowania), z przeszukiwaniem całkowicie poindeksowanej bazy danych (ang. full index) oraz z metodą wykorzystującą częściowe indeksowanie bez włączonej optymalizacji dla flash. Zaproponowana w [A5] koncepcja jest pierwszym frameworkiem do częściowego indeksowania na pamięci flash.

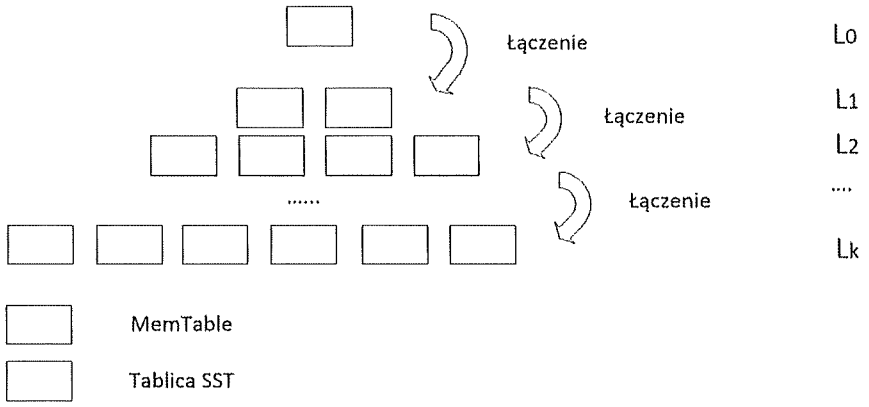
- Stworzenie nowej koncepcji przechowywania kolumnowych baz danych zoptymalizowanych dla pamięci flash ([A6], [A7]). W pracy [A6] przedstawiono pierwsze nasze podejście do przechowywanie danych kolumnowych na pamięci flash. Z kolei praca [A7] jest kontynuacją badań nad tym problemem. Opracowano strukturę zwaną CF-drzewem, na której oparto przechowywanie kolumnowych baz danych. Efektywność metody została oszacowana formalnie. Dodatkowo porównaliśmy naszą koncepcję do metod przedstawionych w [1], [81] oraz [29].

4.3 Indeksowanie na drzewie LSM

Drzewo LSM (ang. LSM-tree) (zobacz [70]) jest popularną strukturą danych stosowaną w wielu systemach NoSQL takich jak: HBase [21], MongoDB [13], LevelDB [22], Cassandra [52], RockDB [18]. Systemy te są z kolei wykorzystywane przez: Facebook, LinkedIn, GoldenLine, Instagram, Twitter oraz wiele innych popularnych serwisów. Prace nad drzewem LSM to obecnie bardzo aktywny obszar badań naukowych: [17], [67], [10], [16], [31], [44], [64], [28].

Drzewo LSM jest zaprojektowane tak, aby efektywnie zarządzać przechowywaniem i odczytem danych szczególnie w sytuacjach wymagających przetwarzania dużych wolumenów danych. Dane przechowywane na drzewie LSM są z reguły reprezentowane jako elementy klucz-wartość (*key*, *val*) i są składowane w oddzielnych plikach zwanych tablicą SST (ang. Sorted String Tables). Drzewo LSM składa się z k poziomów (L_0 do L_k) (zobacz rys. 4), przy czym elementy (*key*, *val*) na poszczególnych poziomach mają unikalne wartości klucza (*key*) i są posortowane po *key*. Każdy poziom obejmuje określoną liczbę plików (tablic SST), co skutkuje limitem rozmiaru danych przypisanym do każdego poziomu. Przyjmuje się, że ilość danych na poziomie L_{k+1} jest c razy większa niż ilość danych na poziomie L_k (c jest zdefiniowaną stałą). Nowe elementy mogą być dodawane tylko do najwyższego poziomu L_0 zwanego MemTable, który jest przechowywany w pamięci RAM. Gdy liczba elementów na poziome L_0 przekroczy określony limit, dane są wpisywane na poziom L_1, który jest zapisywany na dysk. Generalnie, jeśli poziom L_k przekroczy swój limit, dane z tego poziomu są scalane z poziomem L_{k+1}, cały czas zachowując uporządkowanie po kluczu. Modyfikacja elementu (*key*, *val*) po kluczu polega na wstawianiu nowej wersji elementu (*key*, val_1), a usuwanie elementu (*key*, *val*) polega na wstawieniu elementu (*key*, *d*), przy czy *d* oznacza *deleted*. Wyszukiwanie elementu (*key*, *val*) rozpoczyna się od wyszukiwania na poziomie L_0. Gdy klucz *key* zostanie znaleziony, wartość *val* zostanie zwrócona. Jeśli klucz *key* nie zostanie znaleziony, wówczas przeszukiwane są kolejno następne poziomy L_1, L_2, itd. Warto zauważyć, że jeśli element o kluczu *key* został zmodyfikowany, wówczas najnowsza wersja będzie "najwyżej" w drzewie i to ona zostanie zwrócona. Gdy natomiast element został usunięty, wówczas jego wersja (*key*, *d*) zostanie znaleziona i użytkownik otrzyma informację o braku takiego elementu. Wyszukiwanie w drzewie LSM jest bardzo szybkie, gdyż elementy na poszczególnych poziomach są posortowane po *key* oraz każda tablica SST posiada dodatkowo zakres kluczy elementów w niej przechowywanych. Tak więc przeszukiwanie opiera się na wyszukiwaniu binarnym tylko w obrębie takiej tablicy SST, do której zakresu należy wyszukiwana wartość klucza *key*. Mamy więc do czynienia z logarytmiczną złożonością czasową przy wyszukiwaniu po wartości *key*.

Często okazuje się jednak, że indeksowanie po *key* nie jest wystarczające w tego typu bazach danych. Rozważmy bazę danych serwisu Twitter. Każdy tweet jest reprezentowany jako $e =< id, \{user, text\} >$ co oznacza, że użytkownik *user* opublikował opublikował tweet o treści *text* z przypisanym identyfikatorem *id*. Niech *id* oznacza klucz wpisu e, a *user* i *text* są wartościami, które



Rysunek 4: Drzewo LSM.

nie są kluczami. Jak to pokazaliśmy wcześniej wyszukiwanie tweeta e o określonym id jest bardzo proste i szybkie. Gdybyśmy chcieli jednak wyszukać wszystkie tweety użytkownika "Kowalski" ($user$='Kowalski') musielibyśmy przeszukać całą bazę danych, a więc przejść po wszystkich tablicach SST i zwrócić te dane, które są tweetami Kowalskiego. W takim przypadku stworzenie indeksu wtórnego na $user$ znacznie przyspieszy wyszukiwanie danych danego użytkownika. Problem indeksów wtórnych dla baz danych opartych o strukturę LSM jest poruszany w następujących pracach: [82], [89], [57]. Istnieją dwie główne koncepcje implementacji takich indeksów. Pierwsza koncepcja zakłada zintegrowanie nowej struktury indeksowej z tablicą SST w bazie danych (ang. embedded secondary index). Druga koncepcja zakłada stworzenie osobnej struktury indeksowej stojącej niejako "obok" bazy danych (ang. stand-alone secondary index). Porównanie wad i zalet obu koncepcji oraz przegląd literatury dotyczącej tego tematu można znaleźć w pracach [83] oraz [65].

W dalszej części tego rozdziału opisane są nasze prace dotyczące indeksu wtórnego (ang. secondary index) w bazach danych klucz-wartość (ang. key-value) opartych o drzewo LSM. Rozdział składa się z następujących części. Pierwsza część (podrozdział 4.3.1) opisuje metodę ułatwiającą ładowanie wielu elementów indeksu wtórnego na raz do drzewa LSM (ang. bulklaoding). Metoda ta jest zoptymalizowana dla pamięci flash. W części drugiej (podrozdział 4.3.2) zaproponowano metodę częściowego indeksowania baz danych typu klucz-wartość opartych o drzewo LSM. Wreszcie część ostatnia (podrozdział 4.3.3) wprowadza hierarchiczne filtry Blooma jako alternatywną metodę indeksowania w drzewie LSM.

4.3.1 Ładowanie elementów indeksu wtórnego do drzewa LSM.

W pracy [A8] zaproponowaliśmy metodę ułatwiającą ładowanie wielu elementów indeksu wtórnego na raz do drzewa LSM (ang. bulklaoding) znajdującego się na pamięci flash, na którym to drzewie oparty jest indeks wtórny. Zauważyliśmy bowiem, że często występuje sytuacja, w której indeks wtórny (ang. secondary index) jest efektem ubocznym zapytania do bazy danych, którego wynikiem jest duży zbiór danych. Każdy element indeksu wtórnego ma postać (key, ptr), gdzie key oznacza

klucz wtórny (ang. secondary key), a *ptr* jest wskaźnikiem do elementu indeksowanego znajdującego się w głównej bazie danych. Przypomnijmy że zwykłe dodawanie elementu w drzewie LSM polega na wstawieniu go do najwyższego poziomu. Gdy liczba elementów poziomu zostanie przekroczona, poziom ten jest łączony z poziomem niżej, itd. Proces ten został już opisany powyżej. Taka metoda nie jest jednak efektywna w przypadku, gdy wiele elementów jest wstawianych na raz, gdyż operacja łączenia poziomów będzie wykonywana bardzo często, a to jest procesem kosztownym na pamięci flash. Zaproponowane w pracy podejście polega na tym, że zbiór danych jest wstawiany do takiego najwyższego poziomu drzewa LSM, który jest w stanie go zmieścić w całości. W tym celu zaproponowaliśmy dwa typy tablic SST: normalny (ang. normal SSTable) oraz przepełniony (ang. overflow SSTable). W naszym podejściu zakładamy, że tablica normalna przechowuje zakres danych, który nie nakłada się z zakresem danych innej tablicy normalnej SST na tym samym poziomie. Z kolei w tablicy przepełnionej elementy są posortowane po wartości klucza, ale ich zakres może pokrywać się z zakresem innych tablic SST na tym samym poziomie. Dzięki temu wyszukiwanie elementu na danym poziomie jest wprawdzie nieznacznie dłuższe, ale unikamy kosztownego dla pamięci flash łączenia poziomów. Efektywność działania metody potwierdziliśmy w sposób eksperymentalny dla różnego zestawu danych oraz różnego zakresu zapytań.

4.3.2 Częściowe indeksowanie indeksu wtórnego na drzewie LSM.

W pracy [A9] opracowaliśmy metodę częściowego indeksowania dla baz danych opartych o drzewo LSM. Zalety częściowego indeksowania zostały już opisane w poprzednim podrozdziale. Tym razem skupiliśmy się na wykorzystaniu architektury drzewa LSM, a w szczególności na tym, że dane są przechowywane w wielu niewielkich plikach. Implementację oparliśmy o bazę danych LevelDB [22] z kilku powodów. Po pierwsze, baza ta jest dojrzałym produktem firmy Google, który jest używany jako warstwa przechowywania danych w bazie danych RockDB [18], która z kolei wykorzystywana w produktach firm: Facebook, Uber i LinkedIn. Kolejne powody to duża szybkość działania tej bazy, prostota jej implementacji oraz wsparcie wielowątkowości.

Nasza metoda działa w sposób następujący. Na początku wszystkie elementy (*key*, *val*) znajdujące się w każdej tablicy SST są kopiowane do osobnych plików logu adaptacyjnego (zobacz rysunek 5). Jako że tablice SST są niezależne, można wykorzystać wielowątkowość do procesu kopiowania, co znacznie przyspiesza całą operację. Elementy są zapisywane do logu adaptacyjnego w sposób posortowany ale nie po kluczu (*key*), lecz po wartości (*val*). I tak, jeśli tablica SST zawiera elementy (*key*, *val*): $< 6, g >, < 9, i >$, wówczas plik logu adaptacyjnego powstały w wyniku kopiowania tej tablicy będzie wyglądał w sposób następujący: $< g, 6 >, < i, 9 >$. Każdy plik logu adaptacyjnego zawiera zakres wartości *val*. W tym przypadku będzie to przedział $< g, i >$. Gdy system dostanie zapytanie o zakres wartości $< e, p >$ (zobacz krok 1 na rysunku 5), metoda skanuje wszystkie takie logi, których zakres wartości przecina żądany przez zapytanie zakres ($< e, p >$). Następnie, elementy będące w zakresie $< e, p >$ są pobierane z odpowiednich logów i są zapisywane do indeksu. Na rysunku 5 elementy pobrane z logu są zaznaczone szarym wypełnieniem. Stosujemy tutaj usuwanie leniwe polegające na tym, że elementy pobrane z logu nie są natychmiast z niego usuwane, ale są oznaczone jako "pobrane". Dla ułatwienia każdy plik logu adaptacyjnego posiada mapę bitową, która wskazuje, które elementy zostały już pobrane. Po zakończeniu wykonywania zapytania zapisywany jest nowy zakres indeksu ($< e, p >$), który wskazuje jakie elementy są już w indeksie. Tak więc jeśli kolejne zapytanie będzie dotyczyło podprzedziału $< e, p >$, wówczas będzie wiadomo,

że należy pobrać wynik zapytania z indeksu. Jeśli kolejne zapytanie będzie rozłączne z zakresem indeksu, to należy przeskanować logi w celu znalezienia wyniku zapytania. Oczywiście może być też sytuacja, że będzie zapytanie o zakres $< a, j >$ (zobacz krok 2 na rysunku 5). W takim przypadku część odpowiedzi będzie pobrana z logów a cześć z indeksu. Nasze podejście rozpatruje również sytuację, w której główna baza LSM ulegnie zmianie (np. zostanie wykonana operacja usuwania lub modyfikacji). Poprawność koncepcji została udowodniona eksperymentalnie. Eksperymenty porównują zaproponowaną metodę z metodą wykorzystującą indeksowanie całkowite (ang. full index) na drzewie LSM oraz z metodą bez indeksowania w zależności od ilości danych, zakresu selektywności zapytań oraz liczby wątków.

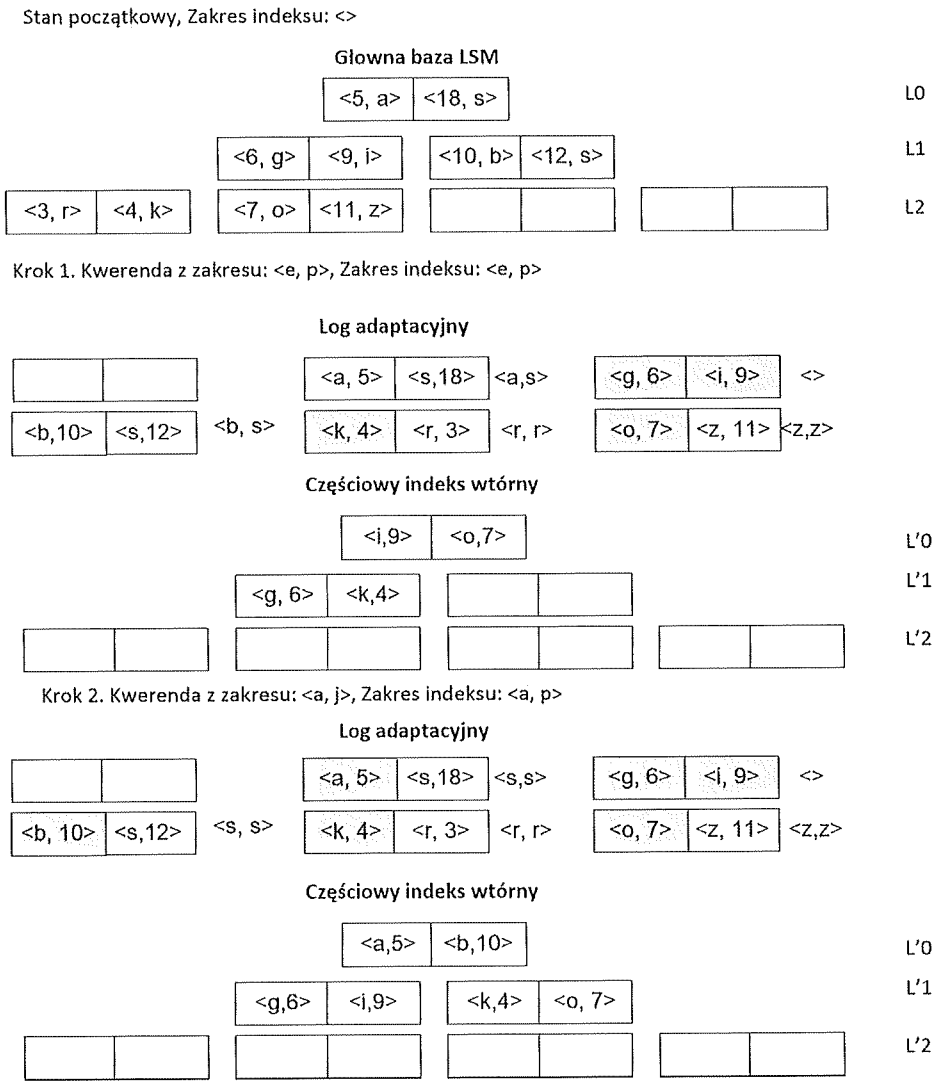
4.3.3 Hierarchiczne filtry Blooma.

W pracy [A10] wykorzystujemy hierarchię filtrów Blooma do szybkiego wyszukiwania wartości nie będących kluczem w bazach danych klucz-wartość opartych o drzewo LSM. Dzięki tej technice łatwo można zidentyfikować takie tablice SST, w których mogą się znajdować żądane wartości i tylko te tablice przeszukiwać. Podejście jest więc alternatywą dla indeksu wtórnego, gdyż nie trzeba tworzyć osobnej struktury indeksowej.

Filtr Bloom [5] to struktura danych służąca do szacowania czy element należy do zbioru. Niech S będzie zbiorem n elementów: $S = \{s_1, s_2, ..., s_n\}$. Filtr Blooma to tablica bitów $T[0, ..., m-1]$ początkowo ustawionych na 0. Filtr Blooma używa k niezależnych funkcji haszujących: $h_1, h_2, ..., h_k$ o zakresie $\{0...m-1\}$. Dla wstawianego elementu $s \in S$ wykonywana jest operacja $T[h_i(s)] = 1$ przy użyciu każdej funkcji haszującej h_i (gdzie: $1 \leqslant i \leqslant k$). Zweryfikowanie czy $x \in S$ polega na sprawdzeniu, czy $T[h_i(x)]$ jest równe 1 dla każdej funkcji haszującej h_i. Jeśli przynajmniej jeden z bitów $T[h_i(x)]$ jest równy 0, wówczas $x \notin S$. W przeciwnym razie zakładamy że $x \in S$, chociaż założenie to może być błędne z małym prawdopodobieństwem oszacowanym jako: $p_{\text{false}} \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k$.

Hierarchia filtrów Blooma (zobacz [14]) jest strukturą drzewiastą podobną do B+-drzewa. Filtry Blooma na poziomie liści są strukturami, które umożliwiają efektywne wyszukiwanie elementów znajdujących się w zbiorach danych, które są z tymi filtrami powiązane. Z kolei węzły nie będące liśćmi uzyskuje się poprzez zastosowanie operacji bitowej "OR" na ich potomkach (zobacz rysunek 6). Każdy filtr Blooma w hierarchii musi mieć taką samą liczbę bitów m oraz identyczne funkcje haszujące $h_1, h_2, ..., h_k$. Parametr rzędu d definiuje strukturę drzewa, gdzie każdy węzeł który nie jest liściem ma f wskaźników do swoich potomków. Zachodzą przy tym następujące zależności: (1) $d \leqslant f \leqslant 2d$ dla węzłów, które nie są liśćmi oraz (2) $2 \leqslant f \leqslant 2d$ dla korzenia drzewa. Rysunek 6 przedstawia hierarchię filtra Blooma z czterema liśćmi, dwoma węzłami pośrednimi i jednym korzeniem. Rozmiar m każdego filtra Blooma wynosi 4, a rząd d wynosi 2. Szacowana głębokość hierarchii to $\log_d B$, gdzie B to liczba filtrów na poziomie liści.

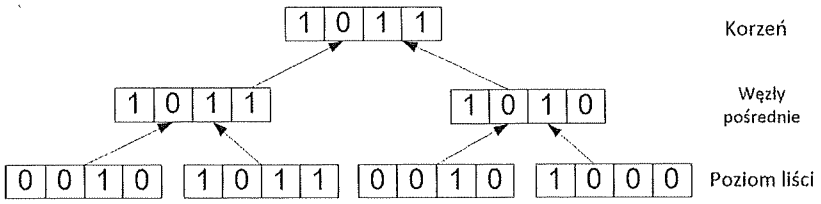
Kluczową cechą tego hierarchii jest to, że wartości bitów na węźle pośrednim filtra Blooma są sumą wartości bitowych filtrów Blooma w poddrzewie zakorzenionym w tym węźle. W związku z tym, jeśli filtr Blooma na liściu hierarchii wskazuje, że szukany obiekt może znajdować się w zbiorze, wówczas wszystkie filtry Blooma od liścia do korzenia muszą wskazywać tak samo. Z kolei, jeśli filtr Blooma na wierzchołku pośrednim wskazuje, że szukany obiekt nie znajduje się w zbiorze, wówczas wszystkie filtry Blooma w poddrzewie, którego korzeniem jest ten wierzchołek także muszą wskazywać, że szukanego obiektu tam nie ma. Dzięki temu nie ma potrzeby przeszukiwania tego



Rysunek 5: Częściowe indeksowanie na drzewie LSM.

poddrzewa. W szczególności może zajść sytuacja, że filtr Blooma na korzeniu hierarchii wskazuje na brak szukanego elementu w całej hierarchii. Wówczas od razu wiadomo, że dany element nie istnieje w żadnym zbiorze danych.

W pracy [A10] wykorzystujemy hierarchię filtrów Blooma do szybkiego wyszukiwania wartości *val* w bazach danych przechowujących elementy klucz-wartość (*key*, *val*) opartych na drzewie LSM. Jak to zostało wcześniej wspomniane, tablice SST z poziomów L_1 do L_k w drzewie LSM zmieniają się rzadko, jedynie w wyniku łączenia poziomów. Jako że klasyczne filtry Blooma nie wspierają usuwania i modyfikowania elementów (jedynie wstawianie), dla każdej tablicy SST w czasie jej tworzenia



Rysunek 6: Hierarchia filtrów Blooma (m=4, d=2).

dodajemy osobny filtr Blooma. Tak zdefiniowany zbiór filtrów Blooma tworzy poziom liści hierarchii filtrów Blooma, na podstawie którego tworzona jest cała hierarchia. W pracy opisaliśmy integrację hierarchii filtrów Blooma z drzewem LSM. Opracowaliśmy algorytmy wyszukiwania wartości nieposortowanych (*val*) oraz tworzenia hierarchii filtrów Blooma przy łączeniu poziomów drzewa LSM. Dodatkowo opisaliśmy jaki wpływ na hierarchię filtrów Blooma mają operacje modyfikowania oraz usuwania elementów.

Hierarchię filtrów Blooma zaimplementowaliśmy w bazie LevelDB i przeprowadziliśmy szereg eksperymentów z różnymi wartościami parametrów takich jak: rozmiar filtrów Blooma, szerokość hierarchii filtrów Blooma, rozmiar dziedziny danych, rozmiar powtarzających się danych, itd.. W wyniku tych eksperymentów otrzymaliśmy potwierdzenie logarytmicznego czasu przeszukiwania wartości nieposortowanych w bazie danych. Dzięki temu czas przeszukiwania jest nawet o 80% krótszy od czasu przeszukiwania bez użycia proponowanej metody. Na uwagę zasługuje również fakt, że metoda ta może zostać użyta wszędzie tam, gdzie chcemy wykonać przeszukiwanie w nieuporządkowanym zbiorze danych przechowywanym w wielu plikach.

4.3.4 Wkład do dyscypliny.

Na podstawie wyżej przedstawionych prac można zidentyfikować następujący wkład do dyscypliny:

- Opracowanie metody dodawania wielu elementów indeksu wtórnego jednocześnie do drzewa LSM (praca [A8]). Metoda jest zoptymalizowana pod kątem pamięci flash. Efektywność działania metody potwierdziliśmy w sposób eksperymentalny dla różnego zestawu danych oraz różnego zakresy zapytań.

- Opracowanie koncepcji indeksu częściowego dla baz danych opartych na drzewie LSM (praca [A9]). Poprawność koncepcji została udowodniona eksperymentalnie. Eksperymenty porównują zaproponowaną metodę z (1) użyciem w bazie danych indeksowania całkowitego (ang. full index) oraz z (2) brakiem indeksu w zależności od ilości danych, zakresu selektywności zapytań oraz liczby wątków. Jest to pierwsza implementacja indeksowania częściowego, gdzie indeks wtórny opiera się na LSM drzewie.

- Zaproponowanie hierarchicznych filtrów Blooma do wyszukiwania elementów niekluczowych w bazach danych opartych o drzewo LSM (praca [A10]). Dzięki hierarchicznym filtrom Blooma zintegrowanym z drzewem LSM przeszukiwanie wartości niekluczowych na drzewie LSM

skraca się z $O(n)$ na $O(\log n)$ bez potrzeby zastosowania osobne struktury indeksu wtórnego. Eksperymenty przeprowadzone z użyciem bazy LevelDB pokazują, że czas przeszukiwania jest nawet o 80% krótszy od czasu przeszukiwania bez użycia proponowanej metody. Metoda ta może zostać użyta wszędzie tam, gdzie chcemy wykonać przeszukiwanie w nieuporządkowanym zbiorze danych przechowywanym w wielu plikach.

4.4 Indeksowanie na pamięci PCM

W rozdziale 4.2 rozpatrywaliśmy pamięć flash, która była pamięcią adresowaną blokowo. W ostatnich kilku latach coraz większe znaczenie mają pamięci adresowana bajtowo. Przykładem takich technologii mogą być pamięci PCM oraz wykorzstująca tę pamięć technologi 3D-Xpoint (zobacz [68]). Pamięć PCM opiera się na nośniku krystalicznym, który wykorzystuje zmiany stanu do przechowywania danych. W temperaturze pokojowej nośnik może występować w stanie amorficznym (odpowiadającym logicznemu 0) lub krystalicznym (logiczne 1). Przełączanie między stanami odbywa się poprzez podgrzewanie nośnika za pomocą wiązki elektronów. Odczyt danych również opiera się na działaniu strumienia elektronów i polega na pomiarze rezystancji nośnika różnej dla obu faz. Tak jak jest to w przypadku pamięci RAM, pamięć PCM jest bajtowo adresowalna. Jednak ze względu na jej budowę oraz lepszą współpracę z pamięcią podręczna procesora, została podzielona na strony o wielkości 64B. Typowa kość pamięci składa się z 4 lub 8 banków, przy czym obecnie najczęściej stosuje się konfigurację z 8 bankami. Umożliwia to odczyt całej strony pamięci PCM (64B) podczas jednej operacji, ponieważ z każdego banku odczytywane jest jedno słowo maszynowe (8B) w architekturze 64-bitowej, która obecnie dominuje na rynku. Taka organizacja pozwala na optymalne wykorzystanie zarówno pamięci PCM, jak i pamięci podręcznej procesora, ponieważ linia pamięci podręcznej (ang. cache line) również zazwyczaj ma rozmiar 64B. Dzięki temu pojedyncza operacja odczytu lub zapisu obejmuje całą linię pamięci podręcznej, co maksymalizuje przepustowość i minimalizuje opóźnienia między pamięcią a procesorem. Pamięć ta może zostać wykorzystana nie tylko jako pamięć masowa, ale także jako pamięć operacyjna komputera (zobacz: [80]). Niestety podobnie jak w przypadku pamięci flash, mimo bardzo szybkiego odczytu, pamięć ta charakteryzuje się wolniejszym (nawet 10 razy) zapisem [68]. Kolejnym problemem jest limit zapisów pojedynczej komórki wynoszący od 10^7 do 10^8. Mimo iż 10 mln może się wydawać limitem trudnym do osiągnięcia, to jednak komputer używający pamięci PCM jako pamięci operacyjnej, mógłby zniszczyć pojedyncze komórki pamięci w zaledwie kilka minut. Jako rozwiązanie tego problemu w pracach [74], [71], [87] zaproponowano kilka algorytmów mapujących linie pamięci PCM (64B) oraz logiczne strony (4KB) w podobny sposób jak algorytm FTL mapujący strony dysków SSD. Porównanie najważniejszych metod mapowania pamięci można znaleźć w pracy [75]. Ze względu na specyficzne właściwości pamięci PCM, indeksowanie na takiej pamięci wymaga utworzenia nowych struktur danych. Istnieje wiele prac poświęconych dostosowywaniu struktur B+-drzewa oraz R-drzewa do pamięci PCM: [11], [60], [12], [59], [36], [37].

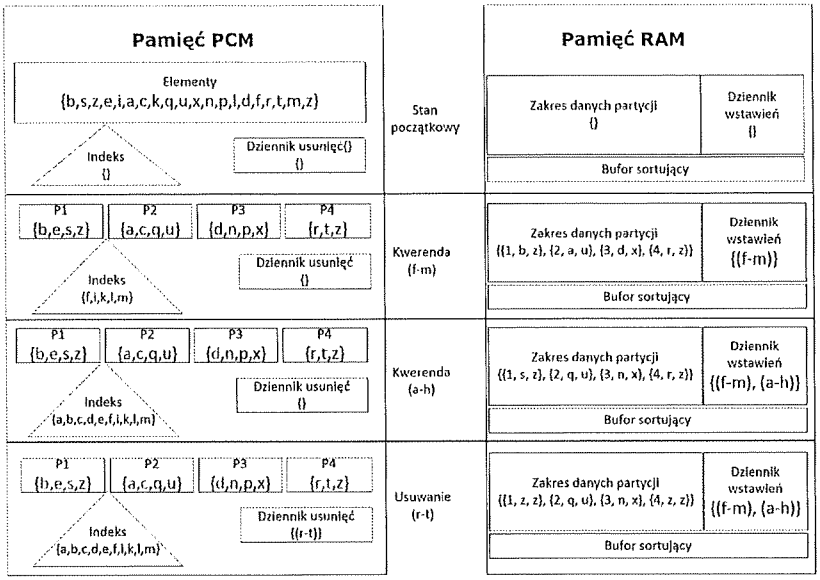
Podrozdział składa się z dwóch części. Pierwsza cześć (podrozdział 4.4.1) opisuje implementację zagregowanych R-drzew na pamięci PCM. W części drugiej (podrozdział 4.4.2) zaproponowaliśmy częściowe indeksowanie zoptymalizowane dla pamięci PCM.

4.4.1 Przechowywanie wartości zagregowanych w R-drzewie.

Podobnie jak [A2], praca [A11] również rozwiązuje problem wartości zagregowanych w R-drzewie. Tym razem jednak koncentrujemy się na pamięci PCM, a nie pamięci pamięci flash. W związku z tym rozszerzyliśmy metodę przechowywania R-drzewa na pamięci PCM przedstawioną w [36] o przechowywanie wartości zagregowanych. Zaproponowana metoda wykorzystuje zagregowany bufor (ang. aggregated cache), w którym te wartości są przechowywane. Zakładamy, że bufor ma ograniczoną pojemność oraz że składa się z $H-1$ partycji, przy czym H jest wysokością R-drzewa. Z każdą partycją powiązany jest zakres kosztów określający jakie elementy (o jakim koszcie) mogą być przechowywane w tych partycjach. Na początku, gdy bufor jest pusty, zakres kosztów jest powiązany z poziomem R-drzewa. Tak więc zagregowane wartości z poziomu drzewa L_i są przypisane do partycji P_i. Wyjątkiem jest poziom liści, który nie przechowuje wartości zagregowanych oraz korzeń drzewa, który jest przypisany do poziomu poniżej. Zakres kosztów dla poszczególnej partycji może być więc oszacowany na podstawie ilości wierzchołków dzieci oraz kosztu odczytu pojedynczej wartości. Gdy dostajemy zapytanie o wartość zagregowaną związaną z wierzchołkiem R, system ją oblicza na podstawie dzieci tego wierzchołka oraz oszacowuje koszt tej kalkulacji. Następnie na podstawie oszacowanego kosztu wartość ta jest wpisywana lub nie do odpowiedniej partycji w buforze. Warto tutaj wspomnieć, że koszt wartości zagregowanej wierzchołków na wyższych poziomach hierarchii może być niski, jeśli wartości zagregowane związane z jego dziećmi są już w buforze. W takim przypadku wartość ta wpadnie do partycji o niższym zakresie. Metoda jest tak skonstruowana, że jeśli rozmiar bufora zostanie przekroczony, wówczas usuwane są wartości z partycji o najmniejszym zakresie kosztów. Przeprowadzono eksperymenty porównujące zaproponowaną metodę do metody FIFO w zależności od wielkości bufora, rodzaju zapytań oraz poziomu R-drzewa, dla którego są odpytywane zagregowane wartości. Eksperymenty pokazały, że nasza metoda wykonuje od 2 do 4 razy mniej operacji zapisu niż metoda FIFO.

4.4.2 Indeksowanie na pamięciach typu PCM

W pracy [A12] zaproponowaliśmy koncepcję częściowego indeksowania na pamięci PCM. Metoda umożliwia indeksowanie tylko takich danych, które są wynikiem działania zapytań o zakres danych (ang. range queries). Zasada działania jest zobrazowana na rysunku 7. Na początku dane na pamięci PCM nie są posortowane. Pierwsza kwerenda (zapytanie o zakres danych $< f-m >$) uruchamia częściowe indeksowanie. Elementy będące w zakresie $< f-m >$ są dodawane do indeksu. Pozostałe dane są umieszczone w pamięci PCM w posortowanych partycjach $P1$, $P2$, $P3$ i $P4$. Z każdą partycją powiązany jest zakres danych partycji zapisany do pamięci RAM. Dzięki temu łatwo jest zidentyfikować, która partycja powinna być brana pod uwagę podczas wyszukiwania danych. Dodatkowo, zakres wyszukiwania $< f-m >$ jest wpisywany do dziennika zapisów. Podczas kolejnego zapytania dane są pobierane albo z indeksu (jeśli zostały już zaindeksowane przez wcześniejsze zapytania) albo z partycji (w przeciwnym razie). W przypadku zapytania o zakres $< a-h >$ przeszukiwane są dane z indeksu oraz z partycji $P1$, $P2$ oraz $P3$. Wynika to z faktu, że zakres danych tych partycji przecina się z szukanym zakresem $< a-h >$. Wyszukane w ten sposób dane są wstawiane do indeksu. Nie są one jednak usuwane z partycji, tylko są oznaczane jako "zaindeksowane" (kolor czerwony). Technicznie jest to zrobione dzięki wykorzystaniu mapy bitowej wskazującej, które elementy partycji są już zaindeksowane a które jeszcze nie. Po wykonaniu kwerendy możliwa jest



Rysunek 7: Częściowe indeksowanie na PCM

zmiana zakresu danych w poszczególnych partycjach. Dodatkowo, zakres $< a - h >$ jest wpisywany do dziennika zapisów. Cechą charakterystyczną zaproponowanego rozwiązania jest możliwość zmiany zbioru indeksowanego. Koncepcja uwzględnia możliwość dodania oraz usunięcia elementu z bazy danych. W przypadku gdy coś zostanie usunięte z bazy danych, informacja ta jest zapisywania w dzienniku usunięć. Przykładowo po usunięciu danych o zakresie $< r - t >$ z bazy danych, fakt ten będzie zapisany w dzienniku usunięć. Wówczas kwerenda $< s - t >$ nie pobierze żadnych danych z partycji pomimo tego, że one tam są przechowywane.

Kolejną cechą nowatorską zaproponowaną w pracy jest stworzenie nowego indeksu. Mimo że nasza koncepcja jest generyczna i można w niej używać dowolnego indeksu, to jednak postanowiliśmy zaproponować nowy indeks zoptymalizowany dla pamięci PCM. W tym celu zmodyfikowaliśmy indeks DPTree zaproponowany w pracy [91] i opracowaliśmy nowy indeks BB+ drzewo (ang. BB+-tree). Zauważyliśmy bowiem, że w wyniku zapytania o zakres (ang. range query) do indeksu wstawiany jest zbiór posortowanych kluczy. Z drugiej strony bezpośrednie modyfikacje na bazie danych mogą dodawać do indeksu nowe elementy w sposób nieposortowany. W takim przypadku lepiej jest wstawiać takie wpisy na końcu węzła w sposób nieposortowany zamiast sortować cały węzeł. Aby to osiągnąć, podzieliliśmy węzeł na dwie sekcje: posortowaną i nieposortowaną. Żeby znaleźć wartość indeksu w sekcji posortowanej, wykorzystujemy wyszukiwanie binarne. Dodatkowo każdy węzeł nowej struktury indeksowej posiada mapę bitową, która wskazuje, czy element jest aktualny, czy nie. Gdy element jest usuwany z węzła, nie jest on natychmiast usuwany z PCM, lecz oznaczany jako nieaktualny w bitmapie. Rozmiar obu sekcji jest wyrównany do wielokrotności rozmiaru linii pamięci podręcznej (ang. cacheline). W ten sposób zmniejsza się liczba nietrafień w pamięci cache (ang. cache misses) podczas wyszukiwania liniowego. Dodatkowo, węzły wewnętrzne (ang. internal nodes) są buforowane w pamięci RAM, co optymalizuje reorganizację drzewa.

Wszystkie eksperymenty przedstawione w pracy zostały zrealizowane na autorskim symulatorze pamięci PCM. Pierwsza część eksperymentów zawiera porównanie indeksów zoptymalizowanych pod PCM w kontekście zaproponowanej koncepcji. Porównaliśmy nasz indeks (BB+-drzewo) do indeksu UB+drzewo z [11], który jest B+drzewem z nieposortowanymi danymi w liściach oraz do indeksu DPTree z pracy [91]. Eksperymenty potwierdzają przydatność zaproponowanego indeksu BB+-drzewo. Warto tutaj nadmienić, że DPTree był najlepszym w tamtym czasie indeksem zaproponowanym dla pamięci PCM. Jeśli pojawi się lepsza struktura indeksowa, wówczas będzie ona mogła być zintegrowana z naszym koncepcją. W drugiej części eksperymentów porównujemy nasze podejście z oryginalną metodą częściowego indeksowania oraz z metodą częściowego indeksowania zoptymalizowaną dla pamięć PCM. Część trzecia eksperymentów sprawdza częściowe indeksowanie w przypadku, gdy zawartość bazy danych będzie się zmieniać podczas częściowego indeksowania.

4.4.3 Wkład do dyscypliny.

Na podstawie wyżej przedstawionych prac można zidentyfikować następujący wkład do dyscypliny:

- Zaproponowano metodę przechowywania danych zagregowanych związanych z węzłami indeksu przestrzennego (R-drzewa) przy ograniczonej pamięci PCM (praca [A11]). Jako, że jest to pierwsza taka koncepcja, porównaliśmy ją eksperymentalnie do metody (1) nie przechowującej wartości zagregowane oraz (2) do metody, która przechowuje te wartości w buforze, ale usuwa je przy przepełnieniu bufora zgodnie z zasadą FIFO (bez uwzględnienia kosztów związanych z dostępem do pamięci PCM).

- W pracy [A12] zaproponowano koncepcję do częściowego indeksowania na pamięci PCM. Jest to pierwsza tego typu implementacja. Porównaliśmy eksperymentalnie nasze rozwiązanie z oryginalną metodą indeksowania częściowego oraz oryginalną metodą indeksowania częściowego zoptymalizowaną dla pamięci PCM. Nasze podejście można traktować jako framework, który może współpracować z jakąkolwiek strukturą indeksową.

- Zaproponowano nowy indeks (BB+-drzewo) przeznaczony do efektywnego indeksowania danych pochodzących z zapytań o zakres danych (ang. range queries) zoptymalizowany dla pamięci PCM ([A12]). Potwierdziliśmy eksperymentalnie, że indeks ten jest wydajniejszy w przypadku częściowego indeksowania od indeksów przedstawionych w pracach [11] oraz [91].

4.5 Inne prace autora

Niniejszy rozdział opisuje inne ważniejsze prace, których jestem współautorem. Większość z nich powstało w wyniku współpracy międzynarodowej z Houston University of Technology i ISAE ENSMA w Poitiers. Poniżej zaprezentowany jest krótki opis każdej z tych prac.

[B1] Ladjel Bellatreche, Fouad Djellali, Wojciech Macyna, Carlos Ordonez. Energy-aware query processing: A case study on join reordering. *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*, strony 3743–3752. IEEE, 2023.

[B2] Łukasz Krzywiecki, Krzysztof Majcher, Wojciech Macyna. Efficient probabilistic methods for proof of possession in clouds. Ying Tan, Yuhui Shi, redaktorzy, *Data Mining and Big Data,*

First International Conference, DMBD 2016, Bali, Indonesia, June 25-30, 2016. Proceedings, wolumen 9714 serii *Lecture Notes in Computer Science*, strony 364–372. Springer, 2016.

[B3] Carlos Ordonez, Wojciech Macyna. Optimizing energy consumed by analytics in the cloud. *2024 IEEE International Conference on Big Data (BigData)*, strony 5201–5210, 2024.

[B4] Carlos Ordonez, Wojciech Macyna, Ladjel Bellatreche. Data engineering and modeling for artificial intelligence. *Data Knowl. Eng.*, 153:102346, 2024.

[B5] Carlos Ordonez, Wojciech Macyna, Ladjel Bellatreche. Energy-aware analytics in the cloud. Philippe Cudré-Mauroux, Andrea Kö, Robert Wrembel, redaktorzy, *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE 2024, Santiago, Chile, June 9-15, 2024*, strony 3:1–3:6. ACM, 2024.

[B6] Carlos Ordonez, Robin Varghese, Nguyen Phan, Wojciech Macyna. Growing a FLOWER: building a diagram unifying flow and ER notation for data science. Jean-Daniel Fekete, Behrooz Omidvar-Tehrani, Kexin Rong, Roee Shraga, redaktorzy, *Proceedings of the 2024 Workshop on Human-In-the-Loop Data Analytics, HILDA 24, Santiago, Chile, 14 June 2024*, strony 1–8. ACM, 2024.

Prace [B1], [B3] oraz [B5] podejmują ważną problematykę zużycia energii w nowoczesnych systemach informatycznych. W pracy [B1] koncentrujemy się na badaniu zużycia energii w systemach bazodanowych. W dużych systemach relacyjnych baz danych (np. hurtowniach danych) często występuje potrzeba łączenia (ang. join) wielu tabel bazodanowych w jednym zapytaniu. Kolejność łączenia tabel jest wybierana przez system zarządzania bazą danych w sposób zoptymalizowany pod kątem szybkości wykonywania zapytania. Taka strategia nie uwzględnia kosztów energetycznych takiego zapytania. Okazuje się bowiem, że łączenie tabel w innej kolejności może nieznacznie zmniejszyć prędkość wykonywania zapytania, ale znacznie zmniejszyć koszt energetyczny. W pracy [B1] zweryfikowano różne techniki optymalizacji łączenia tabel w relacyjnych bazach danych pod kątem efektywności zapytań oraz zużycia energii. Kolejne prace [B5] oraz [B3] skupiają się na analizie energii zużywanej przez centra obliczeniowe. Głównym celem tych prac jest określenie jakie komponenty sprzętowe (hardware) oraz systemowe (software) zużywają najwięcej energii w chmurach obliczeniowych. Na tej podstawie zidentyfikowaliśmy problemy oraz kierunki badań mające na celu zmniejszenie zapotrzebowania energetycznego, a tym samym zmniejszenie emisji gazów cieplarnianych emitowanych przez tego rodzaju infrastrukturę.

W pracy [B2] również zajmujemy się problemami w dużych centrach obliczeniowych. Tym razem proponujemy metodę weryfikacji poprawności przechowywanych danych na zdalnych serwerach. Zamiast skomplikowanych protokołów kryptograficznych, nasze podejście stosuje funkcje haszujące. Dzięki temu można je stosować w przypadkach, gdzie skomplikowane obliczenia nie są możliwe do wykonania.

Wreszcie w pracy [B5] proponujemy rozszerzenie diagramów ER (ang. Entity-Relationship diagrams) do lepszej obsługi procesów obsługujących przetwarzanie danych. W wyniku pracy powstał typ diagram UML o nazwie FLOWER, który integruje źródła danych z procesami na nich zachodzącymi. Tego typu diagramy mają ogromne zastosowanie do wizualizacji procesów zachodzących na przykładach w projektach uczenia maszynowego, gdzie liczba zbiorów danych i procesów na nich zachodzących jest duża i skomplikowana do zrozumienia.

5 INFORMACJA O WYKAZYWANIU SIĘ ISTOTNĄ AKTYWNOŚCIĄ NAUKOWĄ ALBO ARTYSTYCZNĄ REALIZOWANĄ W WIĘCEJ NIŻ JEDNEJ UCZELNI, INSTYTUCJI NAUKOWEJ LUB INSTYTUCJI KULTURY, W SZCZEGÓLNOŚCI ZAGRANICZNEJ

5.1 Realizowane projekty

Podczas pracy w firmie Comarch do moich obowiązków należało rozwijanie współpracy z uczelniami oraz pozyskiwanie dotacji na projekty. Celem było stworzenie innowacyjnych i konkurencyjnych produktów z wykorzystaniem potencjału uczelni wyższych. Udało się uzyskać dofinansowanie do dwóch projektów, w których pełniłem funkcje kierownika z ramienia Comarch. Były to:

- Sektorowy Program Operacyjny Wzrost Konkurencyjności Przedsiębiorstw (SPO WKP), tytuł projektu: *Informatyczny system zarządzania dla małych i średnich przedsiębiorstw zintegrowany z modułem wspomagania decyzji*, beneficjenci: Comarch, Akademia Ekonomiczna w Poznaniu, okres realizacji: IX 2005 – VI 2007.

  Celem projektu było zaprojektowanie architektury nowego systemu ERP na rynki międzynarodowe (obecna nazwa systemu to: Comarch ERP Altum). Do moich obowiązków należało kierowanie zespołem wykonujący zadania dotyczące tego projektu w Comarch oraz integracja wiedzy dostarczonej przez drugiego beneficjenta (Akademia Ekonomiczna w Poznaniu).

- Sektorowy Program Operacyjny Wzrost Konkurencyjności Przedsiębiorstw (SPO WKP), tytuł projektu: *Efektywny system wspomagania decyzji oparty o controllingową hurtownię danych*, beneficjenci: Comarch, Politechnika Łódzka, okres realizacji: IV.2007 – VIII.2008.

  Celem projektu było zaprojektowanie hurtowni danych oraz systemu wspomagania decyzji opartego na tej hurtowni danych dla Comarch ERP Altum. W projekcie pełniłem funkcje głównego projektanta hurtowni danych dla systemu Comarch ERP Altum. Zajmowałem się także tworzeniem raportów z wykonywanych w projekcie zadań.

Projekt wykonywany podczas pracy na Politechnice Wrocławskiej:

- Projekt finansowany przez UE z Europejskiego Funduszu Rozwoju Regionalnego i Budżetu Państwa, POIG 01.03.01-02-002/08, tytuł projektu: *Czujniki i sensory do pomiarów czynników stanowiących zagrożenia w środowisku - modelowanie i monitorowanie zagrożeń*, 2008–2012. Rola: Wykonawca zadania poświęconego analizie on-line.

5.2 Współpraca z uczelniami zagranicznymi

Wyjazd naukowy:

- Xidian University, Xi'an City, Shannxi Province, China, Internship – Cloud computing, 2015 (2 tygodnie).

  W ramach wyjazdu odbyło się kilka spotkań z grupą Prof. Xiaofeng Chen. Podczas wyjazdu wygłosiłem referat: *Efficient Probabilistic Methods for Proof of Possession in Clouds.*

Współpraca z uczelniami:

Od roku 2023 zacieśniłem współpracę z Prof. Carlosem Ordonezem z Houston University, USA, oraz Prof. Ladjelem Bellatreche z National Engineering School for Mechanics and Aerotechnics (ENSMA), Poitiers, Francja. Efekt współpracy to:

- Publikacje [B1], [B5], [B3], [B5] oraz [A10] opublikowane na uznanych konferencjach: IEEE Big Data oraz SIGMOD.
- Zredagowanie *Special Issue-Data Engineering and Modeling for Artificial Intelligence* na podstawie prac zaakceptowanych na workshopie *International Workshop on Data Engineering and Modeling for AI (DEMAI 2023)* oraz praca [B4].
- Opracowanie tutoriala *Green Analytics* oraz zaprezentowanie go na konferencji *IEEE International Conference on Big Data (IEEE BigData 2023)*. Celem tutoriala było zaprezentowanie aktualnych trendów związanych z problemami zużycia energii podczas wykonywania procesów analitycznych w dużych centrach obliczeniowych.
- Organizacja workshopu *International Workshop on Data Engineering and Modeling for AI (DEMAI 2024)* sprzężonego z konferencją *IEEE International Conference on Big Data (IEEE BigData 2024)*. Moja rola: workshop co-chair. Celem workshopu jest dyskusja o problemach badawczych związanych z wykorzystywaniem sztucznej inteligencji w bazach danych oraz baz danych w procesach sztucznej inteligencji.

6 INFORMACJA O OSIĄGNIĘCIACH DYDAKTYCZNYCH, ORGANIZACYJNYCH ORAZ POPULARYZUJĄCYCH NAUKĘ LUB SZTUKĘ

6.1 Kształcenie młodej kadry naukowej

- Byłem promotorem pomocniczym pracy doktorskiej mgr inż. Michała Kukowskiego: "Indeksowanie baz danych na nowoczesnych typach pamięci". Praca została obroniona w 2024 roku w dyscyplinie Informatyka Techniczna i Telekomunikacja na Wydziale Informatyki i Telekomunikacji Politechniki Wrocławskiej.

6.2 Prowadzone prace magisterskie i inżynierskie

Byłem promotorem prac dyplomowych na kierunku informatyka (I i II stopnia) wydziału Podstawowych Problemów Techniki, na kierunku Matematyka Stosowana oraz na kierunku Informatyka Algorytmiczna na Wydziale Informatyki i Telekomunikacji Politechniki Wrocławskiej.

- Byłem promotorem ponad 100 prac inżynierskich.

- Byłem promotorem około 30 prac magisterskich. Wyniki trzech prac: mgr inż. Krzysztofa Kwiatkowskiego, mgr inż Macieja Pawlika oraz mgr inż Michała Kukowskiego zostały opublikowane na konferencjach międzynarodowych. Kilka prac zostało wdrożonych w firmie Comarch.

6.3 Prowadzone zajęcia dydaktyczne

Po uzyskaniu stopnia doktora prowadziłem zajęcia z przedmiotów informatycznych na kierunku informatyka (I i II stopnia) wydziału Podstawowych Problemów Techniki, na kierunku Matematyka Stosowana oraz na kierunku Informatyka Algorytmiczna na Wydziale Informatyki i Telekomunikacji Politechniki Wrocławskiej. Poniżej przedstawiam wszystkie kursy z podziałem na wykłady, ćwiczenia i laboratoria.

Wykłady:

- Bazy danych i systemy informacyjne
- Kurs programowania
- Metody wytwarzania oprogramowania
- Technologie programowania
- Stream Programming (w j. angielskim)
- Aplikacje bazodanowe
- Zarządzanie projektem informatycznym

Ćwiczenia:

- Bazy danych i systemy informacyjne
- Technologie programowania
- Aplikacje bazodanowe

Laboratoria:

- Bazy danych i systemy informacyjne
- Kurs programowania
- Metody wytwarzania oprogramowania
- Technologie programowania
- Aplikacje bazodanowe
- Języki i paradygmaty programowania

- Projekt zespołowy
- Technologie sieciowe
- Technologie WWW

Przygotowałem sylabusy do następujących zajęć:

- Kurs programowania
- Metody wytwarzania oprogramowania
- Aplikacje bazodanowe

6.4 Propagacja wiedzy poza uczelnią

- Prowadziłem kilkanaście edycji kursu *MySQL- relacyjny system baz danych* w Centrum Kształcenia Ustawicznego Politechniki Wrocławskiej.

  Kurs miał charakter praktyczny. Jego celem jest przekazanie podstawowej wiedzy na temat języka SQL oraz systemu MySQL. Kurs ten ukończyło łącznie kilkaset osób z firm zewnętrznych.

- Opracowałem tutorial *Green Analytics* oraz zaprezentowałem go na konferencji *IEEE International Conference on Big Data (IEEE BigData 2023)* w Sorrento (Włochy).

  Celem tutoriala było zaprezentowanie aktualnych trendów związanych z problemami zużycia energii podczas wykonywania procesów analitycznych w dużych centrach obliczeniowych. Tutorial przygotowałem we współpracy z Prof. Carlosem Ordonezem z Houston University, USA, oraz Prof. Ladjelem Bellatreche z National Engineering School for Mechanics and Aerotechnics (ENSMA), Poitiers, Francja.

6.5 Projekty dydaktyczne

Brałem udział w następujących projektach dydaktycznych:

- Program Operacyjny Wiedza Edukacja Rozwój, Priorytet III. Szkolnictwo wyższe dla gospodarki i rozwoju, Działania 3.5 Kompleksowe programy szkół wyższych, współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego. Projekt „ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej" o nr WND-POWR.03.05.00-00-Z301/17, realizowany przez Politechnikę Wrocławską. Okres trwania projektu: 01.10 - 31.12.2018. Charakter uczestnictwa: Opracowanie materiałów dydaktycznych dla przedmiotów praktycznych kierunku Informatyka.
- Projekt "Cloud Computing – nowe technologie w ofercie dydaktycznej Politechniki Wrocławskiej", współfinansowanego ze środków Europejskiego Funduszu Społecznego w ramach Programu Operacyjnego Kapitał Ludzki (Priorytet IV - Szkolnictwo wyższe i nauka; Działanie 4.3. Wzmocnienie potencjału dydaktycznego uczelni w obszarach kluczowych w kontekście celów Strategii Europa 2020. Celem projektu było podniesienie kompetencji dydaktycznych w obszarze związanym z chmurą obliczeniową oraz jej wykorzystaniem do analizowania danych.

6.6 Działalność organizacyjna

- Jestem specjalistą do spraw praktyk studenckich na kierunku "Informatyka Algorytmiczna" na Wydziale Informatyki i Telekomunikacji Politechniki Wrocławskiej. Moja rola polega na opiniowaniu praktyk realizowanych w różnych firmach przez studentów kierunku "Informatyka Algorytmiczna" na Wydziale Informatyki i Telekomunikacji Politechniki Wrocławskiej.

Literatura

[1] Daniel J. Abadi, Samuel Madden, Nabil Hachem. Column-stores vs. row-stores: how different are they really? Jason Tsong-Li Wang, redaktor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, strony 967–980. ACM, 2008.

[2] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, Shashi Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[3] Manoussos Athanassoulis, Kenneth Bøgh, Stratos Idreos. Optimal column layout for hybrid workloads. *Proceedings of the VLDB Endowment*, 2019.

[4] R. Bayer, E. McCreight. Organization and maintenance of large ordered indices. *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, strona 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, Lipiec 1970.

[6] Si-Woo Byun, Seok-Woo Jang. A column-aware index management using flash memory for read-intensive databases. *Journal of Information Processing Systems*, 11(3):389–405, 2015.

[7] Anderson C Carniel, Ricardo R Ciferri, Cristina DA Ciferri. A generic and efficient framework for flash-aware spatial indexing. *Information Systems*, 82:102–120, 2019.

[8] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, Cristina Dutra de Aguiar Ciferri. Analyzing the performance of spatial indices on hard disk drives and flash-based solid state drives. *Journal of Information and Data Management*, 8(1):34–34, 2017.

[9] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, Cristina Dutra de Aguiar Ciferri. A generic and efficient framework for spatial indexing on flash-based solid state drives. *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings 21*, strony 229–243. Springer, 2017.

[10] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, Yinlong Xu. {SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage. *19th USENIX Conference on File and Storage Technologies (FAST 21)*, strony 17–32, 2021.

[11] Shimin Chen, Phillip B Gibbons, Suman Nath, i in. Rethinking database algorithms for phase change memory. *Cidr*, wolumen 11, strony 9–12, 2011.

[12] Ping Chi, Wang-Chien Lee, Yuan Xie. Making b+-tree efficient in pcm-based main memory. *Proceedings of the 2014 international symposium on Low power electronics and design*, strony 69–74, 2014.

[13] Kristina Chodorow, Michael Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly, 2010.

[14] Adina Crainiceanu, Daniel Lemire. Bloofi: Multidimensional bloom filters. *Inf. Syst.*, 54:311–324, 2015.

[15] Ghulam Dastgeer, Sobia Nisar, Aamir Rasheed, Kamran Akbar, Vijay D Chavan, Deok-kee Kim, Saikh Mohammad Wabaidur, Muhammad Wajid Zulfiqar, Jonghwa Eom. Atomically engineered, high-speed non-volatile flash memory device exhibiting multibit data storage operations. *Nano Energy*, 119:109106, 2024.

[16] Niv Dayan, Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. *Proceedings of the 2021 International Conference on Management of Data*, strony 365–378, 2021.

[17] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, Moshe Twitto. Spooky: granulating lsm-tree compactions correctly. *Proceedings of the VLDB Endowment*, 15(11):3071–3084, 2022.

[18] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, Michael Strum. Optimizing space amplification in rocksdb. *CIDR*, wolumen 3, strona 3, 2017.

[19] Athanasios Fevgas, Leonidas Akritidis, Miltiadis Alamaniotis, Panagiota Tsompanopoulou, Panayiotis Bozanis. Hyr-tree: a spatial index for hybrid flash/3d xpoint storage. *Neural Computing and Applications*, strony 1–13, 2023.

[20] Congming Gao, Xin Xin, Youyou Lu, Youtao Zhang, Jun Yang, Jiwu Shu. Parabit: processing parallel bitwise operations in nand flash memory based ssds. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, strony 59–70, 2021.

[21] Lars George. *HBase: The Definitive Guide.* O'Reilly Media, wydanie 1, 2011.

[22] Google. Leveldb. `https://github.com/google/leveldb`.

[23] Marcin Gorawski, Rafal Malczok. Materialized ar-tree in distributed spatial data warehouse. *Intelligent Data Analysis*, 10(4):361–377, 2006.

[24] Goetz Graefe, Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, strony 371–381, New York, NY, USA, 2010. ACM.

[25] Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu. Columnar storage and list-based processing for graph database management systems. *arXiv preprint arXiv:2103.02284*, 2021.

[26] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, strona 47–57, New York, NY, USA, 1984. Association for Computing Machinery.

[27] Runze Han, Yachen Xiang, Peng Huang, Yihao Shan, Xiaoyan Liu, Jinfeng Kang. Flash memory array for efficient implementation of deep neural networks. *Advanced Intelligent Systems*, 3(5):2000161, 2021.

[28] Junjun He, Huahui Chen. An lsm-tree index for spatial data. *Algorithms*, 15(4):113, 2022.

[29] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, Peter A. Boncz. Positional update handling in column stores. Ahmed K. Elmagarmid, Divyakant Agrawal, redaktorzy, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, strony 543–554. ACM, 2010.

[30] Van Phi Ho, Dong-Joo Park. Wpcb-tree: a novel flash-aware b-tree index using a write pattern converter. *Symmetry*, 10(1):18, 2018.

[31] Haoyu Huang, Shahram Ghandeharizadeh. Nova-lsm: a distributed, component-based lsm-tree key-value store. *Proceedings of the 2021 International Conference on Management of Data*, strony 749–763, 2021.

[32] Stratos Idreos, Martin L. Kersten, Stefan Manegold. Database cracking. *In CIDR*, 2007.

[33] Stratos Idreos, Martin L. Kersten, Stefan Manegold. Updating a cracked database. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, strony 413–424, New York, NY, USA, 2007. ACM.

[34] Stratos Idreos, Martin L. Kersten, Stefan Manegold. Self-organizing tuple reconstruction in column-stores. *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, strony 297–308, New York, NY, USA, 2009. ACM.

[35] Kazunari Ishimaru. Challenges of flash memory for next decade. *2021 IEEE International Reliability Physics Symposium (IRPS)*, strony 1–5. IEEE, 2021.

[36] Elkhan Jabarov, Byung-Won On, Gyu Sang Choi, Myong-Soon Park. R-tree for phase change memory. *Comput. Sci. Inf. Syst.*, 14(2):347–367, 2017.

[37] Elkhan Jabarov, Myong-Soon Park, Byung-Won On, Gyu Sang Choi. Pcr*-tree: Pcm-aware r*-tree. *J. Inf. Sci. Eng.*, 33(5):1359–1374, 2017.

[38] L Jeevitha, B Umadevi, M Hemavathy. Sata protocol implementation on fpga for write protection of hard disk drive/solid state device. *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, strony 614–617. IEEE, 2019.

[39] Peiquan Jin, Xike Xie, Na Wang, Lihua Yue. Optimizing r-tree for flash memory. *Expert Systems with Applications*, 42(10):4676–4686, 2015.

[40] Peiquan Jin, Chengcheng Yang, Christian S Jensen, Puyuan Yang, Lihua Yue. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal*, 25:695–717, 2016.

[41] Rize Jin. B-tree index layer for multi-channel flash memory. *Mobile and Wireless Technologies 2017: ICMWT 2017 4*, strony 197–202. Springer, 2018.

[42] Rize Jin, Hyung-Ju Cho, Tae-Sun Chung. A group round robin based b-tree index storage scheme for flash memory devices. *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*, strony 1–6, 2014.

[43] Rize Jin, Hyung-Ju Cho, Tae-Sun Chung. Ls-lru: A lazy-split lru buffer replacement policy for flash-based b+-tree index. *J. Inf. Sci. Eng.*, 31(3):1113–1132, 2015.

[44] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, Julian Feinauer. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, strony 3340–3352. IEEE, 2022.

[45] M. L. Kersten, S. Manegold, Martin Kersten, Stefan Manegold. Cracking the database store. *In CIDR*, 2005.

[46] Giuk Kim, Sangho Lee, Taehyong Eom, Taeho Kim, Minhyun Jung, Hunbeom Shin, Yeongseok Jeong, Myounggon Kang, Sanghun Jeon. High performance ferroelectric field-effect transistors for large memory-window, high-reliability, high-speed 3d vertical nand flash memory. *Journal of Materials Chemistry C*, 10(26):9802–9812, 2022.

[47] Hyeong-Jun Kim, Young-Sik Lee, Jin-Soo Kim. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssd. *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[48] Jun Hyoung Kim, Yongsik Yim, Joonsung Lim, Hyun Suk Kim, Eun Suk Cho, Chadong Yeo, Woongseop Lee, Byungkwan You, Byoungil Lee, Minkyu Kang, i in. Highly manufacturable 7 th generation 3d nand flash memory with cop structure and double stack process. *2021 Symposium on VLSI Technology*, strony 1–2. IEEE, 2021.

[49] Min-Kyu Kim, Ik-Jyae Kim, Jang-Sik Lee. Cmos-compatible ferroelectric nand flash memory for high-density, low-power, and high-speed three-dimensional memory. *Science advances*, 7(3):eabe1341, 2021.

[50] Moosung Kim, Sung Won Yun, Jungjune Park, Hyun Kook Park, Jungyu Lee, Yeong Seon Kim, Daehoon Na, Sara Choi, Youngsun Song, Jonghoon Lee, i in. A 1tb 3b/cell 8th-generation 3d-nand flash memory with 164mb/s write throughput and a 2.4 gb/s interface. *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, wolumen 65, strony 136–137. IEEE, 2022.

[51] M. Knowles. Survey of the storage evolution. *2003 User Group Conference. Proceedings*, strony 362–367, 2003.

[52] Avinash Lakshman, Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[53] Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, Panagiotis Karras. Adaptive indexing in high-dimensional metric spaces. *Proceedings of the VLDB Endowment*, 16(10):2525–2537, 2023.

[54] Eleazar Leal, Le Gruenwald. Spatial database cracking with quadtrees.

[55] Eung Chang Lee, Jinsung Rho, Bong Jae Lee, Heeyoub Kang. Heat dissipation analysis of m.2 nvme solid-state drive in vacuum. *2019 International Vacuum Electronics Conference (IVEC)*, strony 1–2, 2019.

[56] Sung-Tae Lee, Gyuho Yeom, Honam Yoo, Hyeong-Su Kim, Suhwan Lim, Jong-Ho Bae, Byung-Gook Park, Jong-Ho Lee. Novel method enabling forward and backward propagations in nand flash memory for on-chip learning. *IEEE Transactions on Electron Devices*, 68(7):3365–3370, 2021.

[57] Fei Li, Youyou Lu, Zhe Yang, Jiwu Shu. Sinekv: Decoupled secondary indexing for lsm-based key-value stores. *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, strony 1112–1122. IEEE, 2020.

[58] Guohui Li, Pei Zhao, Ling Yuan, Sheng Gao. Efficient implementation of a multi-dimensional index structure over flash memory storage systems. *The Journal of Supercomputing*, 64:1055–1074, 2013.

[59] Lu Li, Peiquan Jin, Chengcheng Yang, Shouhong Wan, Lihua Yue. Xb+-tree: A novel index for pcm/dram-based hybrid memory. *Australasian Database Conference*, strony 357–368. Springer, 2016.

[60] Lu Li, Peiquan Jin, Chengcheng Yang, Zhangling Wu, Lihua Yue. Optimizing b+-tree for pcm-based hybrid memory. *EDBT*, strony 662–663, 2016.

[61] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, Ke Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[62] K. M. Lofgren, Gregory B. Norman, R. D. andv Thelin, Gupta. Wear leveling techniques for flash eeprom systems. *US Patent 6,594,183*, 1998.

[63] K. M. Lofgren, R. D. Norman, Gregory B. Thelin, Gupta. Wear leveling techniques for flash eeprom systems. *US Patent 6,081,447*, 1999.

[64] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, Tongliang Deng. Tridentkv: A read-optimized lsm-tree based kv store via adaptive indexing and space-efficient partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1953–1966, 2021.

[65] Chen Luo, Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[66] J. M. Marshall, C. D. H. Manning. Flash file management system. *US Patent 5,832,493*, 1998.

[67] Yoshinori Matsunobu, Siying Dong, Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[68] Supriya Mishra, Bhavesh N. Gohil, Suprio Ray. A survey on persistent memory indexes: Recent advances, challenges and opportunities. *Journal of Systems Architecture*, 151:103140, 2024.

[69] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[70] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[71] Yi Ou, Lei Chen, Jianliang Xu, Theo Härder. Wear-aware algorithms for pcm-based database buffer pools. *International Conference on Web-Age Information Management*, strony 165–176. Springer, 2014.

[72] Jisung Park, Roknoddin Azizi, Geraldo F Oliveira, Mohammad Sadrosadati, Rakesh Nadig, David Novo, Juan Gómez-Luna, Myungsuk Kim, Onur Mutlu. Flash-cosmos: In-flash bulk bitwise operations using inherent computation capability of nand flash memory. *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, strony 937–955. IEEE, 2022.

[73] Helmut Prodinger. Approximate counting with m counters: a detailed analysis. *Theoretical Computer Science*, 439:58–68, 2012.

[74] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, strony 14–23. IEEE, 2009.

[75] Saeed Rashidi, Majid Jalili, Hamid Sarbazi-Azad. A survey on pcm lifetime enhancement schemes. *ACM Computing Surveys (CSUR)*, 52(4):1–38, 2019.

[76] H Riggs, S Tufail, I Parvez, A Sarwat. Survey of solid state drives, characteristics, technology, and applications. *2020 SoutheastCon*, strony 1–6. IEEE, 2020.

[77] George Roumelis, Michael Vassilakopoulos, Thanasis Loukopoulos, Antonio Corral, Yannis Manolopoulos. The xbr-tree: an efficient access method for points. *International Conference on Data Management in Cloud, Grid and P2P Systems*, strony 43–58. Springer, 2015.

[78] George Roumelis, Polychronis Velentzas, Michael Vassilakopoulos, Antonio Corral, Athanasios Fevgas, Yannis Manolopoulos. Parallel processing of spatial batch-queries using xbr+-trees in solid-state drives. *Cluster Computing*, 23(3):1555–1575, 2020.

[79] Young-Tak Seo, Dongseok Kwon, Yoohyun Noh, Soochang Lee, Min-Kyu Park, Sung Yun Woo, Byung-Gook Park, Jong-Ho Lee. 3-d and-type flash memory architecture with high-κ gate dielectric for high-density synaptic devices. *IEEE Transactions on Electron Devices*, 68(8):3801–3806, 2021.

[80] Giorgio Servalli. A 45nm generation phase change memory technology. *2009 IEEE International Electron Devices Meeting (IEDM)*, strony 1–4. IEEE, 2009.

[81] Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, Goetz Graefe. Fast scans and joins using flash drives. Qiong Luo, Kenneth A. Ross, redaktorzy, *4th Workshop on Data Management*

on New Hardware, DaMoN 2008, Vancouver, BC, Canada, June 13, 2008, strony 17–24. ACM, 2008.

[82] Jaewoo Shin. *EFFICIENT LSM SECONDARY INDEXING FOR UPDATE-INTENSIVE WORKLOADS.* Praca doktorska, Purdue University Graduate School, 2023.

[83] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, Jiwu Shu. Revisiting secondary indexing in {LSM-based} storage systems with persistent memory. *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, strony 817–832, 2023.

[84] David P Williamson, David B Shmoys. *The design of approximation algorithms.* Cambridge university press, 2011.

[85] Chin-Hsien Wu, Li-Pin Chang, Tei-Wei Kuo. An efficient r-tree implementation over flash-memory storage systems. *ACM-GIS 2003, Proceedings of the Eleventh ACM International Symposium on Advances in Geographic Information Systems, New Orleans, Louisiana, USA, November 7-8, 2003*, strony 17–24. ACM, 2003.

[86] Gang Wu, Yidong Song, Guodong Zhao, Wei Sun, Donghong Han, Baiyou Qiao, Guoren Wang, Ye Yuan. Cracking in-memory database index: A case study for adaptive radix tree index. *Information Systems*, 104:101913, 2022.

[87] Zhangling Wu, Peiquan Jin, Lihua Yue. Efficient space management and wear leveling for pcm-based storage systems. *International Conference on Algorithms and Architectures for Parallel Processing*, strony 784–798. Springer, 2015.

[88] Chi-Weon Yoon. The fundamentals of nand flash memory: Technology for tomorrow's fourth industrial revolution. *IEEE Solid-State Circuits Magazine*, 14(2):56–65, 2022.

[89] Jianan Yuan, Huan Liu, Shangyu Wu, Yiquan Lin, Tiantian Wang, Chenlin Ma, Rui Mao, Yi Wang. Work-in-progress: Lark: A learned secondary index toward lsm-tree for resource-constrained embedded storage systems. *2022 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, strony 11–12. IEEE, 2022.

[90] Fatemeh Zardbani, Peyman Afshani, Panagiotis Karras. Revisiting the theory and practice of database cracking. *EDBT*, strony 415–418, 2020.

[91] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, Gang Chen. Dptree: Differential indexing for persistent memory. *Proceedings of the VLDB Endowment*, 13(4):421–434, 2019.