Gadka:.

**Rozbudowana Wypowiedź z Wyjaśnieniem Działania Umbry:**

**1. Motywacja i Cel:**

"Motywacją do stworzenia tej symulacji było zrozumienie, jak systemy bazodanowe takie jak **Umbra** osiągają wysoką wydajność pomimo ograniczeń sprzętowych. Umbra to system bazodanowy, który łączy zalety baz danych działających w całości w pamięci RAM (bardzo szybkich) z zaletami baz danych działających na dysku (mogących obsługiwać ogromne ilości danych). Chciałem zdemaskować 'magię' szybkiego dostępu do danych, nawet gdy bazy danych są ogromne i nie mieszczą się w całości w pamięci RAM. Celem było stworzenie uproszczonego modelu, który pozwoli na eksperymentowanie z różnymi parametrami i obserwowanie ich wpływu na zachowanie systemu."

**2. Jak Działa Umbra (Krótkie Wprowadzenie):**

"Umbra, w skrócie, działa tak: wykorzystuje duży bufor w pamięci RAM do przechowywania często używanych danych i szybkie dyski SSD jako magazyn danych. Dzięki temu, jeśli dane są w buforze, dostęp do nich jest bardzo szybki (jak w bazie danych w pamięci), a jeśli nie, to system nadal może obsłużyć ogromne ilości danych z dysku. Umbra wprowadza kilka innowacji, żeby to działało sprawnie, np. nowy sposób zarządzania buforem, który pozwala na przechowywanie stron o różnych rozmiarach."

**3. Drzewo B+ jako Indeks (Organizacja Danych w Umbrze):**

"Podstawowym elementem zarówno mojej symulacji, jak i systemu Umbra, jest **drzewo B+**. To struktura danych, która pełni rolę potężnego indeksu. Wyobraźmy sobie, że mamy ogromną bibliotekę. Drzewo B+ to katalog, który pozwala nam błyskawicznie znaleźć każdą książkę. W bazie danych, drzewo B+ organizuje dane w posortowany sposób, co umożliwia:

* **Szybkie wyszukiwanie:** Znalezienie konkretnego rekordu po kluczu (np. ID klienta).
* **Efektywne przetwarzanie zakresów:** Wyciągnięcie wszystkich rekordów spełniających określone kryteria (np. zamówienia złożone w danym okresie).
* **Utrzymywanie porządku:** Automatyczne sortowanie danych, co jest ważne dla wielu operacji.

W symulacji, drzewo B+ składa się z węzłów, które przechowują klucze i wskaźniki do innych węzłów lub do samych danych. Operacje takie jak wstawianie i wyszukiwanie są realizowane rekurencyjnie, przechodząc przez kolejne poziomy drzewa. Umbra również używa drzew B+ do organizowania swoich danych."

**4. Bufor Pamięci (Hierarchia Pamięci i Lokalność Odwołań w Umbrze):**

"Jednak dostęp do danych na dysku jest rząd wielkości wolniejszy niż dostęp do pamięci RAM. Dlatego kluczowym elementem zarówno mojej symulacji, jak i systemu Umbra, jest **bufor pamięci**. To obszar RAM, który działa jak podręczny schowek na najczęściej używane dane. Baza danych stosuje zasadę **lokalności odwołań**:

* **Lokalność czasowa:** Jeśli odwołaliśmy się do jakichś danych, prawdopodobnie wkrótce znowu będziemy ich potrzebować.
* **Lokalność przestrzenna:** Jeśli odwołaliśmy się do jakiejś danej, prawdopodobnie będziemy potrzebować danych znajdujących się obok niej.

Bufor wykorzystuje te zasady. Gdy baza danych potrzebuje danych, najpierw sprawdza **frameMap**, czyli mapę stron w buforze. Jeśli dane są w buforze (**HIT**), odczyt jest bardzo szybki. Zwiększany jest licznik countHit. Jeśli danych nie ma (**MISS**), trzeba je pobrać z dysku. Zwiększany jest licznik countMiss, a następnie symulujemy odczyt z dysku i umieszczamy dane w buforze. Umbra ma swój własny, zaawansowany menedżer bufora, który optymalizuje ten proces."

**5. Algorytm Ewikcji (Zarządzanie Buforem w Umbrze):**

"Bufor ma ograniczoną pojemność. Gdy jest pełny, a potrzebujemy załadować nowe dane, musimy zwolnić miejsce. W symulacji użyłem prostego algorytmu **FIFO (First-In, First-Out)**. Oznacza to, że usuwamy stronę, która została załadowana do bufora jako pierwsza. To nie jest idealny algorytm, ale jest prosty i pozwala na demonstrację podstawowych zasad zarządzania buforem. Umbra stosuje bardziej zaawansowane techniki zarządzania buforem, w tym strony o zmiennych rozmiarach, żeby efektywnie obsługiwać różne typy danych."

**6. Fetchowanie (Interakcja z Dyskiem w Umbrze):**

"Proces pobierania danych z dysku do bufora nazywa się **fetchowaniem**. W symulacji, fetchowanie jest symulowane przez utworzenie nowego obiektu Page i umieszczenie go w buforze. W prawdziwym systemie bazodanowym, fetchowanie wiąże się z operacjami I/O (Input/Output), które są kosztowne i czasochłonne. Dlatego tak ważne jest efektywne zarządzanie buforem. Umbra stara się minimalizować operacje I/O, używając szybkich dysków SSD i optymalizując dostęp do danych."

**7. Latchowanie (Współbieżność i Spójność w Umbrze):**

"W systemach bazodanowych, takich jak Umbra, wiele użytkowników (lub aplikacji) może jednocześnie korzystać z bazy danych. Dlatego musimy zapewnić **współbieżność** (jednoczesny dostęp) i **spójność** danych (brak uszkodzeń). Do tego służy **latchowanie**. Latch to lekki mechanizm blokowania, który chroni dane przed jednoczesnymi modyfikacjami. W symulacji użyłem ReentrantReadWriteLock.

* **Read latch (latch odczytu):** Wiele wątków może jednocześnie posiadać latch odczytu do tej samej strony.
* **Write latch (latch zapisu):** Tylko jeden wątek może posiadać latch zapisu do danej strony.

Przed odczytem lub zapisem strony, wątek musi 'złapać' odpowiedni latch. Po zakończeniu operacji, wątek 'zwalnia' latch. To zapewnia, że dane nie zostaną uszkodzone przez jednoczesne operacje. Umbra również używa mechanizmów latchowania, żeby zapewnić bezpieczny dostęp do danych w środowisku wielowątkowym."

**8. Swizzlowanie (Optymalizacja Wskaźników w Umbrze):**

"Ostatnim ważnym elementem, obecnym zarówno w mojej symulacji, jak i w Umbrze, jest **swizzlowanie**. W bazie danych mamy wskaźniki do danych. Na dysku identyfikujemy strony za pomocą identyfikatorów (pageId). W pamięci RAM identyfikujemy je za pomocą adresów. Swizzlowanie to proces zamiany identyfikatorów stron na adresy pamięci, gdy strona jest ładowana do bufora. Dzięki temu dostęp do danych w buforze jest szybszy. W symulacji użyłem klasy Swip do reprezentowania wskaźników, które mogą być swizzled (wskazywać na obiekt Page w RAM) lub unswizzled (zawierać pageId). Umbra również korzysta z technik swizzlingu, żeby zoptymalizować dostęp do danych w pamięci."

**9. Eksperymenty i Wyniki:**

"W symulacji przeprowadziłem eksperymenty, zmieniając różne parametry, takie jak:

* **Rozmiar bufora:** Jak wpływa na liczbę hitów i missów.
* **Rząd drzewa B+:** Jak wpływa na efektywność wyszukiwania.
* **Liczba wątków:** Jak wpływa na wydajność w środowisku współbieżnym.

Obserwowałem takie metryki jak:

* **Hit rate (współczynnik trafień):** Odsetek odczytów z bufora.
* **Miss rate (współczynnik chybien):** Odsetek odczytów z dysku.
* **Czas wykonania:** Całkowity czas potrzebny na wykonanie operacji.
* **Liczba ewikcji:** Ile stron musiało zostać usuniętych z bufora.

Wyniki pokazują, jak ważne są opisane mechanizmy dla wydajności bazy danych. Na przykład, zwiększenie rozmiaru bufora zazwyczaj prowadzi do wzrostu hit rate i zmniejszenia czasu wykonania. Umbra potwierdza te obserwacje w praktyce, osiągając wysoką wydajność dzięki efektywnemu zarządzaniu buforem, optymalizacji dostępu do danych i współbieżności."

**10. Podsumowanie i Wnioski:**

"Podsumowując, moja symulacja ilustruje kluczowe elementy działania systemu bazodanowego, a w szczególności systemu Umbra: efektywną organizację danych (drzewo B+), zarządzanie pamięcią (bufor, algorytm ewikcji), bezpieczny dostęp współbieżny (latchowanie) i optymalizację dostępu (swizzlowanie). Pozwala to zrozumieć, jak bazy danych osiągają wysoką wydajność i jakie kompromisy muszą podejmować. Mam nadzieję, że ta prezentacja rzuciła światło na fascynujący świat wewnętrznych mechanizmów systemów bazodanowych."

WYKRESY GADKA

**Ogólne Tendencje**

* **Wpływ bufferSize (Rozmiaru Bufora):**

Wzrost bufferSize (np. z 256 do 4096) zazwyczaj powoduje:

* + - Znaczący wzrost meanHitRate (średniego współczynnika trafień) i spadek MeanMissRate (średniego współczynnika chybień). Jest to zgodne z oczekiwaniami, ponieważ większy bufor może pomieścić więcej danych, zmniejszając potrzebę odczytu z dysku.
    - Spadek meanEvictions (średniej liczby usunięć). Większy bufor oznacza mniejszą potrzebę zastępowania stron/bloków.
    - Złożony wpływ na sumTime (całkowity czas) i meanElapsedMs (średni czas operacji). Początkowo czas ten maleje (dzięki lepszym trafieniom), ale przy bardzo dużych buforach może nieznacznie wzrosnąć z powodu narzutu związanego z zarządzaniem większym buforem.

**Wpływ TreeOrder (Rzędu Drzewa):**

* + Wzrost TreeOrder (np. z 16 do 128) zazwyczaj powoduje:
    - Wzrost meanHitRate i spadek MeanMissRate. Wyższy rząd drzewa prowadzi do płytszego drzewa B+, co może poprawić efektywność wyszukiwania.
    - Spadek sumTime i meanElapsedMs. Prawdopodobnie dzięki efektywniejszemu wyszukiwaniu i lepszemu wykorzystaniu bufora.

**Szczegółowe Obserwacje**

* **Wysokie Współczynniki Trafień przy Dużych bufferSize i TreeOrder:**
  + Gdy bufferSize wynosi 2048 lub 4096, a TreeOrder to 64 lub 128, obserwujemy bardzo wysokie meanHitRate (powyżej 90%), co wskazuje na efektywne wykorzystanie bufora.
  + W tych konfiguracjach meanElapsedMs jest najniższy, co oznacza optymalną wydajność systemu.

**Kompromisy:**

* + Istnieje wyraźny kompromis między bufferSize a TreeOrder. Dobrą wydajność można osiągnąć przy mniejszym buforze i wysokim rzędzie drzewa (i odwrotnie).
  + Jednak zbyt duży TreeOrder może dać malejące korzyści lub wprowadzić inne obciążenia, których te metryki nie uwzględniają (np. złożoność operacji podziału węzłów).

**Wpływ Liczby Wątków:**

* + Dane są prezentowane dla 1, 8 i 16 wątków. Ważne jest, aby przeanalizować, jak metryki zmieniają się wraz ze wzrostem liczby wątków.
  + Idealnie, sumTime powinno maleć wraz ze wzrostem liczby wątków (dzięki paralelizacji), ale meanElapsedMs może nie maleć proporcjonalnie, a nawet wzrosnąć z powodu narzutu związanego z synchronizacją wątków.
  + hitRate i evictions mogą się zmieniać z powodu konkurencji wątków o dostęp do bufora.

**Klasa B plus tree**

**1. Cel i Funkcja Klasy/Metod**

"Podczas naszej dyskusji o kodzie, chciałbym skupić się na klasie Node i metodach, które zarządzają węzłami w drzewie B+, ponieważ stanowią one fundament efektywnego dostępu do danych. Klasa Node pełni rolę reprezentacji pojedynczego węzła w strukturze drzewa B+. Węzły te, jak Państwo wiedzą, organizują dane w sposób uporządkowany, umożliwiając szybkie wyszukiwanie, wstawianie i usuwanie informacji. Węzły liściowe przechowują rzeczywiste dane, podczas gdy węzły wewnętrzne służą jako swego rodzaju drogowskazy, zawierając klucze i wskaźniki do węzłów potomnych. Metody takie jak loadNode, swizzleNode i generateNewPageId odgrywają kluczowe role w zarządzaniu cyklem życia węzła. Metoda loadNode odpowiada za pobieranie węzłów z pamięci lub dysku, uwzględniając mechanizm swizzlingu, który optymalizuje dostęp do danych. Metoda swizzleNode łączy węzeł z fizyczną stroną pamięci, a generateNewPageId generuje unikalne identyfikatory dla każdej strony. Ponadto, metoda unswizzleTree ma istotne znaczenie dla symulacji, ponieważ pozwala na kontrolowanie, które dane znajdują się w buforze, a które wymagają odczytu z dysku."

**2. Powiązanie z Koncepcjami Umbry**

"W kontekście systemu Umbra, który łączy wydajność systemów in-memory ze skalowalnością systemów dyskowych, klasa Node i te metody nabierają szczególnego znaczenia. Umbra musi efektywnie zarządzać danymi przechowywanymi na dyskach SSD, a nie tylko w pamięci RAM. Koncepcja Swip (Swizzled Pointer) jest kluczowa dla Umbry, a metoda loadNode bezpośrednio obsługuje swizzling i unswizzling. Swizzling polega na tym, że gdy strona z danymi jest wczytywana z dysku do pamięci, wskaźniki do tej strony są aktualizowane, aby wskazywały bezpośrednio na jej lokalizację w pamięci. To eliminuje potrzebę kosztownych odczytów z dysku przy kolejnych dostępach do tych danych. Umbra wykorzystuje również strony o zmiennej wielkości, co odzwierciedla parametr sizeClass w metodzie loadNode. Takie podejście pozwala na lepsze dopasowanie do charakterystyki danych i minimalizację fragmentacji. Wreszcie, generowanie unikalnych identyfikatorów stron (pageId) jest fundamentalne dla Umbry, ponieważ umożliwia jednoznaczną identyfikację i lokalizację każdej strony na dysku, co jest niezbędne w systemie dyskowym."

**3. Znaczenie dla Symulacji**

"W mojej symulacji, klasa Node i związane z nią metody odgrywają kluczową rolę w realistycznym modelowaniu operacji dostępu do danych w drzewie B+. Metoda loadNode symuluje koszt dostępu do danych, uwzględniając potencjalne opóźnienie związane z odczytem z dysku, jeśli strona nie znajduje się aktualnie w buforze. Metoda unswizzleTree pozwala na kontrolowanie, które dane są dostępne w pamięci, a które wymagają odczytu z dysku, co umożliwia badanie wpływu rozmiaru bufora na ogólną wydajność systemu. Metoda generateNewPageId symuluje proces alokacji miejsca na dysku dla nowych stron, choć w uproszczony sposób, ponieważ symulacja nie obejmuje rzeczywistych operacji zapisu na dysku."

**4. Kluczowe Elementy i Optymalizacje**

"Warto podkreślić znaczenie mapy pageToNode, która jest kluczowym elementem mechanizmu swizzlingu. Mapa ta umożliwia szybkie odnalezienie węzła w pamięci na podstawie jego identyfikatora strony (pageId), co znacznie przyspiesza dostęp do często używanych danych. Słowo kluczowe synchronized w metodzie generateNewPageId jest istotne z punktu widzenia bezpieczeństwa wątków, ponieważ zapewnia, że identyfikatory stron są generowane w sposób unikalny w środowisku wielowątkowym, unikając potencjalnych konfliktów. Wreszcie, rekurencyjne podejście zastosowane w metodzie unswizzleNodeRecursively jest efektywnym sposobem na przetworzenie całej struktury drzewa B+ i zaktualizowanie wskaźników do stron."

**KLASA BUFFER MANAGER**

**1. Cel i Funkcjonalność**

* "Chciałbym omówić klasę BufferManager, która odgrywa kluczową rolę w zarządzaniu pamięcią w systemie bazodanowym. Jej głównym zadaniem jest efektywne alokowanie i zwalnianie stron danych między pamięcią RAM a dyskiem (w przypadku Umbry, dyskami SSD)."
* "Klasa ta symuluje działanie menedżera bufora, który jest niezbędny w systemach bazodanowych, aby zminimalizować kosztowne operacje odczytu/zapisu z dysku. W tym uproszczonym modelu, BufferManager zarządza ograniczoną pulą ramek bufora (maxFrames) i używa prostej strategii FIFO do usuwania stron, gdy pamięć jest pełna."
* "Główna metoda, fetchPage(), symuluje pobieranie strony danych. Jeśli strona jest już w pamięci (trafienie w bufor), zwracany jest wskaźnik do niej. W przeciwnym razie (chybienie w buforze), strona jest 'ładowana' z dysku (symulowane przez utworzenie nowego obiektu Page), a jeśli to konieczne, inna strona jest usuwana z bufora."
* "Klasa śledzi również statystyki wydajności, takie jak liczba trafień w buforze, chybien i usunięć stron, co pozwala na ocenę efektywności zarządzania buforem."

**2. Znaczenie w Kontekście Umbry**

* "W systemie Umbra, który łączy zalety systemów in-memory i dyskowych, BufferManager jest kluczowym komponentem. Umbra dąży do osiągnięcia wydajności zbliżonej do systemów in-memory dla często używanych danych, przy jednoczesnym zachowaniu skalowalności systemów dyskowych."
* "Umbra wykorzystuje strony o zmiennych rozmiarach, co jest odzwierciedlone w parametrze sizeClass metody fetchPage(). Pozwala to na optymalne dopasowanie rozmiaru strony do rodzaju przechowywanych danych."
* "Koncepcja 'swizzlingu' jest istotna dla Umbry. BufferManager używa obiektów Swip (Swizzled Pointer), które mogą wskazywać na stronę w pamięci RAM lub na stronie na dysku. Swizzling optymalizuje dostęp do danych poprzez unikanie nadmiernych odczytów z dysku."
* "Efektywne zarządzanie buforem, realizowane przez BufferManager, jest kluczowe dla wydajności Umbry. Odpowiedni rozmiar bufora i strategia usuwania stron mają bezpośredni wpływ na liczbę trafień w buforze i koszt operacji I/O."

**3. Rola w Symulacjach**

* "W mojej symulacji, klasa BufferManager pozwala na modelowanie zachowania bufora pamięci i ocenę wpływu różnych parametrów (takich jak rozmiar bufora) na wydajność systemu."
* "Symulując operacje odczytu stron i śledząc statystyki trafień/chybień/usunięć, mogę analizować, jak zmienia się wydajność systemu w zależności od obciążenia i konfiguracji bufora."
* "Uproszczona strategia usuwania FIFO pozwala na skupienie się na podstawowych aspektach zarządzania buforem, choć można ją rozszerzyć o bardziej zaawansowane algorytmy, aby dokładniej odwzorować zachowanie rzeczywistego systemu."
* "Klasa BufferManager dostarcza narzędzi do eksperymentalnej oceny kompromisów między rozmiarem bufora a kosztami operacji I/O, co jest kluczowe dla optymalizacji wydajności systemu bazodanowego."

**4. Kluczowe Elementy i Decyzje Projektowe**

* "Warto zwrócić uwagę na użycie ConcurrentHashMap do przechowywania stron w pamięci, co zapewnia bezpieczeństwo wątkowe i umożliwia współbieżny dostęp do bufora."
* "Użycie AtomicLong do zliczania trafień, chybien i usunięć gwarantuje poprawność statystyk w środowisku wielowątkowym."
* "Decyzja o użyciu FIFO jako strategii usuwania stron jest podyktowana prostotą implementacji, ale można ją rozważyć zmianę na bardziej zaawansowane algorytmy, takie jak LRU (Least Recently Used), w celu potencjalnej poprawy wydajności."

**Page i Swip**

* Zapytanie zostało przetworzone

OK, przygotujmy omówienie klas Page i Swip do rozmowy z prowadzącym:

**Struktura Omówienia**

1. **Klasa Page - Cel i Funkcjonalność**
2. **Klasa Swip - Cel i Funkcjonalność**
3. **Powiązanie z Umbrą i Symulacjami**
4. **Pytania do Prowadzącego**

**Szczegółowe Punkty**

**1. Klasa Page - Cel i Funkcjonalność**

* "Klasa Page reprezentuje pojedynczą stronę danych, czyli podstawową jednostkę, na której operuje system bazodanowy. W uproszczonym modelu, strona ta zawiera fragment danych, który może być przechowywany na dysku lub w pamięci RAM."
* "Każda strona ma unikalny identyfikator (pageId) i należy do określonej klasy rozmiaru (sizeClass). W tym kodzie, sizeClass 0 odpowiada stronom o rozmiarze 4KB, a sizeClass 1 stronom o rozmiarze 8KB. Umbra wykorzystuje strony o zmiennych rozmiarach, co pozwala na efektywniejsze zarządzanie różnymi typami danych."
* "Klasa Page zawiera również mechanizm blokad (latch) odczytu/zapisu (ReentrantReadWriteLock). Te blokady służą do synchronizacji dostępu wielu wątków do strony, zapewniając spójność danych. Wiele wątków może jednocześnie odczytywać stronę (blokada współdzielona), ale tylko jeden wątek może ją modyfikować (blokada wyłączna)."

**2. Klasa Swip - Cel i Funkcjonalność**

* "Klasa Swip (Swizzled Pointer) to inteligentny wskaźnik do strony. Wprowadza koncepcję 'swizzlingu', która jest kluczowa dla optymalizacji dostępu do danych w Umbrze."
* "Wskaźnik Swip może być w jednym z dwóch stanów: 'swizzlowany' lub 'nieswizzlowany'. Swizzlowany Swip zawiera bezpośrednią referencję do obiektu Page w pamięci RAM, co umożliwia szybki dostęp do danych. Nieswizzlowany Swip zawiera identyfikator strony (pageId) i klasę rozmiaru (sizeClass) i jest używany, gdy strona znajduje się na dysku."
* "Mechanizm swizzlingu pozwala na uniknięcie kosztownych operacji odczytu z dysku przy każdym dostępie do strony. Gdy strona jest wczytywana do pamięci, wskaźniki do niej są aktualizowane (swizzlowane), aby wskazywały bezpośrednio na jej lokalizację w pamięci."

**3. Powiązanie z Umbrą i Symulacjami**

* "Klasy Page i Swip odzwierciedlają kluczowe koncepcje używane w systemie Umbra. Umbra dąży do połączenia wydajności systemów in-memory ze skalowalnością systemów dyskowych."
* "Swizzling, reprezentowany przez klasę Swip, jest kluczową techniką optymalizacji w Umbrze. Pozwala na efektywne zarządzanie dostępem do stron, które mogą znajdować się zarówno w pamięci, jak i na dysku."
* "Mechanizmy blokad w klasie Page są istotne dla zapewnienia spójności danych w systemie współbieżnym, w którym wiele wątków może jednocześnie uzyskiwać dostęp do stron."
* "W symulacjach, te klasy pozwalają na modelowanie operacji odczytu/zapisu stron, zarządzanie buforem pamięci i ocenę wpływu różnych strategii zarządzania buforem na wydajność systemu."

**Przebieg Symulacji**

"Symulacja przebiega w następujących krokach:

* **Iteracja po parametrach:** Kod iteruje po wszystkich kombinacjach zdefiniowanych parametrów. Oznacza to, że dla każdej liczby wątków, dla każdego rozmiaru bufora i dla każdego rzędu drzewa B+ przeprowadzana jest seria eksperymentów.
* **Powtórzenia:** Każdy eksperyment dla danej kombinacji parametrów jest powtarzany określoną liczbę razy (zgodnie z wartością repetitions) w celu uzyskania statystycznie istotnych wyników.
* **Inicjalizacja:** W każdej powtórce tworzone są nowe obiekty BufferManager (menedżera bufora) i BPlusTree (drzewa B+), a także generowane są ziarna dla generatorów liczb losowych, aby zapewnić powtarzalność wyników i losowość między eksperymentami.
* **Wstawianie danych:** Do drzewa B+ wstawiana jest określona liczba losowych rekordów.
* **Odswizzlowanie drzewa:** Po wstawieniu danych, wywoływana jest metoda unswizzleTree(), która powoduje, że wszystkie wskaźniki w drzewie stają się "nieswizzlowane". Ma to na celu symulację sytuacji, w której dane nie znajdują się w całości w pamięci RAM przed rozpoczęciem wyszukiwania, i wymuszenie odczytów stron z dysku (lub bufora).
* **Równoległe wyszukiwanie:** Uruchamiana jest pula wątków, a każdy wątek wykonuje określoną liczbę losowych operacji wyszukiwania w drzewie B+. Wyszukiwane klucze są generowane w sposób symulujący *lokalność odwołań* (częste wyszukiwanie w niewielkim podzbiorze danych).
* **Synchronizacja wątków:** Używany jest mechanizm CountDownLatch, aby główny wątek poczekał na zakończenie wszystkich wątków wyszukujących przed przejściem do dalszych obliczeń.
* **Pomiar czasu i statystyk:** Mierzony jest czas trwania operacji wstawiania i wyszukiwania, a także zbierane są statystyki dotyczące efektywności bufora (liczba trafień, chybien, usunięć stron).
* **Agregacja wyników:** Obliczane są średnie wartości metryk wydajności dla danej kombinacji parametrów (np. średni czas wykonania, średni współczynnik trafień).
* **Wyświetlanie wyników:** Wyniki symulacji są wyświetlane w postaci tabeli, co ułatwia ich analizę i porównywan