Projekt kursu Struktury Baz Danych - Sortowanie Zewnętrzne Szymon Groszkowski 193141

November 18, 2024

1 Cel projektu

Celem projektu jest implementacja programu do sortowania dużych zbiorów danych z wykorzystaniem jednej z metod sortowania zewnętrznego: scalania naturalnego, scalania z użyciem wielkich buforów lub sortowania polifazowego. Program ma za zadanie symulować działanie algorytmu sortowania zewnętrznego w warunkach ograniczonej pamięci RAM. Celem symulacji jest umożliwienie przeprowadzenia doświadczeń, które odzwierciedlają rzeczywiste warunki pracy z dużymi zbiorami danych, które nie mogą zostać w całości załadowane do pamięci operacyjnej. Dzięki temu użytkownik może zaobserwować, jak algorytm sortowania zewnętrznego radzi sobie z zarządzaniem dostępem do danych zapisanych na dysku, co pozwala lepiej zrozumieć wpływ ograniczeń pamięciowych na efektywność algorytmu oraz liczbę operacji I/O.

Ostatecznym celem projektu jest przeprowadzenie eksperymentu, w którym zostaną porównane wyniki teoretyczne i praktyczne liczby faz sortowania oraz operacji dyskowych dla różnych rozmiarów zbiorów danych. Wyniki te zostaną przedstawione na wykresach i poddane analizie w celu zrozumienia efektywności wybranej metody sortowania zewnętrznego.

2 Wybór algorytmu: Sortowanie przy użyciu wielkich buforów

Do realizacji zadania wybrano **algorytm sortowania zewnętrznego przy użyciu wielkich buforów**. Algorytm ten jest efektywny w przypadku pracy z dużymi zbiorami danych, które nie mieszczą się w pamięci RAM, ponieważ wykorzystuje dodatkową pamięć buforową do optymalizacji operacji I/O.

Sortowanie przy użyciu wielkich buforów polega na podziale zbioru danych na mniejsze części, zwane „runami” (partiami), które są następnie wczytywane do pamięci, sortowane i zapisywane na dysku. W kolejnych fazach sortowania algorytm odczytuje po kilka „runów” z dysku do pamięci, łączy je w jeden posortowany zbiór, zapisując wynik w nowym pliku na dysku. Proces ten powtarza się aż do uzyskania pojedynczego, całkowicie posortowanego pliku.

3 Implementacja programu w C# .NET

Program został zaimplementowany w języku C# przy użyciu platformy .NET, co umożliwiło wykorzystanie nowoczesnych narzędzi do zarządzania zależnościami, konfiguracją oraz logowaniem. Dzięki zastosowaniu tych technologii program jest bardziej modularny, konfigurowalny i łatwiejszy w utrzymaniu.

3.1 Symulacja pamięci i odczytu blokowego

Kluczowe funkcje symulacji dostępu do pamięci w programie obejmują zarządzanie odczytem i zapisem stron na dysku oraz operacje na pamięci RAM. Program zapisuje początkowe rekordy do pliku binarnego oraz umożliwia ich odczytanie przed rozpoczęciem sortowania. Symulacja obejmuje operacje ***WritePageToTape*** i ***ReadPageFromTape***, które zapisują i odczytują pełne strony danych, aktualizując statystyki operacji I/O. Strony są ładowane do obiektu RAM, który przechowuje aktualny stan pamięci operacyjnej, umożliwiając zarządzanie rekordami i stronami w RAM. Funkcje ***InsertPageIntoRAM***, ***RemovePageFromRAM***, oraz ***GetPageFromRAM*** zarządzają dostępem do poszczególnych stron. Program umożliwia także monitorowanie liczby operacji odczytu i zapisu, co pozwala na analizę efektywności algorytmu w ograniczonych zasobach pamięci RAM.

3.2 Typ rekordu

W programie rekord jest reprezentowany jako punkt w dwuwymiarowej przestrzeni, opisany przez współrzędne (x, y). Dodatkowo, każdy rekord posiada wartość klucza, który jest wykorzystywany jako kryterium sortowania. Klucz jest obliczany jako odległość punktu od środka układu współrzędnych, co pozwala na naturalne uporządkowanie rekordów według ich położenia względem punktu $(0, 0)$.

Odległość punktu (x, y) od środka układu współrzędnych jest wyznaczana na podstawie wzoru:

$$\text{key} = \sqrt{x^2 + y^2}$$

gdzie x i y są współrzędnymi punktu.

3.3 Format plików dyskowych

W implementacji programu wszystkie pliki pośrednie, w tym początkowe runy oraz pliki wynikowe z kolejnych faz scalania, są przechowywane na dysku w formacie binarnym. Wybór formatu binarnego zamiast tekstowego jest podyktowany względami wydajnościowymi – pliki binarne zajmują mniej miejsca na dysku oraz pozwalają na szybsze operacje odczytu i zapisu, co jest szczególnie istotne przy pracy z dużymi zbiorami danych i ograniczoną pamięcią RAM.

Każdy rekord przechowywany w pliku binarnym zawiera współrzędne punktu `(x, y)` oraz klucz `key`, będący odległością punktu od środka układu współrzędnych. Do zarządzania tymi danymi program wykorzystuje strumienie binarne, co umożliwia efektywne wykonywanie operacji I/O oraz minimalizację czasu potrzebnego na odczyt i zapis danych.

Podczas sortowania zewnętrznego, program wielokrotnie odczytuje i zapisuje runy oraz pliki wynikowe na dysku. Każda faza sortowania generuje nowe pliki pośrednie, które również

są zapisywane w formacie binarnym. Dzięki temu program może łatwo odczytywać całe strony danych, zapisywać przetworzone rekordy oraz zarządzać dostępem do dużych zbiorów danych, które przekraczają dostępny rozmiar pamięci RAM.

Format binarny pozwala również na precyzyjne zarządzanie pozycją w pliku, co umożliwia szybki dostęp do określonych stron lub rekordów w trakcie faz scalania. Dzięki temu program efektywnie symuluje operacje dostępu do danych na dysku, odzwierciedlając rzeczywiste procesy używane w systemach baz danych przy pracy z dużymi zbiorami danych.

4 Plik konfiguracyjny

Program wykorzystuje plik konfiguracyjny `appsettings.json`, który umożliwia dynamiczne dostosowanie kluczowych parametrów bez potrzeby modyfikacji kodu źródłowego. Plik ten pozwala na definiowanie wartości związanych z rozmiarem strony, dostępną pamięcią RAM, liczbą rekordów oraz innymi ustawieniami wpływającymi na działanie programu. Dzięki temu użytkownik może łatwo dostosować środowisko symulacji do przeprowadzenia eksperymentów.

4.1 Najważniejsze parametry

Poniżej przedstawiono trzy kluczowe parametry konfiguracyjne, które mają największy wpływ na działanie programu:

- **PageSizeInNumberOfRecords** – Określa liczbę rekordów, które mogą być przechowywane w jednej stronie pamięci. Ten parametr wpływa na liczbę operacji wejścia/wyjścia, ponieważ większy rozmiar strony zmniejsza częstotliwość odczytów i zapisów, ale wymaga więcej pamięci RAM.
- **RAMSizeInNumberOfPages** – Ustawia rozmiar dostępnej pamięci RAM wyrażony w liczbie stron. Dzięki temu parametr pozwala symulować ograniczoną pamięć operacyjną, co jest kluczowe w kontekście algorytmów sortowania zewnętrznego.
- **NumberOfRecordsToGenerate** – Określa liczbę rekordów, które mają być wygenerowane losowo. Użytkownik może ustawić tę wartość, aby przetestować efektywność algorytmu przy różnych rozmiarach zbiorów danych.

4.2 Przykład pliku konfiguracyjnego

Przykładowa zawartość aktualnego pliku konfiguracyjnego `appsettings.json` wygląda następująco:

```
{
  "Settings": {
    "PageSizeInNumberOfRecords": 10,
    "RecordSizeInBytes" : 24,
    "RAMSizeInNumberOfPages": 1001,
```

```
    "DataSource": "GenerateRandomly",
    "FilePath": null,
    "NumberOfRecordsToGenerate" : 100000000,
    "LogLevel": "Detailed"
  }
}
```

4.3 Opis parametrów konfiguracyjnych

Poniżej znajduje się opis wszystkich parametrów dostępnych w pliku `appsettings.json`:

- **PageSizeInNumberOfRecords** – Liczba rekordów, które mogą być przechowywane w jednej stronie pamięci.
- **RecordSizeInBytes** – Rozmiar pojedynczego rekordu w bajtach. Parametr ten jest używany do wyliczenia rozmiaru strony w pamięci w bajtach.
- **RAMSizeInNumberOfPages** – Rozmiar pamięci RAM wyrażony w liczbie stron, co pozwala symulować środowisko o ograniczonej pamięci.
- **DataSource** – Źródło danych do sortowania. Może przyjąć wartości:
  - `"GenerateRandomly"` – generowanie danych losowo,
  - `"Manual"` – wprowadzanie danych ręcznie z klawiatury,
  - `"File"` – wczytywanie danych z pliku.
- **FilePath** – Ścieżka do pliku wejściowego z danymi testowymi, jeśli `DataSource` jest ustawione na `"File"`. Dla wartości `"GenerateRandomly"` ścieżka może pozostać pusta (ustawiona na `null`).
- **NumberOfRecordsToGenerate** – Liczba rekordów, które mają być wygenerowane losowo. Parametr ten określa rozmiar zbioru danych do przetworzenia.
- **LogLevel** – Poziom szczegółowości logów generowanych przez program. Może przyjąć wartość `"Detailed"` dla pełnej szczegółowości, umożliwiając śledzenie liczby operacji odczytu i zapisu oraz faz sortowania.

Konfiguracja programu za pomocą pliku `appsettings.json` pozwala na elastyczne dostosowanie parametrów, co jest szczególnie istotne podczas przeprowadzania testów i eksperymentów. Użytkownik może łatwo zmieniać wartości kluczowych parametrów, takich jak liczba rekordów czy rozmiar pamięci, co umożliwia lepsze zrozumienie wpływu tych parametrów na działanie algorytmu sortowania zewnętrznego.

5 Analiza wzorów kosztów I/O w sortowaniu z wielkimi buforami

W celu oszacowania liczby operacji wejścia-wyjścia (I/O) wymaganych do posortowania dużego zbioru danych przy użyciu algorytmu sortowania z wielkimi buforami, stosujemy dwie fazy:

1. **Faza 1** – tworzenie początkowych posortowanych runów,
2. **Faza 2** – faza scalania, w której runy są sukcesywnie łączone, aż powstanie jeden posortowany zbiór.

5.1 Wzory opisujące liczbę operacji I/O

5.1.1 Faza 1 (Tworzenie runów początkowych)

W tej fazie odczytujemy kolejne porcje danych z pliku wejściowego, sortujemy je i zapisujemy jako runy początkowe na dysku. Zakłada się, że w tej fazie dostępne są wszystkie n stron pamięci RAM, co pozwala na wykorzystanie pełnej pojemności RAM-u.

Wzór na liczbę operacji I/O dla Fazy 1 jest wyrażony jako:

$$\text{Liczba operacji I/O} = \frac{2N}{b}$$

gdzie:

- N to liczba wszystkich rekordów,
- b to liczba rekordów na stronę (rozmiar bufora w rekordach).

Każdy rekord jest odczytywany raz i zapisywany raz jako część posortowanego runu, stąd współczynnik 2 w liczniku.

5.1.2 Faza 2 (Scalanie)

Podczas fazy scalania kolejne runy są ładowane do pamięci RAM, gdzie są łączone. W tej fazie jednak, aby zapewnić miejsce na stronę wyjściową, używamy $n-1$ stron do ładowania runów, podczas gdy jedna strona jest zarezerwowana jako bufor wyjściowy.

Zakładając, że po pierwszym cyklu scalania długość runów rośnie około n-krotnie, a liczba runów maleje proporcjonalnie, możemy wyrazić liczbę cykli w następujący sposób:

$$\text{Liczba cykli} = \left\lceil \log_n \left(\frac{N}{b} \right) \right\rceil - 1$$

gdzie n oznacza efektywną liczbę stron wejściowych dostępnych do scalania, czyli $n-1$, ponieważ jedna strona musi być przeznaczona na bufor wyjściowy. Praktyka pokazuje, że przy $n \gg 1$, różnica między n a $n-1$ staje się pomijalna, co pozwala na uproszczenie wzoru

w przybliżeniu. Jednakże, przy mniejszych wartościach n, uwzględnienie $n-1$ może być kluczowe dla precyzyjnych obliczeń.

Każdy cykl scalania wymaga odczytania i zapisania $2 \times \frac{N}{b}$ operacji I/O, ponieważ każdy rekord jest odczytywany i zapisywany z powrotem do pliku wyjściowego.

Zatem całkowity koszt operacji I/O dla Fazy 2 wynosi:

$$\text{Liczba operacji I/O w Fazie } 2 = 2 \times \frac{N}{b} \times \log_n \left(\frac{N}{b} \right)$$

5.1.3 Wzór łączny

Podsumowując, całkowity koszt operacji I/O dla obu faz można przedstawić wzorem:

$$\text{Całkowita liczba operacji I/O} = 2\frac{N}{b} + 2\frac{N}{b} \log_n \left(\frac{N}{b} \right)$$

gdzie pierwszy składnik reprezentuje koszt Fazy 1, a drugi koszt Fazy 2.

6 Eksperyment i analiza wyników

Na przygotowanym programie został przeprowadzony eksperyment, w którym:

- Określono liczbę faz sortowania oraz liczbę operacji odczytów i zapisów stron dyskowych w zależności od liczby losowanych rekordów. Przeprowadzono testy dla pięciu różnych rozmiarów zbiorów danych, znacząco różniących się od siebie liczebnością rekordów.
- Wyznaczono teoretyczną liczbę faz oraz teoretyczną liczbę operacji dyskowych dla badanych wielkości zbiorów danych, stosując wyprowadzone wzory.
- Uzyskane wyniki zostały przedstawione na wykresie funkcji, który ilustruje zależność liczby operacji I/O od liczby rekordów (wykres przedstawiony poniżej). Na obu osiach wykresu zastosowano odpowiednią skalę (logarytmiczną dla lepszego zobrazowania różnic między wynikami praktycznymi a teoretycznymi).
- Porównano uzyskane wyniki praktyczne i teoretyczne, próbując wyjaśnić dostrzeżone rozbieżności.

N	**Total I/O**	**Teoretyczne total I/O**	**Cykle**	**Wyznaczone cykle**
700k	16,400	16,531	2	2
1 mln	23,442	24,328	2	2
10 mln	234,378	289,271	2	2
50 mln	1,562,504	1,607,000	3	3
70 mln	2,187,504	2,296,924	3	3

Table 1: Porównanie liczby operacji dla różnych rozmiarów zbiorów danych.

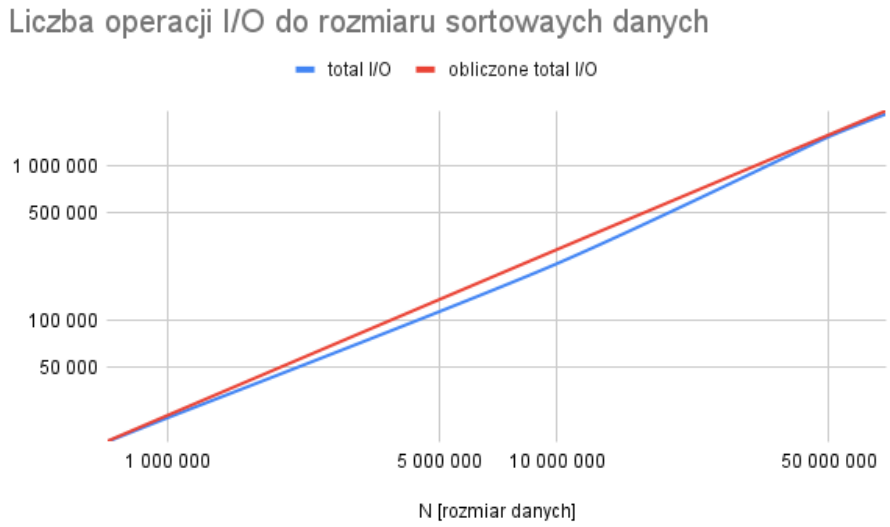

Figure 1: Wykres przedstawiający zależność liczby operacji I/O od liczby rekordów dla wyników praktycznych i teoretycznych (zastosowana skala logarytmiczna na obu osiach)

6.1 Analiza wyników

Porównanie liczby operacji I/O

Z przedstawionej tabeli i wykresu wynika, że liczba rzeczywistych operacji I/O zawsze jest **mniejsza lub równa liczbie teoretycznych operacji I/O**. Wynika to z charakterystyki wzoru i praktycznej realizacji procesu sortowania. Według wzoru, rozmiar kolejnych runów (wyjściowych porcji danych) zwiększa się proporcjonalnie do $n^2 \cdot b$, gdzie n to liczba taśm wejściowych, a b to rozmiar strony w liczbie rekordów. Jednak w praktyce to założenie nie zawsze dokładnie oddaje realne zwiększanie się runów, co powoduje, że liczba rzeczywistych operacji I/O pozostaje poniżej teoretycznego wykresu.

Optymalizacja przy ostatnim runie

Jednym z czynników wpływających na zmniejszenie liczby operacji I/O w praktyce jest **optymalizacja w końcowej fazie sortowania**, gdy pozostaje tylko jeden run. W takiej sytuacji, zamiast wykonywać pełny cykl merge'owania, run ten jest po prostu kopiowany do nowego pliku wyjściowego. Takie podejście eliminuje zbędne operacje scalania, co redukuje całkowitą liczbę operacji I/O, szczególnie dla dużych zbiorów danych.

Wnioski z wykresu

Na wykresie widać, że:

1. Zarówno liczba rzeczywistych, jak i teoretycznych operacji I/O rośnie **logarytmicznie** wraz z liczbą rekordów N.

2. Różnica między teoretyczną a rzeczywistą liczbą operacji jest niewielka, ale systematyczna, co wynika z opisanych wcześniej optymalizacji oraz specyficznych warunków implementacji algorytmu.

Podsumowanie

Wyniki te pokazują, że teoretyczne oszacowania są dobrą podstawą do analizy algorytmu, ale rzeczywiste implementacje mogą być bardziej efektywne dzięki optymalizacjom, takim jak pominięcie scalania w ostatnim etapie. W praktyce liczba operacji I/O pozostaje bliska teoretycznym obliczeniom, co potwierdza efektywność zastosowanego podejścia w realizacji algorytmu sortowania zewnętrznego.