Indeksowa organizacja pliku

Dominik Lau (188697)

23 października 2023

1 Wprowadzenie

Do zaimplementowania wybrałem indeksowo-sekwencyjną organizację pliku Wylosowane przeze mnie typy rekordów to **numery rejestracyjne samochodów**. Implementacji dokonałem w języku C++.

2 Opis działania

2.1 Struktura kodu

W moim rozwiązaniu stosuję:

- klasy rekordów indeksu i pliku z danymi, udostępniające narzędzia do serializacji
- klasę *File* z pliku *generic/File.h* zawierającą implementację pliku dyskowego o dostępie blokowym, która obsługuje logikę zapisu/odczytu stron z pliku dyskowego tak,że dostęp do niego poprzez metody *get* czy *insert* są na podobieństwo dostępu do zwykłej tablicy w pamięci RAM. Struktura ta cache'uje strony, odczytuje i zapisuje tylko w przypadku konieczności jej wymiany (ponadto zapis następuje tylko w przypadku modyfikacji "bitu" *dirty* na true)
- klasę *IndexedFile* udostępniającą m.in. metody *remove, insert, update, reorganise* (o których więcej w następnym podpunkcie)
- folder *cli/* zawierający "agenty" - *InteractiveAgent, RandomAgent, FileAgent* definiujące źródło wejścia do programu, odpowiednio z wiersza poleceń, losowe oraz z pliku
- folder *time/* zawierający zegary i klasy pomiarowe do zliczania liczby operacji dyskowych

2.2 Serializacja/deserializacja

Rekordy z pliku indeksowego składają się z dwóch 4-bajtowych pól (numer strony i klucz), których kolejne bajty umieszczam w pliku zgodnie z konwencją little endian. W przypadku rekordu z danymi mamy trzy pola do serializacji - klucz, wskaźnik na następny rekord w obszarze przepełnienia (jeśli następny taki rekord nie znajduje się w obszarze przepełnienia, to pole to ma wartość **0xFFFFFFFF**) oraz **7-bajtowe pole z danymi**.

2.3 Operacje na pliku indeksowym

- tworzenie (konstruktor) - oferuje możliwość podania ilości stron głównych do stworzenia, ilość stron przeznaczonych na obszar przepełnienia to $\lceil$il. stron głównych $* 0.2\rceil$. **Na początku działania programu tworzony jest plik o 3 stronach głównych** (a co za tym idzie jednej stronie na obszar przepełnienia).
- insert - wstawienie nowego rekordu - pierw algorytmem binary search **przeszukujemy indeks** w poszukiwaniu odpowiedniego bloku, następnie **przeszukujemy blok w poszukiwaniu poprzednika** wstawianego klucza, jeśli nie możemy wstawić rekordu bezpośrednio za poprzednikiem (np. już jest tam jakiś rekord lub strona jest pełna), **umieszczamy go w łańcuchu przepełnienia tego rekordu**
- reorganise
  1. tworzymy tymczasowy plik indeksowy z indeksami *temp_ index i temp_ data*, z których plik data ma ilość głównych stron zgodną ze wzorem $\lceil \frac{N+V}{b \cdot \alpha} \rceil$, gdzie N, V - ilość odpowiednio rekordów głównych i rekordów w przepełnieniu, b - ilość rekordów danych na stronę, α - średnie zapełnienie strony po reorganizacji plik tymczasowy index ma odpowiednią ilość stron tak, żeby zmieścić wskaźniki do tych wszystkich stron
  2. **odwiedzamy kolejne rekordy zgodnie z kolejnością rosnących kluczy** (czyli bierzemy też po uwagę obszar przepełnienia) i umieszczamy je na kolejnych stronach nowego pliku respektując przy tym α

3. zmieniamy pliki *temp_index i temp_data* na *data i index*, w ten sposób dane po reorganizacji przypisujemy do pliku indeksowego, na którym operujemy

- remove - znalezienie odpowiedniego miejsca w strukturze i usunięcie z niego rekordu, tutaj na miejsce, w którym był ten plik "wciągamy" następny z łańcucha przepełnienia
- update - **jeśli chcemy aktualizować klucz** dla danego rekordu to najpierw usuwamy go z pliku a następnie wstawiamy na nowo ze zmodyfikowanymi danymi; **jeśli aktualizujemy tylko dane** to pobieramy pozycję rekordu o danym kluczu i zmieniamy jego numer rejestracyjny

3 Prezentacja wyników programu

3.1 Menu wyboru źródła danych

Po włączeniu programu użytkownik ma do wyboru trzy tryby

```
[INFO] debug mode false
CHOOSE INPUT TYPE: CLI/FILE/RANDOM, set/unsets DEBUG
```

- CLI - tryb interaktywny
- FILE <nazwa_pliku> - dane z pliku o podanej nazwie
- RANDOM <N> - generacja losowej ilości poleceń (INSERT/REMOVE/UPDATE)
- DEBUG - włączenie wypisywania stanu pliku co każdą operację

Po zakończeniu dla każdego z tych plików użytkownik ma możliwość wypisania końcowej postaci pliku.

3.2 Menu interaktywne

Menu oferuje następujące komendy

```
>>HELP
INSERT <key> <value>
REMOVE <key>
UPDATE <key> <newKey> <newValue>
REORGANISE
INORDER
EXIT
```

komenda INORDER oferuje wypisanie pliku zgodnie z kolejnością kluczy, oto przykładowe wywołanie komendy INSERT z wyłączonym trybem debug

```
>>INSERT 3 GS2138
[Measurement] r: 1 w: 0 io(r+w): 1
```

Otrzymujemy informacje o wykonanych zapisach i odczytach (0 zapisów, ponieważ rekord jest narazie w cache'u, 1 odczyt ponieważ strona z pliku indeksowego najwyraźniej nadal jest w cache).

3.3 Wypisywanie pliku

Są dwa sposoby, w jaki program wypisuje plik, pierwszy - wypisanie struktury pliku razem z pustymi miejscami i zaznaczeniem jaki rekord jest na której stronie, oto przykład

```
___INDEXED__FILE___
-------INDEX-------
=======PAGE0======
page: 0 key: 0
page: 1 key: 98852148
page: 2 key: 198206151
-----INDEX-END-----

------PRIMARY------
```

```
=======PAGE0======
#0 key: 0 data: BRUUUUH ptr: 14
#1 key: 68030329 data: DEBUG69
#2 ********************
#3 ********************
=======PAGE1======
#4 key: 98852148 data: 4O353I3
#5 ********************
#6 ********************
#7 ********************
=======PAGE2======
#8 key: 198206151 data: P5MP165
#9 ********************
#10 ********************
#11 ********************
----PRIMARY-END----

-----OVERFLOW------
=======PAGE0======
#12 key: 68030327 data: BUP30QP ptr: 13
#13 key: 68030328 data: DEBUG70
#14 key: 35496734 data: K7302PD ptr: 12
#15 ********************
----OVERFLOW-END---
```

---

a oto wypisanie tego samego pliku zgodnie z kolejnością klucza

---

```
___INDEXED__FILE___
#0 key: 0 data: BRUUUUH
#14 key: 35496734 data: K7302PD
#12 key: 68030327 data: BUP30QP
#13 key: 68030328 data: DEBUG70
#1 key: 68030329 data: DEBUG69
#4 key: 98852148 data: 4O353I3
#8 key: 198206151 data: P5MP165
```

---

4 Eksperyment

4.1 Szczegóły implementacyjne

Kod przeprowadzonego eksperymentu umieściłem w pliku *perf2.cpp* jako część biblioteki *sbd_test*. Test uruchamiany jest za pomocą frameworka do testowania gtest. W celu zliczania ilości operacji wejścia-wyjścia w *libsbd* zdefiniowałem dwa zegary: *writeClock*,oraz *readClock*. W pomiarach wykorzystuję również klasę *Measurement*, która zbiera pomiary na wzór paradygmatu RAII - w konstruktorze zapisywany jest aktualny stan zegara a w destruktorze nowy stan zegara jest odejmowany od starego, w ten sposób otrzymuję liczbę wywołań funkcji *tick* danego zegara. Ponadto pomiary te dodawane są do obiektu klasy *MeasurementAggregate*, który umożliwia policzenie wartości średniej z zebranych próbek.

W przypadku testów przyjęto $b = 50$ rekordów.

4.2 Rozmiar a ilość rekordów

Pierwszym przeprowadzonym elementem było zbadanie zależności między ilością rekordów a rozmiarem pliku. W tym celu przyjęto $\alpha = 0.5$ i przeprowadzono N losowych operacji INSERT.

N	rozmiar indeksu [B]	rozmiar obszaru głównego [B]	rozmiar obszaru przepełnienia [B]
10	?	?	?
50	?	?	?
100	?	?	?
500	?	?	?
1000	?	?	?
5000	?	?	?
10000	?	?	?

TODO: wykres
TODO: omówienie

4.3 Rozmiar a α

W następnym eksperymencie przeprowadzono 1000 losowych operacji INSERT dla różnych współczynników średniego zapełnienia bloku i zbadano końcowy rozmiar pliku

α	rozmiar indeksu [B]	rozmiar obszaru głównego [B]	rozmiar obszaru przepełnienia [B]
0.1	?	?	?
0.1	?	?	?
0.1	?	?	?
0.1	?	?	?
0.1	?	?	?
0.1	?	?	?
0.1	?	?	?
0.1	?	?	?

TODO: wykres
TODO: omówienie

4.4 Ilość operacji

W kolejnym eksperymencie wstawiono wygenerowano 1000 losowych rekordów, przyjęto $\alpha = 0.5$ a następnie badano działanie różnych przypadków aktualizacji/usuwania.

operacja	ilość operacji	średnia liczba zapisów	średnia liczba od czytów	średnia liczba operacji i/o
wstawianie	1000	?	?	?
usuwanie ist.	100	?	?	?
usuwanie nieist.	100	?	?	?
aktualizowanie ist.	100	?	?	?
aktualizowanie nieist.	100	?	?	?

TODO: wykres
TODO: omówienie

5 Podsumowanie