Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

М.Ф. Копитко

Основи інформаційних технологій

Тексти лекцій

Львів
Видавничий центр ЛНУ імені Івана Франка
2007

УДК [004.08+004.42+004.6](042.4)
ББК З 973я73-2+3988я73-2
К-65

Рецензенти:

Д-р. фіз.-мат. наук, проф. М.М.Притула
(Львівський національний університет імені Івана Франка)

Д-р. фіз.-мат. наук, проф. Р.О.Слоньовський
(Національний університет «Львівська політехніка»)

Рекомендовано до друку Вченою радою
факультету прикладної математики та інформатики
Протокол № 4 від 21.09.2005р.

Рекомендовано до друку Вченою радою
Львівського національного університету імені Івана Франка
Протокол № 35/9 від 28.09.2005р.

Копитко М.Ф.
К-65 Основи інформаційних технологій: Тексти лекцій. − Львів: Видавничий центр ЛНУ ім. Івана Франка, 2007.− 259 с.

Викладено основи інформаційних технологій: вирішення проблем зберігання інформації, представлення елементів даних у блоках і файлах, організація структур індексів для ефективного доступу до даних.

Для студентів, аспірантів та співробітників університету, програмістів, які створюють сучасні інформаційні технології.

© Копитко М.Ф.,2007

ВСТУП

Інформаційні технології – це сукупність методів і програмно-технічних засобів, об'єднаних у технологічний ланцюжок, що забезпечує збирання, оброблення, зберігання, поширення та відображення інформації з метою зменшення трудомісткості процесів використання інформаційного ресурсу, а також підвищення їхньої надійності та оперативності.

За роки розвитку технічних засобів комп'ютера і відповідного програмного забезпечення виникло стільки інформаційних технологій, скільки потрібно, щоб охопити всі сторони життєдіяльності людини. Залежно від того, яку інформацію вони обробляють або де їх використовують, інформаційні технології отримують назви: Internet-технології, WWW-технології, офісні інформаційні системи, мережеві технології, геоінформаційні технології, технології баз даних тощо. Кожен з цих напрямів використання вивчають студенти в базових і спеціальний курсах. Тож що є основою усіх цих технологій?

На думку автора, в основі інформаційних технологій, передусім найскладніших, якими сьогодні є системи керування базами даних, виникають такі проблеми: як зберігати великі об'єми інформації; як змістовно формувати інформацію в записи і блоки; як ефективно доступитися до інформації, враховуючи її великі об'єми. Саме висвітленню цих проблем присвячено тексти лекцій.

У першому розділі розглянуто проблеми зберігання інформації. Значну увагу приділено пристроям, призначеним для зберігання інформації, передусім дискам. Оцінено швидкодію алгоритмів обробки даних, які необхідно переміщати між оперативною пам'яттю і вторинними або третинними пристроями зберігання. На прикладі алгоритму сортування двофазним багатокомпонентним злиттям проілюстровано, як використовувати різні прийоми пониження часових затрат на операції читання даних з диска і записування даних на диск. Розглянуто також методи підвищення надійності дискових систем, які дають змогу подолати періодичні помилки читання-запису, а також зберегти дані після поломки одного чи декількох дисків.

Другий розділ присвячено питанням зберігання окремих елементів даних – відношень, кортежів, значень атрибутів і рівноцінних щодо них сутностей, які застосовують в інших моделях представлення інформації, – згідно з вимогами блочної моделі дискових даних. Тобто проілюстровано відображення атрибутів як байтових послідовностей (їх називають полями) постійної або змінної довжини. Як поля, у свою чергу, об'єднуються в набори даних постійної або змінної довжини (їх називають записами) і відповідають кортежам або об'єктам. Зазначено як записи зберігаються у фізичних блоках. Для формування блоків використано різні підходи, корисні, передусім, при модифікації даних.

У третьому розділі розглянуто проблеми використання різноманітних структур даних для побудови індексів з метою забезпечення ефективного доступу до інформації. На конкретних прикладах проілюстровано як будуються індекси для послідовних файлів, вторинні індекси для невпорядкованих файлів, B-дерева для організації індексів для довільних файлів, геш-таблиці для представлення індексів.

Четвертий розділ присвячено поняттю багатовимірних даних, які використовують в геоінформаційних системах і системах підтримки прийняття рішень. У ньому розглянуто особливості побудови індексів для швидкого доступу до просторових даних. Обговорюються характеристики сіткових файлів, роздільних геш-функцій, індексів з декількома ключами, kd-дерев, дерев квадрантів, R-дерев.

Протягом усього видання виклад матеріалу супроводжується прикладами і вправами для самостійного виконання.

ПРИНЦИПИ ЗБЕРІГАННЯ ІНФОРМАЦІЇ

У лекціях йдеться про те, як *системи керування базами даних* (СКБД) справляються з великими об'ємами інформації. Проблема зводиться до двох основоположних питань:

1) як комп'ютерна система зберігає дані і керує ними;
2) які форми представлення і структури даних найкраще сприяють підвищенню ефективності операцій з обробки даних?

Нижче ми опишемо пристрої, які використовують для зберігання крупних масивів інформації (зокрема, диски), розглянемо ієрархію пристроїв пам'яті і продемонструємо, як швидкодія алгоритмів обробки даних залежить від способів переміщення порцій інформації між оперативною пам'яттю і вторинними (зазвичай, дисками) або навіть третинними пристроями зберігання. Звернемо увагу на різні прийоми зниження часових затрат на операції читання даних з диска та їхнє записування. Також обговоримо методи підвищення надійності дискових систем, які дають змогу подолати помилки читання-записування і навіть випадки цілковитої «відмови» диска.

Розпочнемо з прикладу, який демонструє особливості деякої вигаданої СКБД, в основу якої покладено неправильні проектні рішення, що не відповідають загальноприйнятій практиці.

Тема 1. Приклад вигаданої системи баз даних

Якщо Ви вже користувались будь-якою СКБД, то, ймовірно, Вам здається, що реалізація подібних систем не надто складна. Назвемо нашу вигадану СКБД Straw. Ця система належить до категорії реляційних, підтримує *SQL* і розповсюджується як окремі версії для багатьох операційних систем, у тім числі *UNIX*.

1.1. Особливості реалізації СКБД Straw

У базовій версії Straw для збереження відношень (інформацію про реляційну модель даних подано у додатку А) використовують файлову систему *UNIX*. Наприклад, відношення *Students (name, id, dept)* можна подати як файл */USR/db/Students*. Кожен запис у

файлі *Students* відповідає одному кортежу. Значення компонентів кортежу зберігаються у рядках, розділених спеціальними символами-маркерами #. Вміст файла */USR/db/Students* може виглядати, наприклад, так:

Smith # 123 # CS

Vasyl Sydorchuk # 522 # EE

…

Опис схеми бази даних розміщується у спеціальному файлі */usr/db/schema*. Кожному відношенню бази даних відповідає один рядок файла *schema*, який розпочинається з назви відношення, далі подано пари імен атрибутів і їхні типи. Всі елементи рядка розділяються позначкою # Файл */usr/db/schema* може зберігати такі рядки:

Students # name # STR # id # INT # dept # STR

Depts # name # STR # office # STR

…

Тут ми вважаємо, що файл *schema* містить опис відношення *Students (name, id, dept)* із рядковими (*STR*) атрибутами *name* і *dept* і цілочисловим (*INT*) атрибутом *id*, а також відношення *Deps(name, office)*.

Приклад 1. Розглянемо звичайний сеанс взаємодії з СКБД Straw. Нам надано право звертатися до користувальницького інтерфейсу Straw, вводячи необхідні команди SQL у відповідь на запрошення (&) системи. Отож запит до бази даних вигляду

& SELECT * FROM Students #

повертає таблицю:

Name	*id*	*dept*
Smith	*23*	*CS*
Vasyl Sydorchuk	*522*	*EE*

СКБД Straw дає змогу зберігати результати виконання запитів у нових файлах: з цією метою текст команди необхідно доповнити

символом вертикальної риски з іменем файла. Наприклад, виконання команди

& SELECT ∗ FROM Students WHERE id >= 500 | *Highid* #

створить файл */usr/db/Highid*, який містить один рядок:

Vasyl Sydorchuk #522 # EE

1.2. Як СКБД Straw виконує запити

Розглянемо типову форму *SQL*-запиту:

SELECT ∗ FROM R WHERE <умова>

У відповідь на подібний запит Straw виконує такі дії:

1) звертається до файла *schema* за інформацією про атрибути відношення R і їхні типи;
2) перевіряє, чи правильно задана <умова> з погляду семантики відношення R;
3) передає на екран комп'ютера заголовок результату запиту з найменуванням атрибутів, підкреслений лінією;
4) зчитує файл, який відповідає відношенню R, і для кожного рядка-запису:
   а) перевіряє <умову>;
   в) якщо <умова> правильна, виводить на екран рядок-кортеж.

Під час виконання команди типу

SELECT ∗ FROM R WHERE <умова> | T

Straw виконує такі дії:

1) опрацьовує запит так, як і раніше, проте пропускає п.3, тобто не генерує заголовка результуючого відношення;
2) зберігає результат у новому файлі */usr/db/T;*
3) додає у файл */usr/db/schema* рядок з описом відношення *T*, подібний до рядка з інформацією про схему відношення R.

Приклад 2. Розглянемо складніший запит, який передбачає з'єднання (*join*) кортежів двох відношень *Students* і *Depts* :

```
SELECT office
FROM Students, Depts
WHERE Students.name = `Smith`  AND
          Studets.dept = Depts.name #
```

Запит змушує систему здійснити з'єднання відношень *Students* і *Depts,* тобто розглянути кожну пару кортежів (по одному з перелічених відношень) і визначити, чи :

1) містять компоненти *Studets.dept* і *Depts.name* один і той самий рядок з назвою факультету;
2) є прізвищем студента (*Students.name*) рядок "Smith".

Алгоритм виконання цього запиту неформально може виглядати так:

FOR (ДЛЯ) кожного кортежу *s* із *Studets* (ВИКОНАТИ) *DO*
    FOR (ДЛЯ) кожного кортежу *d* із *Depts* (ВИКОНАТИ) *DO*
        IF (ЯКЩО) *s* і *d* задовольняють умову *WHERE* (ТОДІ) *THEN*
            відобразити значення компонента *office* кортежу *d*;

1.3. Негативні моменти в системі Straw

Зазвичай, ні для кого не є сюрпризом, що «справжні» СКБД реалізуються зовсім не так, як вигадано Straw. Можна назвати безліч причин, через які наша система не відповідає вимогам, які ставлять у зв'язку зі зростанням об'ємів даних і/або кількістю водночас підключених до системи користувачів. Перерахуємо тільки деякі з багатьох можливих проблем.

- Методика розміщення кортежів на дисках не відзначається гнучкістю, необхідною для виконання операцій модифікації. Якщо, наприклад, в одному з кортежів відношення *Students* необхідно змінити значення компонента *dept* з EE на ECOM, то системі необхідно буде переписати весь файл, зсуваючи кожен наступний символ на дві позиції «вниз».
- Операції пошуку надзвичайно трудомісткі. Система завжди повинна переглядати відношення загалом – навіть у тих випадках, коли (як в запиті з прикладу 2) задано одне чи декілька значень, які дають змогу зосередити увагу тільки на обмеженій групі кортежів.
- Система не передбачає можливості буферизації окремих «корисних» фрагментів даних в оперативній пам'яті; кожного разу, коли виникає потреба в отриманні тієї чи іншої інформації, вони зчитуються з диска знову.
- У системі не передбачено засобів керування паралельними завданнями. Декілька користувачів може водночас

модифікувати один і той самий файл, що може спричинити до непередбачуваних наслідків.

- Система ненадійна. У випадку збоїв дані можна втратити, а операції – перервати.

Згодом ми розглянемо технології, які дають змогу ефективно вирішувати перелічені вище проблеми і які використовують для роботи з великими об'ємами інформації.

Тема 2. Ієрархія пристроїв пам'яті

Типова комп'ютерна система налічує кілька компонентів, призначених для зберігання даних. Компоненти розрізняють за *місткістю* і *швидкістю* доступу до даних, а також *вартістю зберігання* байта інформації. Ступінь відмінностей значень перших двох параметрів оцінюється, щонайменше, у *сім* порядків. Величини вартості зберігання байта інформації в компонентах також відрізняються, проте не так серйозно – вартість найдешевших компонентів на *три порядки* нижча, ніж найдорожчих. Не має нічого дивного в тому, що пристрої найменшої ємності мають найвищі характеристики швидкості доступу до даних і найбільшу вартість зберігання одного байта. Схему ієрархії пристроїв пам'яті комп'ютера наведено на рис. 1.

✓ **Корисно знати.** *Одиниці вимірювання об'ємів пам'яті.* За традицією говорять про місткість комп'ютерних компонентів пам'яті так, ніби її вимірюють степенями числа 10 – мегабайти, гігабайти і т.д. Насправді, пристрої пам'яті, (наприклад, напівпровідниковий чіп) проектують так, що вони зберігають кількість бітів, яка дорівнює степені числа 2. Саме тому, що величина 2^{10}=1024 не дуже відрізняється від тисячі, прийнято використовувати для скороченого запису 2^{10} префікс «кіло», 2^{20} – «мега», 2^{30} – «гіга», 2^{40} – «тера» і 2^{50} – «пета». Ті ж префікси в областях, далеких від інформаційних технологій, означають, відповідно – 10^3, 10^6, 10^9, 10^{12}, 10^{15}. У міру зростання величина невідповідності збільшується, адже 10^6=1 000 000, а один мегабайт –

2^{20}=1 048 576 байт. Для позначення префікса «кіло» використовують стандартне скорочення «К», «мега» – «М», «гіга» –»Г», «тера» – «Т», «пета» – «П». Наприклад, 16 Гбайт – це, строго кажучи, 2^{34} байт.

Для вимірювання часу роботи тих чи інших пристроїв комп'ютера використовують, здебільшого, секунди, помножені на від'ємну степінь числа 10, а саме 10^{-3} с – 1 мілісекунда (мс); 10^{-6} с – 1 мікросекунда (мкс), 10^{-9} с – 1 наносекунда (нс), 10^{-12} с – 1 пікосекунда (пс).

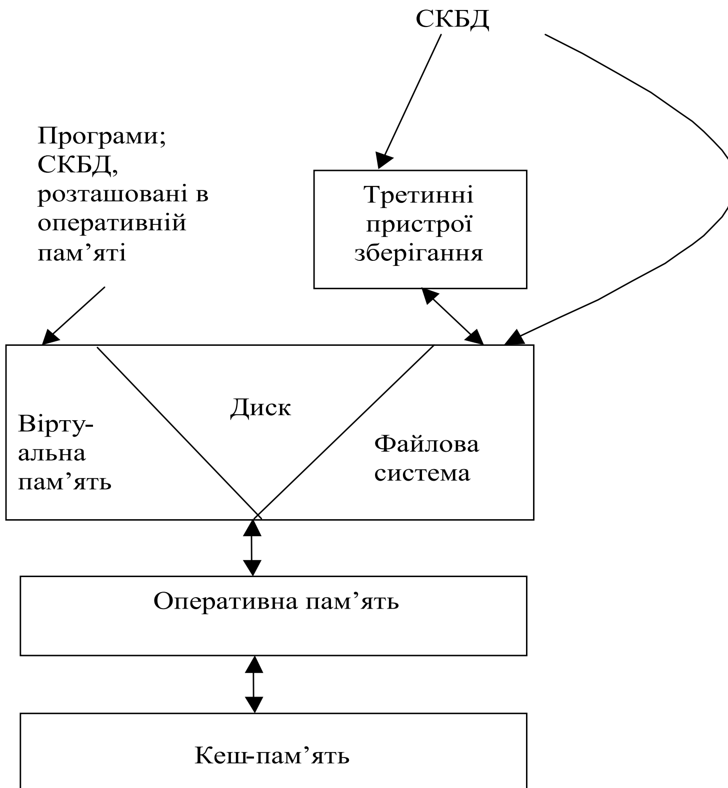


Рис.1. Ієрархія пристроїв пам'яті

2.1. Кеш-пам'ять

На нижньому рівні ієрархії розташована *кеш-пам'ять* (*cache memory*). *Вбудований кеш* (*on-board cache*) перебуває у тому ж *чіпі*, що і мікропроцесор, а додатковий *кеш 2-го рівня* (*level-2 cache*) реалізується як окремий чіп. Елементи даних (у тім числі машинні інструкції) у кеш-пам'яті є копіями вмісту певних ділянок *оперативної пам'яті* (*main memory*). Обмін інформацією між кеш-пам'яттю та оперативною пам'яттю, зазвичай, здійснюється декількома порціями, об'єм яких обчислюється декількома *байтами.* Кеш-пам'ять – це сховище окремих машинних інструкцій, цілочислових величин, чисел з плаваючою комою і коротких символьних рядків.

У комп'ютерах, які виготовлено до 2001 року, місткість пристроїв кеш-пам'яті сягала 1 Мбайт. Час обміну даними між кеш-пам'яттю і процесором визначається частотою процесора й обчислюється декількома наносекундами (10^{-9} с). Проте, час, необхідний для передачі даних з оперативної пам'яті в кеш-пам'ять, і навпаки, значно більший і сягає 100 наносекунд.

2.2. Оперативна пам'ять

Центром подій, які відбуваються у комп'ютері, є *оперативна пам'ять* (*main memory*). Усі дії, які виконуються комп'ютером, так чи інакше стосуються програмних інструкцій і даних, розташованих в оперативній пам'яті (ОП). Комп'ютери, які виготовлено до 2001 року, оснащували оперативною пам'яттю, середній об'єм якої становив близько 100 Мбайт – 10 Гбайт.

Пристрої оперативної пам'яті забезпечують довільний доступ (*random access*) до комірок, а це означає, що на отримання будь-якого байта даних затрачається один і той самий час – зазвичай, це 10–100 наносекунд (10^{-8} – 10^{-7} с).

2.3. Віртуальна пам'ять

Під час виконання програми дані, якими вона оперує (змінні чи значення, зчитані з файла), розміщуються в просторі адрес *віртуальної пам'яті* (*virtual memory*). Чимало комп'ютерів підтримує 32-розрядний адресний простір, тобто можна звертатися

за адресами, загальна кількість яких є 2^{32} (тобто 4 Гбайти). Оскільки простір віртуальної пам'яті значно перевищує розмір оперативної пам'яті звичайного комп'ютера, то більшу частину вмісту віртуальної пам'яті насправді розташовано на дисках. Диск ділиться на логічні одиниці – *блоки*, розміром від 4 Кбайт до 56 Кбайт. Пересилка даних між диском і оперативною пам'яттю в рамках віртуальної пам'яті здійснюється цілими блоками, які в контексті оперативної пам'яті називають сторінками. Апаратне забезпечення та операційна система комп'ютера дають змогу розташовувати сторінки віртуальної пам'яті в довільній частині оперативної пам'яті і забезпечують коректне відображення кожного байта в середині блока адресами віртуальної пам'яті.

Частина схеми на рис. 1, яка стосується віртуальної пам'яті, представляє те, як остання трактується звичайними програмними застосуваннями і не має відношення до способів керування інформацією в системах баз даних. Винятком є СКБД, розташовані в оперативній пам'яті, які насправді керують даними засобами механізму сторінкової пересилки інформації в оперативну пам'ять, що підтримуються на рівні операційної системи. Системи баз даних, розташовані в оперативній пам'яті, як і значна кількість «звичайних» застосувань демонструють найбільшу ефективність у тих випадках, коли розмір даних, якими вони оперують, настільки малий, що дані можуть перебувати в оперативній пам'яті протягом всього сеансу роботи, і необхідності в їхній підкачці (*swapping*) засобами ОС немає.

Зазвичай, для багатьох «звичайних» програм 32-х розрядів адресного простору достатньо. Однак його недостатньо для серйозних застосувань систем баз даних. Керування інформацією в крупномасштабних СКБД не обходиться без участі дисків.

2.4. Вторинні пристрої зберігання

Практично кожен комп'ютер оснащено тими чи іншими вторинними пристроями зберігання (*secondary storage devices*) інформації, які мають значно більшу місткість і меншу швидкодію порівняно з оперативною пам'яттю. У сучасних комп'ютерних системах роль вторинних пристроїв зберігання відіграють *диски* (*disks*) – *магнітний, оптичний, магнітооптичний*. Останній,

зазвичай, дешевший, однак він забезпечує обмежену підтримку режиму запису, отож виконує функцію сховища архівної інформації.

На рис. 1 проілюстровано, що диск виконує підтримку як віртуальної пам'яті, так і файлової системи. Обмін файловими даними між диском і оперативною пам'яттю виконується блоками під керівництвом ОС або СКБД. Переміщення блока з диска в ОП називають читанням дискових даних, а обернену операцію – записом дискових даних. Обидві операції прийнято називати терміном *дискове введення/виведення* (*disk I/O*). Деякі фрагменти оперативної пам'яті використовують для буферизації дискових файлів, тобто для тимчасового зберігання цих файлів. Наприклад, якщо файл відкривається для читання, ОС може зарезервувати в ОП блок розміром 4 Кбайти як буфер. Спочатку в буфер копіюється перший блок файла. Після того, як прикладна програма виконає необхідні дії з опрацювання зчитаних даних, у буфер поміщають наступний блок, який потрапляє на місце попереднього. Процес, проілюстрований на рис. 2, продовжується доти, доки файл не буде оперативно або примусово закрито.

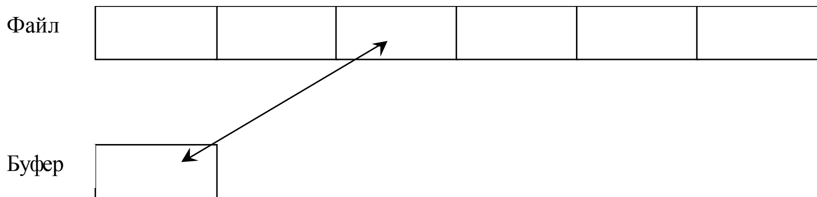


Рис. 2. Файл і відповідний буфер ОП

СКБД керують переміщенням дискових блоків між ОП і вторинними пристроями зберігання самостійно, не посилаючись на засоби менеджера файлів ОС. Функції керування і проблеми, які щодо цього виникають, для них не зникають. Час, необхідний для здійснення типової операції читання чи запису, вимірюється у 10-30 мілісекунд (0,01 – 0,03 с). За цей час сучасний процесор виконує декілька мільйонів машинних інструкцій. Як наслідок виникає ситуація, коли затрати на читання-запис дискового блока значно перевищує ресурс часу, необхідний для опрацювання вмісту

цього блока. Отож життєво важливою є реалізація такої умови: у будь-яких ситуаціях, якщо це можливо, дисковий блок, вміст якого має опрацьовуватися, повинен завчасно бути присутнім в ОП. Якщо цю умову виконано, то системі не треба витрачати дорогоцінний час на трудомісткі операції дискового введення/виведення.

Значення вмісту магнітного диска досягають позначки 100 Гбайт і більше. Оскільки комп'ютер можна оснастити декількома дисковими пристроями, то йдеться про сотні гігабайт, доступних у рамках однієї системи. Хоча вторинна пам'ять на 5 порядків менш продуктивна від оперативної, зате місткість дискових пристроїв у сотні разів вища від об'єму ОП. Питома вартість зберігання одного байта дискових даних значно нижча порівняно з ОП. Ціна в перерахунку на 1 Мбайт місткості дисків коливається від 1-го до 2-х центів, водночас вартість зберігання цього мегабайта в ОП становить 1–2 долари.

2.5. Третинні пристрої зберігання

Деколи не вдається розмістити на вторинних пристроях зберігання (дисках) реальні бази даних. Наприклад, магазини роздрібної торгівлі кожного року накопичують терабайти (10^{12} байт) даних з описом здійснених покупок, а об'єми інформації, яку надсилають супутники, вимірюють петабайтами (10^{15} байт).

Щодо таких випадків розроблено *третинні* пристрої зберігання (*tertiary storage devices*) даних, яким властиві значно нижчі показники продуктивності операцій читання-запису і дешевша питома вартість зберігання даних порівняно з магнітними дисками.

До третинних пристроїв зберігання даних належать:

1. *Магнітні стрічки, які встановлюють за вимогою*. Простий і в минулому єдиний підхід до організації третинних сховищ інформації, що стосується запису даних на бобіни або касети з магнітними стрічками та їхнім розміщенням на стійках-стелажах.
2. *Автоматичні приводи оптичних дисків.* За допомогою робота автоматичний привід обслуговує стійки, на яких розташовані оптичні компакт-диски CD-ROM.

3. *Стрічкові бункери* (*tape silos*). Бункер – це спеціально обладнане приміщення, в якому є стійки з магнітними стрічками. *Робот*, яким керує комп'ютер, встановлює потрібні касети у вільні приводи.

Середня місткість касети зі стрічкою в 2001 році оцінювалась у 50 Гбайт. Місткість стандартного оптичного компакт-диска CD-ROM є 2/3 гігабайта, диски нового покоління (DVD) здатні зберігати до 2,5 Гбайт даних. Стрічкові бункери і масиви дисків дають змогу зберігати терабайти інформації.

Для доступу до заданого елементу даних на носії третинного пристрою необхідно від декількох секунд до декількох хвилин.

Отже, час доступу до даних на носіях третинних пристроїв приблизно у 1 000 разів перевищує аналогічні характеристики вторинних пристроїв (секунди проти мілісекунд). Якщо порівняти середню місткість третинних і вторинних пристроїв, то за цим показником перші перевищують другі у ті ж 1 000 разів (сотні терабайт проти сотень гігабайт).

Графік, наведений на рис. 3, демонструє логарифмічну залежність між показниками швидкості і місткістю пристроїв. Горизонтальна – часова – вісь, градуйована значеннями степеня числа 10: наприклад, -3 означає 10^{-3} секунд. Вертикальна вісь представляє величину місткості в байтах: наприклад, 8 означає 100 Мбайт (10^{8} байт).

2.6. Енергозалежні та енергонезалежні пристрої пам'яті

Додатковий критерій зіставлення різних пристроїв комп'ютерної пам'яті стосується їхньої залежності від зовнішніх джерел енергоживлення. *Енергозалежний* пристрій у вимкненому стані або під час збою живлення «забуває» все, що в ньому зберігалось. Енергонезалежні пристрої, навпаки, здатні підтримувати дані цілими протягом тривалого часу без живлення. Важливість аспекту незалежності від електроживлення очевидне, оскільки одна із ключових характеристик СКБД полягає у здатності підтримувати цілісність даних навіть за умови збоїв і відмов (зокрема тих, що стосуються порушення живлення).

Магнітні матеріали зберігають свій фізичний стан і за відсутності зовнішнього джерела електроенергії. Отож пристрої пам'яті з

магнітними носіями, такі як диски і стрічки, належать до енергонезалежних. Оптичні властивості матеріалів компакт-дисків також не залежать від живлення. Матриця «білих» і «чорних» цяток на поверхні оптичного диска, збережена в момент запису, залишається незмінною. отож практично всі вторинні і третинні пристрої збереження інформації є енергонезалежними.

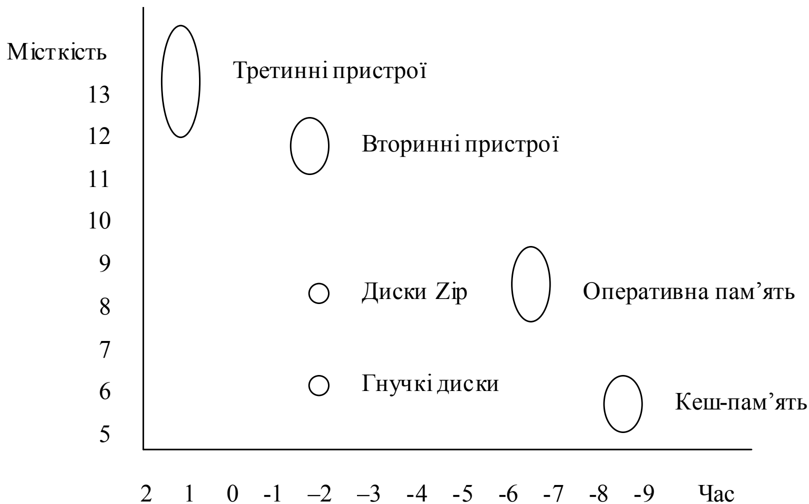


Рис. 3. Залежність значень часу доступу до даних і місткості пристроїв, які належать різним рівням ієрархії пам’яті

Зазначимо, що пристрої оперативної пам’яті, здебільшого, енергозалежні. Система баз даних, яка функціонує на комп’ютері, оснащеному енергозалежною оперативною пам’яттю, повинна невідкладно фіксувати на дисках всі внесені зміни. Інакше у випадку збою електроживлення виникає серйозна небезпека втрати даних. Як наслідок, під час виконання операцій модифікації системі доводиться звертатися до диска безліч разів.

Тема 3. Диски

Те, як СКБД використовують вторинні пристрої зберігання, значною мірою визначає життєво важливі функціональні характеристики цих систем. Роль вторинних пристроїв зберігання

практично належить магнітним дискам. Отож перед ознайомленням з технологіями обробки інформації доцільно детально розглянути внутрішню будову дискового пристрою.

3.1. Внутрішні механізми дискових накопичувачів

На рис. 4 проілюстровано головні рухомі елементи дискового приводу – *блок дисків* (*disk assembly*) і *блок головок* (*head assembly*). Блок дисків налічує одну чи декілька круглих *пластин*, які обертаються навколо центрального шпинделя.

Верхня і нижня поверхні пластин покриті тонким шаром магнітного матеріалу, призначеного для зберігання бітів інформації. Звичайний діаметр пластин диска становить 3,5 дюйма = 8,80 см (1 дюйм = 2,54 см; 1 фут = 12 дюймів = 30,48 см), хоча випускаються диски з пластинами діаметром від 1 дюйма до декількох футів.

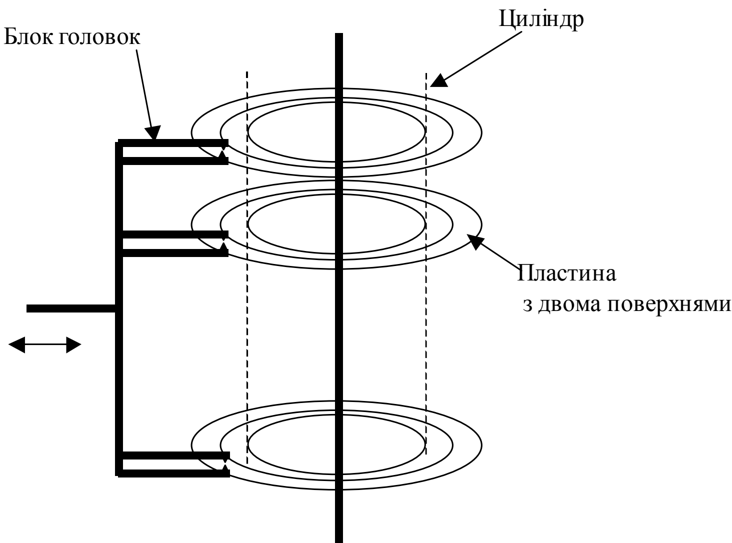


Рис. 4. Схема внутрішньої будови приводу магнітних дисків

Ділянки намагніченої поверхні пластин організовані у формі *доріжок* (*tracks*), які є концентричними колами. Доріжки займають майже всю поверхню, за винятком середини (рис. 5). Вони розбиті

на *сектори* за допомогою *зазорів* (які не є намагніченими). Сектор є неподільною одиницею місткості диска – як і з погляду операцій зчитування та запису, так і що стосується стійкості пристрою до можливих пошкоджень. Зазори використовують для ідентифікації початку секторів. Сумарна площа зазорів становить близько 10% від площі доріжки. Дискові блоки (про які говорилося вище) є логічними одиницями виміру об'ємів фрагментів даних, які переміщаються між диском і оперативною пам'яттю, і складаються з одного або декількох секторів.

Тепер визначимо, що таке блок головок. Головки закріплені на важелі, який рухається поступово. Кожній поверхні пластин відповідає власна головка, яка рухається дуже близько від поверхні, проте не доторкається до неї. Головка здатна визначити заряд елементів магнітного шару, тобто зчитувати біти даних, і змінювати заряд елементів з метою збереження інформації. Якщо головка випадково доторкнеться до поверхні пластини, то поверхня зруйнується і дисковий пристрій зіпсується.

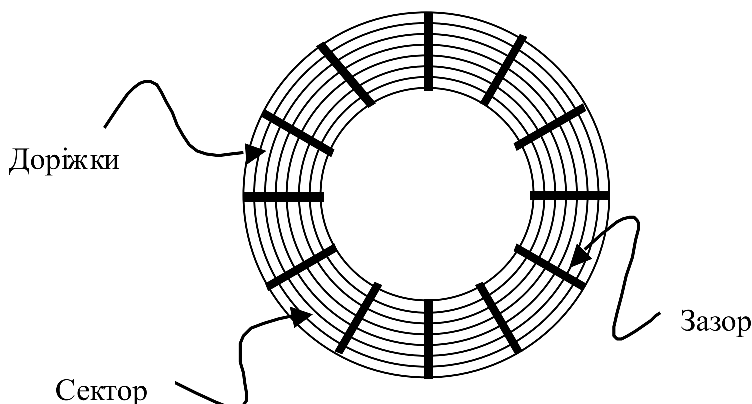


Рис. 5. Структура поверхні пластини диска: вигляд зверху

✓ **Корисно знати.** *Сектори і блоки.* Нагадаємо, що сектор – це фізична структурна одиниця поверхні диска, а блок – логічна одиниця об'єму дискових даних, яку використовують для передачі

інформації від диска до ОС або до СКБД, і навпаки. Ми вже говорили, що звичайний блок може відповідати за об'ємом одному або декільком секторам, хоча немає нічого неприродного в тому, якщо величина блока виявиться меншою від місткості сектора і в секторі зможуть «поміститися» декілька блоків (у деяких застарілих системах використано саме такий підхід).

3.2. Контролер дисків

Одним або декількома дисковими пристроями комп'ютера керує *контролер дисків* (*disk controller*) – невеликий спеціалізований процесор. Опишемо його функції:

1. Керування електромеханічним виконавчим приводом, який переміщає блок головок на необхідну лінійну величину так, щоб поряд з кожною головкою виникла певна доріжка відповідної поверхні пластини, доступна для читання або запису даних. Сукупність доріжок, які перебувають водночас під/над кожною головкою, називають *циліндром* (*cylinder*).
2. Вибір поверхні пластини і визначення сектора цієї поверхні з метою здійснення операцій зчитування чи запису інформації.
3. Передача зчитаних з сектора бітових даних в оперативну пам'ять і запис у необхідний сектор тих даних, які отримано з оперативної пам'яті.

На рис. 6. проілюстровано схему простої однопроцесорної комп'ютерної системи.

Процесор за допомогою *шини даних* взаємодіє з оперативною пам'яттю і контролером дисків. Контролер дисків може одночасно керувати декількома дисковими пристроями (на рис. 6 їх є три).

3.3. Загальні параметри дисків

Потреби ефективного збереження постійно зростаючих об'ємів інформації вимагають вдосконалення технології виготовлення дискових пристроїв. Перелічимо головні критерії оцінки і середнє значення, що відповідає зразкам пристроїв, які випускали в 2001 р.

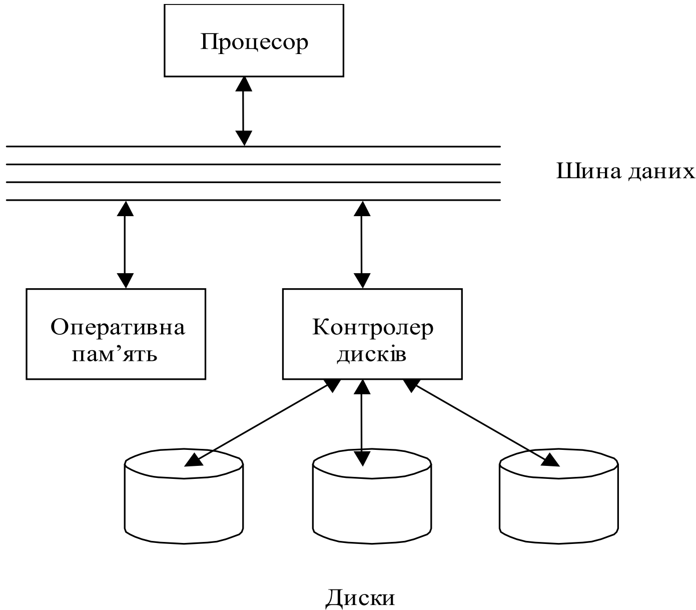


Рис. 6. Схема простої комп’ютерної системи

- *Швидкість обертання блока дисків:* 5 400 об./хв., тобто приблизно одне обертання кожні 11 мілісекунд, хоча існують диски з більшою або меншою швидкістю обертання.
- *Кількість пластин.* Звичайний дисковий пристрій має 5 пластин (10 поверхонь). Але гнучкий диск (дискета) або диск формату «Zip» складається з однієї пластини. Також випускають моделі приводів, які дають змогу використовувати до 30-ти поверхонь.
- *Кількість доріжок на поверхні.* Поверхня звичайної пластини диска здатна містити до 20 000 доріжок, хоча для дискет цей показник значно нижчий (40 доріжок).
- *Кількість байтів на доріжці.* Звичайна доріжка може зберігати близько мільйона байтів, однак доріжки дискети мають значно меншу ємність (18 750 байтів). Нагадаємо, що доріжки розбиті на сектори. На рис. 5 проілюстровано

12 секторів на доріжці, хоча насправді доріжки сучасних моделей дисків здатні містити до 500 секторів. Місткість одного сектора вимірюється декількома тисячами байтів.

Приклад 3. Розглянемо характеристики гіпотетичної моделі дискового пристрою «Soaring 747», одного з найкращих представників дисків:

- 8 пластин або 16 поверхонь;
- 2^{14} або 16 384 доріжок на поверхні;
- в середньому 2^7, або 128 секторів на доріжці;
- 2^{12} або 4 096 байтів у секторі.

Для обчислення загальної місткості дискових пластин приводу «Soaring 747» достатньо помножити між собою значення 16 (поверхонь), 16 384 (доріжок), 128 (секторів) і 4 096 (байтів). Як результат отримаємо 2^{37} байтів, або 128 Гбайт. Одна доріжка може зберігати 128*4096 байтів, або 512 Кбайт. Якщо розмір блока дорівнює 2^{14} або 16 384 байтів, то один блок може містити дані з чотирьох послідовних секторів, і тоді на одну доріжку припадатиме 128/4 = 32 блока.

Діаметр пластин дисків «Soaring 747» дорівнює 3,5 дюйми. Доріжки займають зовнішню поверхню пластини шириною 1 дюйм, а внутрішня частина шириною 0,75 дюйма залишається вільною. Питома густина запису в радіальному напрямі становить 16 384 бітів на дюйм, відповідно до кількості доріжок.

Однак густина запису в межах доріжки суттєво більша. Припустимо, що кожна доріжка містить у середньому по 128 секторів, а зазори займають 10% площі доріжки, отож згадані вище 512 Кбайт (4 Мбіт) повинні розподілитися на 90% площі доріжки. Довжина зовнішньої доріжки дорівнює 3,5 π, або близько 11 дюймів. Дані об’ємом 4 Мбіт зберігаються на ділянках доріжки загальною довжиною близько 9,9 дюйма (90% від 11 дюймів), отож густина запису на зовнішній доріжці становить приблизно 420 000 бітів на дюйм.

Внутрішня доріжка, однак, має діаметр тільки 1,5 дюйма, так що ті ж 4 Мбіт даних повинні вміститися на фрагментах доріжки довжиною 0,9*1,5*π, або 4,2 дюйма. Густина запису на внутрішній доріжці становить, відповідно, близько 1 Мбіт на дюйм.

Оскільки (у випадку однорідного розподілу секторів по доріжках) значення густини запису на зовнішній і внутрішній доріжках суттєво відрізняється, у «Soaring 747», як і в інших сучасних моделях дискових пристроїв, підтримується конструктивне рішення, згідно з яким зовнішні доріжки містять більшу кількість секторів, ніж внутрішні. Наприклад, множину доріжок можна було б розділити в радіальному напрямі на три частини і вважати, що «внутрішні» доріжки містять по 96, «середні» – по 128, а «зовнішні» – по 160 секторів. Тоді значення густини запису змінюватиметься не так широко – від 530 000 біт до 742 000 біт у напрямі від зовнішніх доріжок до внутрішніх.

Приклад 4. Якщо пристрої на магнітних дисках впорядкувати за значеннями ємності їхніх носіїв і розмістити на гіпотетичній осі, то недалеко від нуля відокремлено розміститься група приводів стандартних 3,5-дюймових дискет. Дискета містить одну пластину з двома поверхнями, кожна з яких містить по 40 доріжок. Загальна місткість дискети у форматі PC або MAC становить близько 1,5 Мбайт, або 150 000 бітів (18 750 байтів) з розрахунку на одну доріжку. Близько четвертини площі поверхні пластини відводиться під зазори та інші службові області, незалежно від способу форматування.

3.4. Параметри доступу до даних на диску

Вивчаючи інформаційні технології, необхідно розуміти не тільки те, як зберігаються дискові дані, але й зрозуміти механізми маніпуляції цими даними. Оскільки в процесах обчислень безпосередню участь бере тільки та інформація, яку розташовано в оперативній пам’яті або кеш-пам’яті, то єдиний аспект, який співвідносить ці процеси з дисками, стосується функцій передачі блоків даних між дисками та ОП. Операції зчитування і запису блоків (або послідовних секторів диска, що відповідають блокам) здійснюють за виконання таких умов:

а) головки займають позицію на циліндрі, що містить доріжку, на якій розташовані потрібні сектори;

б) сектори поверхні пластини (що обертається), які треба зчитати або записати, розташовані безпосередньо під/над головкою.

Проміжок часу, який відділяє момент активізації запиту на читання або запис дискового блока від моменту, коли вміст блока з’явиться в оперативній пам’яті або збережеться на диску, називають *часом затримки* (*latency time*) диска.

Час затримки налічує кілька періодів:

1. Час, який витрачає процесор або контролер дисків на активізацію запиту, що, зазвичай, дорівнює часткам мілісекунди (ним можна знехтувати). Проміжками часу, протягом яких контролер і/або шина зайняті опрацюванням запитів на читання чи запис дискової інформації, посланих іншими процесами можна також проігнорувати.
2. *Час пошуку* (*seek time*) – це проміжок часу, необхідний для встановлення блока головок на певний циліндр. Час може дорівнювати нулю, якщо головки, як виявилось, вже перебувають на потрібному циліндрі. Загалом блокові головок необхідні деякі незначні періоди часу для того, щоб *розпочати рух*, а потім (у момент досягнення потрібного циліндра) – *щоб зупинитись*, а також для *виконання переміщення* як такого. Періоди старту, руху від однієї доріжки до сусідньої і зупинки в сумі обчислюються декількома мілісекундами, проте для переміщення всіма доріжками потрібен час – 10-40 мілісекунд.
3. *Час обертання* – це період часу, необхідний для обертання пластини диска до моменту, доки головка не досягне першого із секторів, які належать блокові, що підлягає читанню або записуванню. Час повного обертання пластини, зазвичай, становить 10 мілісекунд. У середньому наближенні можна вважати, що для встановлення потрібного сектора навпроти головки необхідно здійснити півоборот пластини, тобто витратити 5 мілісекунд. Цей процес проілюстровано на рис. 7.
4. *Час передачі* – це проміжок часу, необхідний для обертання пластини на кут, достатній для того, щоб головка могла зчитати вміст усіх секторів, що формують блок. Якщо

доріжка містить 250 000 байт, а час повного обертання пластини оцінюється величиною 10 мілісекунд, то за 1 мілісекунду можна зчитати 25 Кбайт даних, а час передачі блока в 16 384 байт становить близько 2/3 мілісекунди.

Приклад 5. Спробуємо оцінити час, необхідний для читання блока даних розміром 16 384 байтів з диска моделі «Soaring 747» (див. приклад 3). Спочатку необхідно запам'ятати такі часові характеристики пристрою.

- Блок дисків обертається зі швидкістю 7 200 об./хв., тобто здійснює одне обертання за 8,33 мілісекунди (60 000 мс/7200).
- Блок головок на старт, переміщення на віддаль в 1 000 циліндрів і зупинку витрачає по одній мілісекунді. Отож головки переміщаються від доріжки до доріжки за 1,001 мілісекунди, а від внутрішньої до зовнішньої (усіх доріжок 16 384) – приблизно за час в 17,38 мілісекунди.

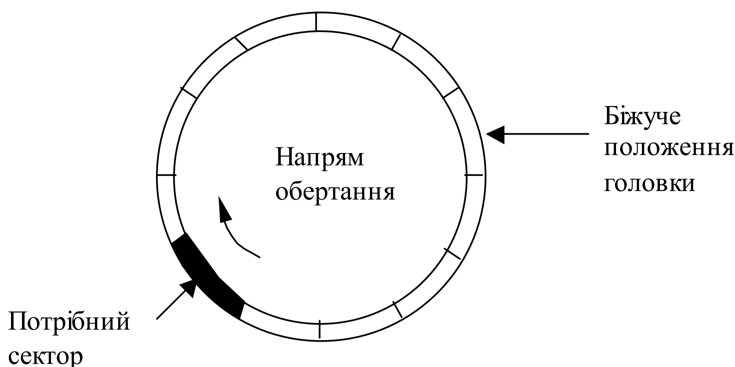


Рис. 7. Обертання для зайняття позиції сектором навпроти головки

Обчислимо максимальний, мінімальний і середній час, необхідний для читання блока розміром 16 384 байт. Мінімальний час читання блока за винятком величин можливих періодів зайнятості контролера і/або шини, якими можна знехтувати, насправді зводиться до часу передачі. У найкращому випадку

головка може бути безпосередньо навпроти потрібних доріжок і першого сектора блока, який необхідно прочитати.

Оскільки «Soaring 747» може зберігати 4 096 байтів у секторі, то блок займає чотири сектори. Отож для читання блока головка повинна «пройти» над чотирма секторами і трьома зазорами, що їх розділяють. Нагадаємо, що сумарні площі зазорів і секторів становлять, відповідно, 10% і 90% від площі доріжки. Доріжка на пластині приводу «Soaring 747» містить 128 секторів і стільки ж зазорів. Оскільки зазори всі разом покривають кут 36 градусів, а сектори – 324 градуси, то величина центрального кута дуги, яка охоплює 4 сектори і 3 зазори, дорівнює 36∗(3/128)+324∗(4/128)=10,97 градуса. Отже, час передачі становитиме (10,97/360)∗0,00833=0,000253 секунди, або близько четвертої частини мілісекунди, де 10,97/360 – частина повного кута повороту, яка забезпечує можливість читання блока, а 0,00833 секунди – час, який витрачається на повне обертання пластини.

Тепер оцінимо максимально можливу тривалість проміжку часу, необхідного для читання дискового блока. У гіршому випадку можна вважати, що головка розташована над самим внутрішнім циліндром, а читати потрібно сектори, що належать самому зовнішньому циліндрові (або навпаки). Перше, що повинен виконати контролер, – це перемістити блок головок. Як згадувалось вище, для переміщення головок приводу «Soaring 747» по всіх циліндрах необхідно затратити 17,38 мілісекунди – це найбільший час пошуку.

Найгірше, що може статися після того, як головка прибула до потрібного циліндра, – це виникнення ситуації, коли головка розташована безпосередньо після першого сектора потрібного блока. Оскільки блок зчитуватиметься з першого сектора, то нам доведеться почекати 8,33 мілісекунди, доки пластина не виконає повне обертання, щоб початок блока виявився безпосередньо біля головки. Як тільки це сталося, блок зчитається протягом часу передачі, який становитиме 0,25 мілісекунди. Отже, час затримки диска в найгіршому випадку становитиме 17,38+8,33+0,25=25,96 мс.

Зрештою, обчислимо середній час читання блока. Середнє значення часу передачі і часу обертання підрахувати досить легко:

перше, як і раніше, становить приблизно чверть мілісекунди, а друге – це часовий проміжок, необхідний для повороту пластини на 180 градусів, він дорівнює 4,17 мілісекунди. Можна вважати, що середній час пошуку – це просто період, протягом якого головка може «пробігти» у радіальному напрямі через половину доріжок. Така гіпотеза, однак, не цілком вірна, оскільки, ймовірніше, що спочатку головки є на позиції ближче до середини пластини, отож для переміщення до необхідного циліндра вони, у середньому, пройдуть коротший шлях. Точнішу оцінку середньої кількості доріжок, через які повинна переміститися головка, щоб потрапити на шукану доріжку, можна отримати наступним чином. Припустимо, що в початковий момент часу головка може перебувати з однаковою ймовірністю на будь-якому з 16 384 циліндрів. Якщо це циліндри з номерами 1 або 16 384, тоді середня величина дорівнюватиме (1+2+…+16 383)/16 384, або близько 8192. Якщо головка спочатку перебуває на середньому циліндрі з номером 8 192, тоді для переміщення до потрібного циліндра у довільному напрямі їй необхідно буде подолати четвертину (4 096) від загальної кількості циліндрів. Отже, якщо вихідна позиція головки зміниться в межах від 1 до 8 192, то середня віддаль зменшиться з 8 192 до 4 096 за квадратичним законом. Аналогічно, якщо початкове положення головки – 8 192-16 384, то середня віддаль також оцінюється квадратичною функцією, що зростає від 4 096 до 8 192.

Якщо просумувати отримані результати, то неважко зробити висновок, що середня віддаль, яку долає головка вздовж диска в радіальному напрямі, дорівнює одній третій від загальної кількості доріжок, або 5 461. Отже, середній час пошуку, який складається з суми величини періоду старту і переміщення по третій частині доріжок, дорівнює 1+5 461/1 000=6,46 мілісекунди. А середній час затримки диска зі значенням трьох компонентів – середнього часу пошуку, середнього часу обертання і середнього часу передачі – 6,46+4,17+0,25=10,88 мілісекунди.

3.5. Записування блоків

Процес запису блоків в його простій формі (без перевірки коректності збереження інформації) аналогічний до функції

зчитування блоків. Мінімальне, максимальне і середнє значення часу записування блока збігається з однойменними параметрами процесу читання блока.

3.6. Модифікація вмісту блоків

Модифікувати вміст блока безпосередньо на диску неможливо. Навіть у тому випадку, коли змінити необхідно лише декілька байтів блока (наприклад, значення компоненти одного з кортежів відношення), необхідно виконати такі дії:

1) здійснити читання блока в оперативну пам'ять;
2) внести необхідні зміни у вміст копії блока в ОП;
3) зберегти блок на диску;
4) здійснити, за необхідності, перевірку коректності записаних даних.

Отже, значення загального часу модифікації вмісту блока визначається сумою періодів читання, оновлення даних у пам'яті (можна знехтувати з огляду малості у порівнянні з тривалістю дискових операцій), записування, а також, якщо виконується перевірка коректності, часу повного обертання диска і додаткової операції читання.

Вправи для опрацювання

Вправа 1. Дисковий привід "Soaring 777" має такі параметри:

- десять поверхонь, по 10 000 доріжок на кожній;
- кожна доріжка містить 1 000 секторів місткістю 512 байтів;
- зазори займають 20% площі доріжки;
- пластини обертаються зі швидкістю 10 000 об./хв.;
- час переміщення головки через n доріжок оцінюється величиною $1+0{,}001*n$.

Визначити:

1. Яка загальна місткість дисків приводу?
2. Яке максимальне значення часу пошуку?
3. Яке максимальне значення часу обертання?
4. Якщо вважати, що блок містить 16 384 байти, тобто охоплює 32 сектори, то який час передачі такого блока?
5. Яке середнє значення часу пошуку?
6. Яке середнє значення часу обертання?

Вправа 2. Припустимо, що головка приводу "Soaring 747" (див. приклад 3) знаходиться на доріжці з номером 2 048, тобто на віддалі від краю пластини, що дорівнює 1/8 її радіуса. Припустимо також, що черговий запит передбачає необхідність читання блока з довільної доріжки. Обчисліть середній час виконання подібної операції читання.

Тема 4. Використання вторинних пристроїв зберігання

Здебільшого, під час дослідження алгоритмів обробки даних вважають, що інформація розташована в ОП і на доступ до одного елемента даних витрачається не більше і не менше часу, ніж на звертання до будь-якого іншого елемента. Подібну схему дій прийнято називати *моделлю обчислень з довільним доступом до даних*. Якщо ж йдеться про розробку технологій обробки значних обсягів інформації (наприклад, СКБД), то необхідно прийняти положення про те, що якою б ємною не виявилась ОП, загалом не вдається розмістити в ній усі необхідні дані цілковито. У цій ситуації при проектуванні ефективних алгоритмів маніпуляції даними не можна не враховувати необхідності звертання до вторинних і навіть третинних пристроїв зберігання. Найкращі алгоритми обробки інформації надто великих об'ємів дуже серйозно відрізняються від аналогів, орієнтованих на розв'язання тих же проблем у рамках моделі обчислень з довільним доступом до даних.

У цій темі ми зосередимо увагу переважно на способах ефективної взаємодії пристроїв оперативної і вторинної пам'яті. Зокрема, продемонструємо, що найбільшу перевагу має той алгоритм, який забезпечує мінімальну кількість звертань до диска, – навіть у тому випадку, коли з погляду моделі обчислень в ОП його вважають не найефективнішим.

4.1. Модель обчислень з функціями введення/виведення

Розглянемо простий комп'ютер з одним процесором, контролером і диском, на якому працює СКБД, що обслуговує значну кількість користувачів, які звертаються до неї як із запитами, так і з командами модифікації даних. Припустимо, що база даних настільки велика, що не може бути завантажена в ОП

цілковито. У випадку наявності значної кількості одночасно працюючих користувачів, кожен з яких надсилає запити на виконання операцій дискового введення/виведення, контролер дисків змушений організовувати чергу запитів. Далі ми розглянемо різні способи підвищення продуктивності такої системи, дотримуючись такого правила, яке визначає суть *моделі обчислень з функціями введення/виведення* ((*I/O computation model*).

- ***Презумпція пониження затрат на операції введення/виведення.*** Якщо доведеться опрацювати блок дискових даних, то час, необхідний для виконання операцій введення/виведення, суттєво перевищить час, який витрачається на маніпуляцію даними в ОП. Отож продуктивність алгоритмів значною мірою залежить від кількості звертань до диска для читання і записування. Останнє необхідно мінімізувати.

У подальших прикладах вважатимемо, що розмір блока даних дорівнює 16 Кбайт і використовуватимемо моделі дискового пристрою «Soaring 747», часові характеристики якого (зокрема, середній час читання і записування блока, який дорівнює 11 мілісекунд) визначено у прикладі 5.

Приклад 6. Нехай у базі даних є відношення R і запит полягає у відшуканні в R кортежу, який містить ключове значення k. Важливо (згодом ми розтлумачимо це детальніше) створити індекс для R, який даватиме змогу ідентифікувати дисковий блок, що містить кортеж із заданим значенням k. Проте загалом не зовсім важливо, чи може індекс «підказати», в якому місці блока розташовано шуканий кортеж. Адже на операцію читання дискового блока розміром 16 Кбайт потрібно витратити 11 мілісекунд, а за такий час сучасний мікропроцесор може виконати мільйони інструкцій. Водночас на пошук значення k у блоці, який вже є в ОП, доведеться витратити всього декілька тисяч елементарних інструкцій (навіть у тому випадку, коли використовується «найдурніша» стратегія лінійного пошуку). Отже, проміжок часу, необхідний для пошуку значення в ОП, дуже малий (ним можна знехтувати) порівняно з часом зчитування дискового блока.

4.2. Сортування даних у вторинних сховищах

Як приклад, що ілюструє необхідність модифікації алгоритмів опрацювання даних при переході до моделі обчислень з функціями введення/виведення, розглянемо проблему сортування даних, загальний об'єм яких значно перевищує місткість ОП. Спочатку конкретизуємо задачу і подамо опис характеристик комп'ютера, на якому її необхідно виконувати.

Приклад 7. Припустимо, що ми маємо справу з відношенням R крупного розміру, яке налічує 10 000 000 кортежів. Кожен кортеж представлений записом з декількома полями, одне з яких є полем *ключа сортування* (*sort key*), або просто *ключовим* полем. Мета, поставлена перед алгоритмом сортування, полягає у впорядкуванні записів за зростанням значень ключа сортування.

Ключ сортування може бути або може не бути «ключем» у тому розумінні, яке надається в *SQL* поняттю *первинного ключа* (*primary key*). Як відомо, первинний ключ гарантує, що значення його компонентів у межах всіх кортежів відношення є унікальними. Якщо ж можливість дублювання значень ключа сортування передбачено, то вважають можливим довільний порядок розташування записів з однаковим вмістом ключа. Вважатимемо, що значення ключа сортування унікальні.

Множина записів (кортежів) відношення R розміщується в дискових блоках розміром 16 384 байтів. Припустимо, що один блок налічує 100 записів (кожен запис займає 160 байтів). Якщо врахувати, що разом з записами у блоці міститься додаткова службова інформація (розглянемо нижче), то для 100 записів якраз підійде блок розміром 16 384 байтів. Отже, відношення R займатиме 100 000 блоків загальною місткістю близько 1,64 мільярдів байтів ≈1,6 Гбайт.

Комп'ютер, на якому виконується сортування, оснащений одним дисковим приводом «Soaring 747» (див. приклад 3) і ОП, 100 Мбайт якої доступні для буферизації блоків з кортежами відношення R. Фактичний об'єм ОП більший, однак частину пам'яті використовує система. Кількість блоків, які можна розмістити у 100 Мбайтах пам'яті (це $100 \cdot 2^{20}$ байт), становить $100 * 2^{20}/2^{14}$, або 6 400.

Якщо дані, які необхідно посортувати, поміщаються в ОП цілковито, то їх можна впорядкувати за допомогою різноманітних алгоритмів [4, 5]. Найефективнішим є алгоритм швидкого сортування. Однією з кращих вважають версію, яка передбачає впорядкування тільки ключових полів, зберігаючи разом з ними покажчики на повні записи.

Зауважимо, що ці алгоритми втрачають свої переваги, коли дані розташовані на вторинних носіях. Якщо інформація, яку необхідно посортувати, розташована, здебільшого, поза ОП, то доцільно використати такий підхід, який передбачає виконання мінімальної кількості переміщень блоків даних із вторинної пам’яті в оперативну. Часто подібні алгоритми передбачають здійснення декількох проходів, і під час одного проходу кожен блок один раз зчитується з диска в ОП і зберігається на диску. У пункті 4.4 ми розглянемо один із таких алгоритмів детальніше.

4.3. Сортування злиттям

Відомо, що алгоритм *сортування злиттям* (*merge sort*) дає змогу об’єднувати посортовані списки в крупніші посортовані списки. Цей алгоритм передбачає циклічне зіставлення найменших ключів, по одному з кожного списку, переміщення запису, який відповідає знайденому найменшому ключу, в підсумковий список, і повторення операцій доти, доки один зі списків не буде вичерпано. Потім до результуючого списку долучають залишок одного зі списків. Таким шляхом отримують вичерпний список записів з двох вихідних списків, посортований за зростанням значень ключа.

Приклад 8. Нехай маємо два посортовані списки з чотирма записами у кожному. Вважатимемо, що структура запису містить тільки ключовий елемент цілочислового типу. Списки такі: (1, 3, 4, 9) і (2, 5, 7, 8). Ітерації процесу злиття проілюстровано на рис. 8.

На першій ітерації порівнюють найменші елементи, 1 і 2, з обох списків. Оскільки 1<2, то елемент 1, відібраний з першого списку, стає початковим елементом підсумкового списку. На другій ітерації порівнюють елементи 3 і 2; елемент 2 «перемагає» і переноситься в підсумковий список. Процес продовжується доти, доки на ітерації 7 другий список не буде вичерпано, після чого

частина першого списку, що залишилася, переноситься в кінець підсумкового списку, і злиття закінчується. Зверніть увагу, що елементи об'єднаного списку, як і вимагалося, розташовані у порядку зростання, оскільки на кожній ітерації обирали найменший із елементів, що залишилися.

Ітерація	Список 1	Список 2	Підсумковий список
Початок	1, 3, 4, 9	2, 5, 7, 8	Порожній
1)	3, 4, 9	2, 5, 7, 8	1
2)	3, 4, 9	5, 7, 8	1, 2
3)	4, 9	5, 7, 8	1, 2, 3
4)	9	5, 7, 8	1, 2, 3, 4
5)	9	7, 8	1, 2, 3, 4, 5
6)	9	8	1, 2, 3, 4, 5, 7
7)	9	Порожній	1, 2, 3, 4, 5, 7, 8
8)	Порожній	Порожній	1, 2, 3, 4, 5, 7, 8, 9

Рис. 8. Злиття двох посортованих списків в один

Час злиття списків в оперативній пам'яті лінійно залежить від суми їхніх довжин: оскільки вихідні списки раніше посортовані, то претендентами на долучення до результуючого списку є тільки елементи на першій позиції списків, і кожна операція їх порівняння виконується протягом постійного проміжку часу. Класичний варіант алгоритму сортування злиттям, який виконується рекурсивно за $log_2 n$ ітерацій (якщо сортується n елементів), можна описати так.

Основа. Якщо список складається з одного елемента, то нічого не потрібно робити, оскільки список посортований.

Індукція. Якщо список налічує понад один елемент, то ділимо його довільно на два списки однакової довжини або один на одиницю довший у випадку непарної кількості елементів у вихідному списку. Рекурсивно посортувати два підсписки, а потім їх злити в один посортований список.

Аналітичні оцінки якості цього алгоритму добре відомі. Час *T(n)*, необхідний для сортування *n* елементів, можна оцінити як $T(n)=O(n\log_2 n)$, тобто величина пропорційна $n\log_2 n$.

4.4. Сортування двофазним багатокомпонентним злиттям

Розглянемо, як можна використати алгоритм *сортування двофазним багатокомпонентним злиттям* (*two –phase, multiway merge sort – TPMMS*) з метою впорядкування відношення *R* з прикладу 7, на комп'ютері з параметрами, зазначеними в тому ж прикладі. TPMMS – один з найпопулярніших алгоритмів, який використовують в багатьох застосуваннях баз даних. Він полягає в наступному:

- *Фаза 1.* Виконати сортування довільної кількості списків даних, максимальний об'єм яких дорівнює доступній місткості ОП.
- *Фаза 2.* Здійснити злиття посортованих підсписків, отриманих на попередній фазі, в один посортований список.

Якщо ж дані значного об'єму розташовані на вторинних носіях, то розпочинати сортування, починаючи з базису рекурсії, згідно з яким необхідно розглядати тільки один або декілька записів, не має сенсу. Отож доцільно здійснювати рекурсію, заповнюючи всю доступну область ОП цілковито і використовуючи швидкі алгоритми сортування. Для виконання першої фази повторюють наступні ітерації необхідну кількість разів:

1) заповнити вільну область ОП блоками, які містять відношення, що треба посортувати;
2) посортувати записи в ОП, використовуючи швидкодіючі алгоритми;
3) зберегти посортований підсписок у новій ділянці вторинної пам'яті.

Після завершення першої фази всі записи вихідного відношення будуть один раз зчитані в ОП і долучені до посортованих підсписків, збережених на диску.

Приклад 9. Розглянемо відношення R з прикладу 7. Як було встановлено, ОП комп'ютера з параметрами, заданими у тому ж прикладі, може вмістити 6 400 з загальної кількості блоків, яка становить 100 000. Отож для виконання першої фази необхідно 16 разів заповнювати ОП зчитаними блоками, сортувати записи, а потім зберігати посортовані підсписки на диску. Останній підсписок має 4 000 блоків, а не 6 400, як кожен із 15-ти попередніх підсписків.

То ж скільки часу знадобиться на виконання першої фази алгоритму? Нам доведеться один раз зчитати і заново записати 100 000 блоків, тобто виконати 200 000 операцій дискового введення/виведення. Для прикладу можна вважати, що блоки розташовані на диску в довільному порядку. У цьому випадку середній час кожної операції зчитування і записування дорівнює 11 мілісекунд. Отож на операції введення/виведення, які виконуються під час першої фази, необхідно витратити 2 200 секунд, або майже 37 хв. (понад 2 хв. на введення/виведення кожного підсписку). Оскільки за 1 с процесор може виконати сотні мільйонів інструкцій, то часом сортування в ОП можна знехтувати. Отож вважатимемо, що тривалість першої фази процесу сортування становить близько 37 хвилин.

Тепер розглянемо процедуру злиття посортованих підсписків, отриманих на першій фазі. Використовуючи класичну стратегію сортування злиттям, можна здійснити послідовне об'єднання кожної пари підсписків. Однак у цьому випадку доведеться зчитувати і записувати кожен з блоків даних $2log_2n$ раз, де n – кількість посортованих підсписків, отриманих після першої фази. У нашому прикладі для злиття 16-ти посортованих підсписків необхідно зчитати і зберегти блоки кожної пари підсписків з метою формування 8-ми об'єднаних підсписків, потім послідовно повторити те саме для отримання 4-х і 2-х крупніших підсписків і, зрештою, виконати злиття двох підсписків у підсумковий посортований список. Отже, на кожен блок припадає 8 операцій дискового введення/виведення.

Доречніше зчитувати перші блоки кожного з посортованих підсписків у вхідні буфери ОП. Якщо вихідне відношення настільки об'ємне, що після виконання першої фази процесу

кількість підсписків дуже велика, то ймовірно, що неможливо зчитати по одному блоку в ОП (про це у пункті 4.5). Шістнадцяти підспискам із прикладу 7 це не загрожує.

Ще один (вихідний) буфер ОП використовують для зберігання блока, який містить біжучу порцію даних посортованого списку, і об'єм буфера повинен бути настільки великий, наскільки це можливо. Спочатку вихідний буфер порожній. Схему організації процесу обчислень з участю вхідних і вихідного буферів наведено на рис. 9.

Щоб здійснити злиття посортованих підсписків, необхідно виконати такі дії.

1. Серед ключів перших елементів усіх підсписків знайти ключ з мінімальним значенням. Оскільки операції виконуються в ОП, то цілком достатньо використати алгоритм лінійного пошуку (або інший [1]).
2. Перемістити знайдений мінімальний елемент на першу доступну позицію списку у вихідному буфері.
3. Якщо вихідний буфер заповнено, то зберегти його вміст на диску і знову ініціювати цей буфер для зберігання наступної порції даних.
4. Якщо блок, з якого взято мінімальний елемент, вичерпаний, то здійснити читання наступного блока цього ж посортованого підсписку в цей же вхідний буфер. Якщо вичерпано і підсписок, то залишити буфер порожнім і більше до нього не звертатися.

На відміну від першої фази, при виконанні другої фази блоки зчитуються в довільному порядку, оскільки наперед невідомо, який з блоків вичерпається першим. Але кожен блок читається з диска тільки один раз. Отож загальна кількість операцій читання блоків дорівнює 100 000. Аналогічно, кожен запис надходить у вихідних буфер, вміст якого потім зберігається на диску. Відповідно, записувань буде 100 000. Оскільки і в цьому випадку часом обчислень в ОП можна знехтувати, то час виконання другої фази також становитиме 37 хв. Загальний час сортування – 74 хв.

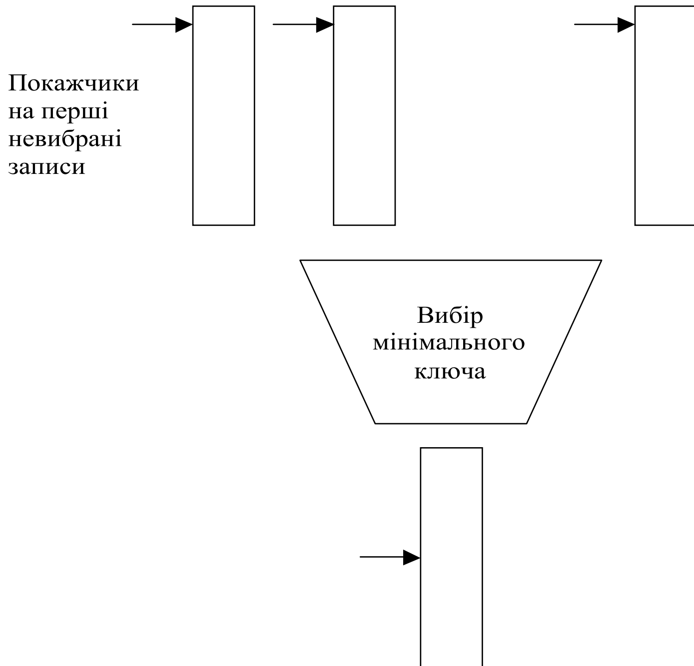


Рис. 9. Схема обчислень при виконанні другої фази сортування багатокомпонентним злиттям

✓ ***Корисно знати.*** *Наскільки великими повинні бути блоки?* Вивчаючи характеристики дискового приводу «Soaring 747», ми вважаємо, що розмір блока становить 16 Кбайт. Однак можна навести аргументи на користь вибору блока більшого об’єму. У прикладі 5, нагадаємо, ми встановили, що час передачі блока довжиною 16 Кбайт становить 1/4 мілісекунди, проте сума середніх значень часу пошуку і часу обертання – 10,63 мілісекунди. Якщо розмір блока подвоїти,

то нам вдасться удвічі знизити кількість операцій дискового введення/виведення, необхідних для виконання алгоритмів обробки даних, подібних до TPMMS. Щодо цього час передачі блока зростає всього до 0,50 мілісекунди. Отже, загальний час сортування зменшиться майже вдвічі. Якщо збільшити розмір блока до 512 Кбайт (тобто до величини об'єму цілої доріжки диска «Soaring 747»), то час передачі зросте до 8 мілісекунд, а загальний час читання блока – до 20 мілісекунд, але для здійснення двох фаз сортування того ж масиву даних, який розглянуто в прикладі 9, нам необхідно виконати загалом 12 500 операцій з блоками, що даватиме змогу зменшити загальний час сортування у понад 17 разів.

Існують аргументи і проти збільшення розміру блока. По-перше, якщо блок охоплює декілька доріжок, то запропонувати ефективні способи використання подібних блоків проблематично. По-друге, відношення невеликого розміру можуть займати тільки частину блока, що спричинить до зайвих витрат дискової пам'яті. Окрім того, для зберігання інформації на носіях вторинних пристроїв використовують спеціальні структури даних, характеристики яких відзначаються підвищеною ефективністю, якщо дані розподілено між значною кількістю блоків, тобто розміри блоків не надто великі. Як буде проілюстровано у пункті 4.5, чим крупніші блоки, тим меншу кількість записів можна посортувати за допомогою алгоритму TPMMS. Однак зі зростанням продуктивності процесорів і ємності дискових пристроїв існує тенденція і до збільшення розмірів блоків.

4.5. Багатокомпонентне злиття і надзвичайно великі відношення

Розглянутий вище алгоритм сортування двофазним багатокомпонентним злиттям дає змогу опрацьовувати і надзвичайно

крупні множини записів. Щоб отримати числові оцінки допустимих об'ємів даних, введемо такі позначення (у байтах):

1) B – розмір блока;
2) M – об'єм ОП, доступний для буферизації блоків;
3) R – розмір запису.

Кількість буферів, які вдасться розмістити в ОП, становить M/B. Максимальна кількість посортованих підсписків оцінюється величиною $(M/B)-1$ (один буфер – вихідний). Цим числом визначається і кількість стадій заповнення оперативної пам'яті записами, які необхідно посортувати. На кожній стадії сортуватиметься максимум M/R записів. Отож верхня межа кількості записів, які можна посортувати, становитиме $(M/R)\ ((M/B)-1)$, або наближено $M^2/(RB)$.

Приклад 10. Якщо брати до уваги значення параметрів з прикладу 7, то отримаємо M = 104 857 600, B = 16 384, R = 160. Верхня межа кількості записів, які можна посортувати, у цьому випадку становитиме M^2/RB = 4,2 мільярда. Для їхнього зберігання необхідно 0,67 Тбайт. Відношення такого об'єму не можна розмістити на носіях приводу «Soaring 747».

Якщо необхідно посортувати більшу кількість записів, то можна доповнити процес третьою фазою: спочатку, використовуючи алгоритм TPMMS, створити на основі груп з M^2/RB записів посортовані підсписки, а згодом здійснити злиття $(M/B)-1$ таких підсписків у підсумковий посортований список.

Третя фаза дає змогу сортувати майже M^3/RB^2 записів, які займають M^3/B^3 блоків. З урахуванням даних з прикладу 7 ці параметри матимуть значення: 27 трильйонів записів і 4,3 Пбайт об'єму. Сьогодні це нечувані дані. Оскільки місткість сучасних вторинних пристроїв далека навіть від 0,67 Тбайт (припустима для алгоритму TPMMS), то двофазної версії алгоритму цілком вистачає для вирішення будь-яких реальних задач.

Вправи для опрацювання

Вправа 3. Розглянемо відношення R, яке налічує 10 000 000 кортежів. Записи розміщуються в дискових блоках розміром 16 384 байтів. Припустимо, що в один блок вміщається 100 записів. Отож відношення R займатиме 100 000 блоків. Комп'ютер, на

якому виконується сортування, оснащений одним дисковим приводом "Soaring 777", характеристики якого задано у вправі 1, і має оперативну пам'ять, 100 Мбайтів якої доступні для буферизації блоків. Яким буде час сортування двофазним багатокомпонентним злиттям відношення R?

Вправа 4. Скільки операцій введення/виведення необхідно виконати для сортування за допомогою алгоритму двофазним багатокомпонентним злиттям відношення R з прикладу 7, якщо значення окремих параметрів відношення і конфігурації комп'ютера змінено так:

1) кількість кортежів відношення подвоєно і вона становить 20 000 000;
2) довжину кортежу відношення збільшено вдвічі, вона становить 320 байтів.
3) розмір блока подвоєно до 32 768 байтів.
4) об'єм доступної ОП збільшено до 200 Мбайтів.

Вправа 5. Нехай задано відношення, яке містить n кортежів, довжина кожного з яких дорівнює R байтів, і параметри комп'ютера – об'єм M оперативної пам'яті і розмір B дискового блока. Це дає змогу посортувати кортежі відношення за допомогою алгоритму TPMMS. Як змінити n, залишаючи усі інші умови без змін, якщо подвоїти значення: а) B; б) R; в) M?

Вправа 6. Виконайте вправу 5 за умови, що для впорядкування кортежів застосовують алгоритм сортування трифазним багатокомпонентним злиттям.

Вправа 7. Як представити кількість записів, які можна впорядкувати за допомогою алгоритму сортування k-фазним багатокомпонентним злиттям як функцію від параметрів R, M і B (див. вправу 5) і цілочислового значення k?

Тема 5. Підвищення ефективності дискових операцій

Здійснюючи аналіз продуктивності алгоритмів сортування дискових даних (у пункті 4.4), ми вважали, що інформація

зберігається на єдиному диску і блоки зчитуються з довільних позицій. Подібна гіпотеза має сенс у тому випадку, коли йдеться про систему, що виконує водночас безліч запитів, які стосуються невеликих фрагментів даних. Однак, якщо у будь-який період задача системи зводиться до сортування крупного відношення, то можна суттєво зменшити витрати часу на виконання дискових операцій, якщо потурбуватися про строгий порядок розміщення блоків даних, які сортуються, на носії.

Наступні дії дають змогу пришвидшити обробку запитів і/або за одиницю часу виконувати їхню більшу кількість.

- Розмістити блоки, які необхідно опрацьовувати разом, на одному циліндрі, що даватиме змогу зменшити до нуля значення часу пошуку і, можливо, часу обертання.
- Розмістити інформацію на декількох дисках меншого об’єму замість того, щоб використовувати один диск того ж об’єму. За наявності блоків головок, які рухаються незалежно, об’єм даних, які можуть бути зчитані/записані за одиницю часу, зростає пропорційно кількості дискових пристроїв.
- Створити «дзеркальні» копії одних і тих самих даних на декількох дисках. Це даватиме змогу не тільки зберегти дані у випадку виходу з ладу одного з дискових пристроїв, але і збільшити пропускну здатність системи.
- Використовувати алгоритми рівня ОС, СКБД або контролера дисків, які слугують для впорядкування в часі запитів до блоку диска.
- Застосовувати стратегії попереднього зчитування певних порцій дискових даних в ОП, які передують використанню цих даних.

5.1. Групування даних по циліндрах диска

Оскільки час пошуку забирає майже не половину середнього часу зчитування/запису блока, то доцільно передбачити можливість розміщення даних, які, ймовірно, опрацьовуються спільно, у межах

одного циліндра. Якщо одного циліндра не достатньо, то використовуються декілька суміжних.

Якщо всі блоки однієї доріжки чи одного циліндра зчитуються послідовно, ми можемо знехтувати всіма складовими часу читання, окрім першого періоду часу пошуку, необхідного для переміщення головок до потрібного циліндра, першого проміжку часу обертання для досягнення початкового сектора послідовності блоків і, очевидно, часу передачі. У такому випадку нам вдасться наблизитися до теоретичної межі швидкості переміщення даних між диском і ОП.

Приклад 11. Знову проаналізуємо продуктивність алгоритму сортування двофазним багатокомпонентним злиттям. У прикладі 5, нагадаємо, ми визначили, що середнє значення часу передачі, пошуку та обертання дискового приводу «Soaring 747» становить 0,25, 6,46 і 4,17 мілісекунди, відповідно. Встановлено також, що процедура сортування відношення, яке складається з 10 000 000 записів і займає близько 1 Гбайта дискового простору, виконується за 74 хв. Цей час ділиться на 4 рівні проміжки, необхідні для здійснення функцій запису і читання блоків для двох фаз алгоритму.

Спробуємо відповісти на питання, чи допоможе впорядкування даних по циліндрах зменшити час сортування. Перша стадія процесу стосується читання вихідних блоків. У прикладі 9 ми встановили, що ОП заповнюється даними об’ємом 6 400 блоків 16 разів.

Цілком можливо передбачити зберігання 100 000 блоків даних на послідовних циліндрах. Кожен із 16 384 циліндрів приводу «Soaring 747» може зберігати близько 8 Мбайт у 512-ти блоках. Отож для розміщення вхідних даних необхідно 196 циліндрів, а для одноразового заповнення ОП необхідно звертатися до 13-ти циліндрів.

На читання даних з одного циліндра необхідно один період часу пошуку. Головки потрібно переміщати від одного циліндра до сусіднього 12 разів, на що витрачається по 1 мілісекунді. Отже, час, який необхідний для одноразового читання даних в ОП, визначається сумою таких компонентів:

1) 6,46 мілісекунди (середній час однієї операції пошуку);
2) 12 мілісекунд (час пошуку кожного із 12-ти сусідніх циліндрів);
3) 1,6 секунди (час передачі 6 400 блоків).

Двома першими компонентами можна без будь-якої шкоди знехтувати. Оскільки заповнювати ОП даними доходиться 16 разів, то загальний час зчитування даних на першій фазі процесу становить близько 26 с. Нагадаємо, що за «випадкового» розміщення блоків на цю ж операцію необхідно було б витратити майже 18 хв. Для збереження даних 16-ти посортованих підсписків можуть використовуватися інші 196 сусідніх циліндрів і часові параметри процесу записування будуть такими ж: 26 с. Час виконання першої фази загалом – 52 с (порівняйте: 37 хв. необхідно у випадку «випадкового» розміщення блоків).

З іншого боку, прийоми організації збереження даних на послідовних циліндрах не здатні зменшити час виконання другої фази сортування. Нагадаємо, що під час виконання другої фази блоки зчитуються з перших позицій 16-ти посортованих підсписків у порядку, обумовленому властивостями даних як таких. Вихідні блоки результуючого посортованого списку зберігаються по одному і чергуються з операціями читання. Отож для виконання другої фази алгоритму необхідно, як і раніше, 37 хв. Отже, впорядкування даних по циліндрах здатне зменшити час сортування вдвічі, і кращого результату без використання додаткових альтернативних підходів досягнути не вдається.

5.2. Використання декількох дискових пристроїв

Пропускну здатність системи вдається суттєво підвищити, якщо замінити один дисковий привід декількома. Подібну архітектуру схематично проілюстровано на рис. 6, коли один контролер дисків керує водночас трьома дисками. Оскільки продуктивність контролера, шини даних і ОП значно вища, то загальний час виконання дискових операцій за подібного підходу зменшується пропорційно до кількості дискових пристроїв. Проілюструємо сказане прикладом.

Приклад 12. Припустимо, що дисковий пристрій «Soaring 737» має ті ж характеристики, що й модель «Soaring 747», однак містить тільки дві пластини (4 поверхні) та дає змогу зберегти майже 32 Гбайт даних. Припустимо також, що конфігурацію комп'ютерної системи змінено так, що один привід «Soaring 747» замінено чотирма накопичувачами «Soaring 737». Проаналізуємо, як поводитиме себе в таких умовах алгоритм сортування двофазним багатокомпонентним злиттям для сортування того ж відношення R, що містить 100 000 блоків по 16 384 байтів у кожному.

Передусім, множину записів відношення R доцільно розділити між чотирма дисками так, щоб на кожному диску дані зберігалися на 49-ти сусідніх циліндрах. У цьому випадку ми зможемо скористатися перевагами групування даних по циліндрах, розглянутими у прикладі 11. При заповненні ОП під час виконання першої фази під дані, які зчитуються з кожного з дисків, необхідно виокремити 1/4 частини пам'яті. На операцію читання 1 600 блоків з кожного диска необхідно буде лише 400 мілісекунд замість 1,6 с при використанні одного диска.

Аналогічно, виконуючи під час першої фази записування даних, ми можемо розподілити вміст будь-якого з посортованих підсписків у 4-х дисках і 13-ти сусідніх циліндрах кожного диска, зберігаючи це й же 4-кратний виграш у швидкості. Отже, на виконання першої фази загалом необхідно 13 с замість 52-х с, які необхідні для випадку застосування лише однієї стратегії групування даних по циліндрах, і 37 хв., необхідних при використанні прямолінійного підходу, що передбачає «випадкове» розміщення блоків.

Тепер дослідимо параметри другої фази алгоритму. Нам, як і раніше, необхідно зчитувати блоки з перших позицій посортованих підсписків у деякому порядку, близькому до «випадкового», який продиктовано характеристиками даних як таких. Отож наявність 4-х дисків замість одного переваг не забезпечує. Частина другої фази, яка стосується запису даних, піддається оптимізації значно легше. Для кожного з чотирьох дисків можна створити власний вихідний буфер. Однак не можна гарантувати, що чотири диски функціонуватимуть паралельно протягом всього часу сортування.

Отож приріст продуктивності оцінюється приблизним коефіцієнтом 2–3. Проте і це непогано: якщо взяти 2, то економія становитиме 9 хв. Отже, заміна одного диска на чотири і використання стратегії групування даних по циліндрах загалом даватимуть змогу знизити загальний час сортування з 74-х хвилин на 27 хв 13 с.

5.3. Створення дзеркальних копій дисків

Виникають ситуації, коли доцільно зберігати на двох або декількох дисках ідентичні копії одних і тих самих даних. Такі диски називають *дзеркальними* (*mirror*). Один із аргументів на користь дзеркальних дисків полягає в тому, що у випадку виходу з ладу одного диска дані вдається зчитати з диска-копії. Окрім головного призначення, яке полягає у підвищенні рівня надійності зберігання інформації, дзеркальні диски часто виконують функції збільшення пропускної здатності системи.

Якщо придбати 4 дискових пристрої «Soaring 747» замість одного, то система водночас виконуватиме чотири операції читання блоків. Іншими словами, незалежно від того, які саме чотири блоки потрібні в той чи інший момент, нам вдається зчитати кожен необхідний блок з будь-якого із чотирьох дисків.

Використання дзеркальних дисків не дає змоги пришвидшити процеси запису, хоча сповільнення роботи також не спостерігається. Блок записується зразу ж на всі диски, проте (оскільки операції виконуються паралельно) вони вимагають приблизно того ж часу, що і в системі з єдиним диском. Звичайно, абсолютно точної синхронності операцій запису на дзеркальні диски досягнути навряд чи вдасться, оскільки головки одного диска, наприклад, можуть бути розміщені «невигідно», тобто безпосередньо після сектора, з якого необхідно розпочинати записувати блок.

5.4. Впорядкування дискових операцій і алгоритм ліфта

Ще один спосіб підвищення пропускної здатності системи полягає у застосуванні *алгоритмів* (ми говоритимемо про алгоритми рівня контролера дисків, хоча цей аспект не має принципового значення), які дають змогу *впорядковувати* (*schedule*) запити на виконання дискових операцій у часі та обирати, які з них

мають виконуватися першими. У випадку сортування даних відношення, коли система зчитує і записує дискові блоки у певній послідовності, ці алгоритми не є надто корисними. Проте в ситуаціях, коли система виконує одночасно чимало запитів, які зачіпають незначні порції даних, засоби побудови динамічних розкладів обслуговування запитів можуть бути досить корисними.

Один із простих та ефективних способів впорядкування запитів до диска в часі називають *алгоритмом ліфта* (*elevator algorithm*). Цілком правомірно вважати, що блок головок приводу здійснює переміщення від внутрішнього до зовнішнього циліндра (і навпаки) так само, як ліфт високого будинку, рухаючись догори і донизу.

Приклад 13. Нехай за допомогою алгоритму ліфта необхідно впорядкувати звернення до диска «Soaring 747», середні часові параметри якого, нагадаємо (див. приклад 5) є такими: час пошуку – 6,46, час обертання – 4,17 і час передачі – 0,25 (усі значення в мілісекундах). Припустимо, що в деякий (нульовий) момент часу надійшли 3 запити, які передбачають звернення до циліндрів з номерами 2 000, 6 000 і 14 000, а блок головок розташований навпроти циліндра 2 000. Окрім того, припустимо, що через короткий час надійде ще три запити. Параметри всіх шести запитів наведено на рис. 10 (наприклад, запит, який передбачає необхідність звертання до циліндра з номером 4 000, надходить протягом 10-ти мілісекунд).

Для виконання однієї операції з блоком необхідно витратити такий час: 0,25 мілісекунди на надсилання, 4,17 мілісекунди – на обертання. Загалом 4,42 мілісекунди, плюс відповідний час пошуку конкретного циліндра. Час пошуку легко обчислити, користуючись правилом, викладеним у прикладі 5: 1 + кількість доріжок, поділена на 1 000. Розглянемо дію алгоритму ліфта. Для обробки запиту щодо 2 000-го циліндра пошуку робити не потрібно, оскільки блок головок на місці. Отож у час 4,42 цей запит буде цілковито виконано. Запит до циліндра 4 000 ще не надійшов, отож головки переміщаються до циліндра 6 000. Час переходу з 2 000-го до 6 000-го циліндра становить 5 мілісекунд, отож опрацювання другого запиту розпочнеться в 9,42 і завершиться за 4,42 мс, тобто в 13,84. На цей час вже активізований запит до 4 000 циліндра,

проте головки вже його минули, і не повернуться до нього, доки не опрацюють всі запити до «дальніх» циліндрів.

Номер циліндра	Момент надходження (мс)
2 000	0
6 000	0
14 000	0
4 000	10
16 000	20
10 000	30

Рис. 10. Дані, які ілюструють застосування алгоритму ліфта

Тепер надійшла черга запиту до 14 000 циліндра, і на переміщення до нього буде витрачено 9 мс, а на зчитування блока – 4,42 мс. Отож опрацьовування третього запиту завершиться за 27,26 мс. На цей момент вже надійшов запит до 16 000 циліндра, отож рух головки в обраному напрямі продовжується. Час пошуку становить 3 мс, отож операція завершиться за 27,26 + 3 + 4,42 = = 34,68 мс.

На цей час вже є запит до 10 000 циліндра, отож залишаються неопрацьованими загалом два запити, адресовані до 10 000 і 4 000 циліндрів, а запитів до циліндрів з більшими номерами немає. Отже, напрям переміщення головок зміниться. Послідовно опрацьовуватимуться два запити, які залишилися. На рис. 11 подано моменти завершення опрацювання всіх запитів.

Номер циліндра	Момент завершення (мс)
2 000	4,42
6 000	13,84
14 000	27,26
16 000	34,68
10 000	46,10
4 000	57,52

Рис. 11. Розклад обслуговування запитів за алгоритмом ліфта

Спробуємо порівняти продуктивність системи з тією, яка використовує прямолінійну стратегію «*першим прийшов – першим обслуговується*» (FIFO). Три перші запити опрацьовуються так само, якщо врахувати, що їхній порядок такий: 2 000, 6 000 і 14 000.

Однак після обробки третього запиту ми переходимо до запиту, який передбачає звернення до циліндра з номером 4 000, оскільки цей запит надійшов четвертим. Час пошуку циліндра (перехід від циліндра 14 000) становить 11 мс. Для виконання п’ятого запиту, адресованого до циліндра 16 000, час пошуку становитиме 13 мс, а на повернення до циліндра з номером 10 000 з метою виконання останнього запиту витратиться 7 мс. На рис. 12 наведено результати виконання запитів згідно з алгоритмом «*першим прийшов – першим обслуговується*».

Номер циліндра	Момент завершення (мс)
2 000	4,42
6 000	13,84
14 000	27,26
4000	42,68
16 000	60,10
10 000	71,52

Рис. 12. Розклад опрацювання запитів за принципом «*першим прийшов – першим обслуговується*»

Різниця між отриманими результатами на перший погляд не дуже значна – всього 14 мс, однак зазначимо, що кількість запитів у нашому прикладі надто мала (6) і, окрім того, три з них у двох випадках опрацьовувались в одному і тому ж порядку.

Якщо середня кількість запитів, які очікують опрацювання, зросте, то, відповідно, збільшиться і відносна ефективність алгоритму ліфта. Якщо ж кількість запитів стає надзвичайно великою, то час, який витрачається на очікування обслуговування кожного запиту, в середньому, також зросте.

5.5. Попереднє зчитування і крупномасштабна буферизація даних

Розглянемо останню методику підвищення ефективності використання вторинних пристроїв зберігання. Пов’язана вона з *попереднім зчитуванням або крупномасштабною буферизацією* даних. У деяких застосуваннях можна наперед передбачити порядок, в якому блоки повинні зчитуватися з диска. Якщо справа є саме такою, то можна завантажити дискові дані в ОП ще до того моменту, коли вони насправді будуть потрібні. Одна з переваг у цьому випадку зумовлена тим, що якість розкладу обслуговування звернень до диска, згідно з тим же алгоритмом ліфта, може бути підвищена за рахунок зменшення середнього часу обробки блока.

Приклад 15. Продемонструємо вигоду, отриману під час використання попереднього зчитування даних. Знову розглянемо другу фазу алгоритму сортування двофазним багатокомпонентним злиттям. Нагадаємо (див. приклад 9), що в процесі злиття в ОП зчитується по одному блокові з кожного посортованого підсписку, загальна кількість яких є 16. Якщо припустити, що підсписків так багато, що множина наступних блоків, які зчитуються, така велика, що цілковито заповнює ОП, то нічого кращого придумати, очевидно, вже не вдасться. Проте в нашому прикладі частина ОП залишається вільною. Отож цілком правомірно створити для кожного підсписку не один, а два буфери, і заповнювати їх у той час, коли інший «надає» дані, що підлягають злиттю. Якщо один буфер вичерпано, то без будь-яких зволікань можна переключитися на інший.

Зазвичай, схема, запропонована у прикладі 15, самостійно не дає змоги зменшити загальний час, необхідний для зчитування усіх 100 000 блоків посортованих підсписків. Можна було б об’єднати стратегії попереднього зчитування з групуванням даних по циліндрах, якщо передбачено:

- зберігання посортованих підсписків на послідовних циліндрах з дотриманням того порядку розташування блоків на кожній доріжці і на доріжках у межах циліндра, який задано підсписком;

* читання доріжок або циліндрів, які представляють дані того чи іншого підсписку, загалом.

Приклад 16. Знову розглянемо другу фазу алгоритму TPMMS (див. приклад 9). Кількість вільної ОП дає змогу організувати по два буфери об'ємом, який дорівнює місткості доріжки, для кожного з 16-ти посортованих підсписків. Нагадаємо, що доріжка диска «Soaring 747» дає змогу зберегти 512 Кбайт даних, отож об'єм необхідної області ОП – 16 Мбайт. Можна починати зчитування доріжки з будь-якого сектора. Час, необхідний для завантаження даних з однієї доріжки, складається з одного періоду середнього часу пошуку і часу повного обертання диска, або 6,46+8,33=14,79 мс. Оскільки для зчитування даних усіх 16-ти посортованих підсписків потрібно звернутися до 3 136-ти доріжок 196-ти циліндрів, то загальний час завантаження інформації в ОП становить 46 с.

Показники продуктивності можна значно покращити, якщо використати для кожного підсписку по два буфери об'ємом, який має місткість циліндра. Оскільки циліндр у приводі «Soaring 747» має 16 доріжок, то нам необхідно 32 буфери по 4 Мбайт кожен, тобто 128 Мбайт. Однак пам'ять нашого піддослідного комп'ютера тільки 100 Мбайт, хоча 128 Мбайт –реальніше число.

За умови використання буферів розміром з об'ємом циліндра необхідно тільки один період часу пошуку в розрахунку на 1 циліндр. Загальний час пошуку і зчитування даних з 16-ти доріжок циліндра у цьому випадку становитиме 6,46+16∗8,33=140 мс, а час завантаження усіх даних з 196-ти циліндрів – 27 с.

Ще продуктивнішою є концепція створення крупномасштабних вихідних буферів об'ємом з доріжку або циліндр диска. Якщо параметри системи і характер застосування дають змогу здійснити запис даних такими крупними порціями, це допоможе зменшити до нуля час пошуку і обертання та досягти теоретично максимальної швидкості запису даних на диск. Якщо, наприклад, передбачити, що під час виконання другої фази процесу сортування використовується два вихідні буфери об'ємом по 4 Мбайт, то ми могли б «скидати» на циліндр посортовані дані з одного буфера паралельно із заповненням іншого. У цьому випадку час запису

становитиме 27 с (як час читання в прикладі 16), а час виконання другої фази загалом – менше, ніж 1 хв.

Продумане поєднання і вдала реалізація технології групування даних по циліндрах, попереднього зчитування і крупномасштабної вихідної буферизації здатні зменшити час сортування розглянутого масиву даних до 2-х хвилин (замість 74 хв, які необхідно витратити при використанні стратегії «як Бог дасть»).

5.6. Прийоми оптимізації дискових операцій: переваги і недоліки

Ми ознайомилися з п’ятьма підходами, які дають змогу підвищити продуктивність дискових систем:

1) групування даних по циліндрах диска;
2) використання декількох дискових пристроїв;
3) створення дзеркальних копій дисків;
4) впорядкування дискових операцій за алгоритмом ліфта;
5) попереднє зчитування і крупномасштабна буферизація даних.

Розглянули також приклади їхнього застосування до двох, деякою мірою протилежних, ситуацій:

I. Виконання першої фази алгоритму TPMMS, де блоки читаються і записуються у наперед відомій послідовності (в один і той же момент часу до диска звертається тільки один процес).
II. Активізація окремих процесів, пов’язаних, наприклад, з бронюванням авіаквитків або виконанням банківських операцій. Процеси протікають паралельно і використовують одні і ті ж диски і не допускають можливості хоча б якось адекватно передбачити їхні характеристики.

Коротко опишемо переваги і недоліки кожного із підходів.

Групування даних по циліндрах диска:

- *Перевага:* Ілюструє хороші показники для застосувань типу (I), де порядок звернень до диска наперед відомий і в будь-який момент часу з диском працює тільки один процес.

- *Недолік:* не може оптимізувати час виконання застосувань типу (II), де режим використання диска непередбачений.

Використання декількох дискових пристроїв:

- *Перевага*: збільшує швидкість обслуговування запитів на читання/запис дискових даних для застосувань обох типів.
- *Проблема*: запити на читання/запис даних, які передбачають звернення до одного і того ж диска, не можна виконати водночас, отож коефіцієнт підвищення продуктивності може виявитися нижчим, ніж кількість дисків.
- *Недолік*: вартість декількох дискових пристрої вища, ніж вартість одного пристрою тієї ж сумарної місткості.

Створення дзеркальних копій дисків:

- *Перевага*: збільшує швидкість обслуговування запитів на читання/запис дискових даних для застосувань обох типів; не спричинює проблеми зниження продуктивності, притаманної для використання декількох «звичайних» (не «дзеркальних») дисків.
- *Перевага*: забезпечує підвищення ступеня надійності зберігання даних.
- *Недолік*: при підвищеній вартості декількох дискових пристроїв фактично використовується об’єм одного з них.

Впорядкування дискових операцій за допомогою алгоритму ліфта:

- *Перевага*: скорочує середній час зчитування/запису блоків у ситуаціях, коли порядок запитів до диска непередбачуваний.

- *Проблема*: алгоритм проявляє найбільшу ефективність у тих випадках, коли існує безліч неопрацьованих запитів; однак у цьому випадку зростає середній час очікування на обслуговування кожного запиту.

Попереднє зчитування і крупномасштабна буферезація даних:

- *Перевага*: зменшує загальний час виконання дискових операцій, якщо порядок зчитування/запису даних відомий наперед, хоча характеристики процесу залежать від властивостей самих даних (як при виконанні другої фази алгоритму TPMMS).

- *Недолік*: вимагає нарощування об'єму оперативної пам'яті для створення додаткових буферів; не надає суттєвої вигоди, якщо потік запитів до диска випадковий.

Вправи для опрацювання

Вправа 8. Припустимо, що необхідно впорядкувати процес обробки запитів (перелічених на рис. 13) до дискового приводу моделі «Soaring 747» за умови, що у початковий момент часу блок головок розміщений навпроти циліндра з номером 8 000. Побудуйте розклад обслуговування запитів:

а) згідно з алгоритмом ліфта (початковим напрямом переміщення блока головок можна обрати будь-який);

б) за стратегією «першим прийшов – першим обслуговується».

Вправа 9. Нехай два дискових приводи «Soaring 747» використовують для зберігання дзеркальних копій даних. Припустимо, що замість того, щоб блоки даних могли зчитуватися з будь-якого диска, головки першого диска переміщаються тільки внутрішньою половиною циліндрів, а головки другого диска – зовнішньою. Зважаючи на те, що запити на читання адресовані до випадкових блоків, дайте відповіді на такі запитання:

1. Яким є середній час опрацювання одного запиту?

2. Наскільки відрізняється середній час читання блока у випадку, коли зазначене обмеження, що стосується можливості переміщення блоків головок дзеркальних дисків, зняте?

3. Які недоліки, зумовлені накладанням обмеження, Ви можете назвати?

Номер циліндра	Момент надходження (мс)
2000	0
12000	1
1000	10
10000	20

Рис. 13. Дані до вправи 8

Вправа 10. Якщо необхідно опрацювати запити на читання k блоків, які випадково розподілені в межах деякого циліндра, то який середній час обертання, необхідний для зчитування цих блоків?

Тема 6. Відмови дискових пристроїв

У цій і наступній темах ми розглянемо ситуації, пов'язані з відмовами дискових приводів, і способи подолання можливих наслідків. Нижче перелічені різні критерії відмов, класифіковані у порядку зростання ступеня їхньої важкості.

1. *Періодична відмова* – виникає періодично і виражається в тому, що одна спроба читання/запису сектора виявляється безуспішною, проте за умови її повторення операцію вдається виконати.
2. *Відмова запису* – часто виникає унаслідок збою енергоживлення в момент виконання операції запису сектора і проявляється у неможливості запису інформації та зчитування попереднього вмісту сектора.
3. *Руйнування носія* – частина сектора стає непридатною, отож операцію зчитування даних із сектора не можна виконати за жодних умов і незалежно від кількості спроб.

4. *Цілковита відмова диска* – раптова поломка, унаслідок якої подальша експлуатація пристрою стає неможливою.

Отож зупинимося на простій моделі відмов дисків, розглянемо технологію *контролю парності*, які дають змогу виявити ситуації періодичних відмов, і з'ясуємо, як організовуються *стійкі сховища*, які застерігають від можливості втрати даних під час відмови запису і руйнування носія.

6.1. Періодичні відмови

Сектори диска, зазвичай, містять додаткові надлишкові біти. Їхні значення дають змогу визначити коректність інформації, зчитаної з сектора або записаної в сектор.

За однією зі зручних моделей операцій зчитування дискових даних вважають, що функція читання повертає пару (w, s) значень, де w – власне дані, а s – *біт індикації стану* (*status bit*), який засвідчує, чи успішний результат операції, тобто чи w правильно відображає сектор. У випадку періодичних відмов функція читання може повертати «хибний» біт індикації стану декілька разів. Якщо ж повторювати операцію тривалий час (звичайна межа – до 100 разів), ймовірно, що функція все-таки поверне «істинний» біт, який підтвердить коректність зчитаних даних. Хоча можлива ситуація, коли біт індикації містить значення «істина», а зчитані дані насправді погані. Прийнявши відповідні міри, які полягають у використанні у межах сектора додаткових надлишкових бітів, можна знизити ймовірність виникнення такої події настільки, наскільки це необхідно.

Біт індикації стану корисний і у випадку виконання операції запису. Адже для перевірки коректності результату запису інформації в сектори можна здійснити повторне читання і порівняти зчитаний блок з записаним. Однак замість здійснення такої перевірки контролер дисків після читання може перевірити лише біт індикації стану: значення «істина» засвідчує, що запис виконано успішно; в іншому випадку записування необхідно повторити.

Зазначимо, що як і під час виконання операції читання, «істинне» значення біта індикації стану ще не є цілковитою гарантією правильності результату запису. З метою зниження

ймовірності виникнення подібних явищ необхідно застосовувати адекватні міри, про які розкажемо нижче.

6.2. Контрольні суми

На перший погляд, завдання перевірки «якості» даних у секторі при виконанні їхнього читання виглядає нерозв'язним. Однак справа є значно простішою: у сучасних дискових пристроях кожному секторові носія відповідають декілька додаткових бітів, які називають *контрольною сумою* (*check sum*). Їхнє значення встановлюється залежно від змісту даних, які зберігаються у секторі. Якщо функція читання виявляє невідповідність даних контрольній сумі, то вона повертає стан «хибна», а в протилежному випадку – «істина». Проте існує мала ймовірність виникнення такої події, коли біти даних зчитані неправильно, а їхня контрольна сума збігається з тією, яка відповідає коректним бітам. Цю ймовірність вдається знизити до будь-якого припустимого значення шляхом збільшення кількості бітів контрольної суми.

Контрольна сума в своїй простій формі ґрунтується на властивості *парності* (*parity*) усіх бітів сектора. Якщо в наборі бітів даних є *непарна* (*odd*) кількість одиниць, то біт парності має значення 1. Якщо ж кількість одиниць *парна* (*even*), то бітові парності присвоюється 0. Існує таке правило:

- кількість одиниць у наборі бітів даних, у тім числі біт парності, завжди парна.

Виконуючи записування даних у сектор, контролер дисків здатний обчислити значення біта парності і додати його до послідовності бітів, які записуються. Отже, кожен сектор міститиме парну кількість бітів з одиничками.

Приклад 17. Припустимо, що в секторі необхідно зберегти послідовність бітів даних 01101000. Кількість одиниць є непарною, тому біту парності треба присвоїти значення 1. Отримаємо 011010001. Якщо початкова послідовність набуде вигляду 11101110, то біт парності буде нульовим: 111011100. Зверніть увагу, що кожна з остаточних послідовностей містить парну кількість одиниць.

Довільна помилка, яка виникає під час спотворення одного біта послідовності даних, зумовлює отримання набору даних з непарною кількістю одиниць. Контролер диска одразу виявляє, що результат читання неправильний.

Очевидно, що пошкодженими можуть виявитися і декілька бітів сектора водночас. У цьому випадку з імовірністю 50% кількість одиниць у секторі виявиться парною і помилку не виявлять. Для того, щоб виявити помилку, яка охоплює декілька бітів, набір бітів парності необхідно розширити. Наприклад, якщо передбачити 8 бітів парності (по одному на один біт кожного байта), то ймовірність того, що жоден з бітів парності не зможе підтвердити помилку, оцінюється величиною 1/256. Якщо ж використовувати n незалежних бітів парності, то ймовірність виникнення незауваженої помилки становитиме $1/2^{n}$. Наприклад, якщо виокремити під контрольну суму 4 байти, то помилка залишиться невиявленою тільки в одному з 4 млрд випадків.

6.3. Стійкі сховища

Хоча механізм контрольних сум завжди дає змогу виявити збійні ділянки носія і помилки операцій читання/запису, він не в змозі забезпечити можливість виправлення помилок. Можлива ситуація, коли після виконання спроби запису виявляється, що попередній вміст сектора вже втрачено, а новий зчитати повторно вже не можна. Уявіть, в яку проблему все це може обернутися, якщо мова йтиме про зміни залишку на банківському рахунку, коли виявляються втраченими обидва значення залишку, старе і нове. З метою уникнення такої ситуації на базі одного чи декількох дисків можна організувати так зване *стійке сховище* (*stable storage*). Загальна ідея полягає в тому, що сектори диска (дисків) об'єднуються в пари, і кожна пара представляє вміст X одного сектора. Ми посилатимемось на сектори пари як на «лівий» (X_L) і «правий»(X_R), і вважатимемо, що обидва сектори містять достатню кількість бітів парності. Отож вважатимемо, що w відображається вірно, якщо для сектора X_L або X_R функція читання повертає значення w і «істинний» біт індикації стану.

Алгоритм запису даних у стійке сховище можна представити так:

1. Зберегти X в X_L. Перевірити, чи біт індикації стану містить «істина», тобто біти збереженої копії даних вірні. Якщо ні, то повторити операцію. Якщо після декількох спроб зберегти X в X_L не вдається, то вважати, що сектор пошкоджений, і замінити сектор X_L іншим.
2. Повторити дії п.1 для сектора X_R.

Процедура читання даних зі стійкого сховища така:

1. Зчитати значення X з X_L. Якщо функція читання повертає «хибне» значення біта індикації стану, то декілька разів повторити операцію. Якщо на деякій ітерації повертається «істинний» біт індикації стану, то вважати, що зчитані дані є значеннями X.
2. Якщо спроби читання з X_L виявилися невдалими, то повторити дії п.1 стосовно сектора X_R.

6.4. Подолання наслідків помилок

Використання ступенів захисту інформації (стійких сховищ) дає змогу відновити інформацію у таких випадках.

1. *Руйнування носія* (*media decay*). Якщо після збереження даних X у секторах X_L і X_R один із них стає непридатним через руйнування носія і стає нечитабельним, то відновити значення X вдається завжди, звернувшись до іншого – вцілілого – сектора. Якщо сектор X_R зіпсутий, а X_L – ні, то, згідно з алгоритмом читання, дані беруться з X_L, навіть не звертаючись до X_R. Факт виходу з ладу X_R буде виявлено під час наступної спроби запису нового значення X. Якщо ж, навпаки, зруйнований тільки сектор X_L, то необхідно зчитати X з X_R. Якщо ж водночас зруйновані обидва сектори, то значення X буде втрачено, проте ймовірність виникнення такої події дуже мала.
2. *Відмова запису* (*write failure*). Припустимо, що в момент запису значення X стався короткочасний збій системи (втрата електроживлення тощо). Імовірна ситуація, коли значення X в ОП відсутнє, а його копія пошкоджена в момент збереження на диску: наприклад, у половині

сектора є вже частина нового значення X, а в іншій половині – попереднє значення. Після запуску системи ми зможемо відновити або старе, або нове значення X. Можливі такі ситуації:

а) збій відбувся під час запису X_L, отож значення біта індикації стану для X_L буде «хибний»; оскільки до моменту збою нові дані X у сектор X_R ще не записувались, тому із X_R можна взяти старе значення X (для відновлення стану сектора X_L у нього треба скопіювати дані з X_R);

б) збій системи стався після завершення запису X_L, отож X_L містить нове значення X, яке за необхідності можна успішно зчитати; незалежно від стану сектора X_R, у нього треба скопіювати вміст X_L.

Вправи для опрацювання

Вправа 11. Обчисліть значення біта парності для таких бітових послідовностей:

1) 00111011;
2) 00000000;
3) 10101101.

Вправа 12. Для кожної з бітових послідовностей визначте значення двох бітів парності, перший з яких відповідає бітам на непарних позиціях послідовності, а другий – бітам на парних позиціях:

1) 00111011;
2) 00000000;
3) 10101101.

Тема 7. Відновлення даних за цілковитої відмови диска

У цій темі ми розглянемо найсерйозніший різновид відмов, коли дисковий пристрій стає непридатним, наприклад, унаслідок доторкання головки до поверхні пластини. Якщо інформацію попередньо не скопійовано на інший носій – наприклад, резервну магнітну стрічку чи дзеркальний диск, – то її безнадійно втрачено.

Для багатьох застосувань СКБД (таких як банківські та інші фінансові системи, системи керування польотами, бронювання квитків, ведення складського обліку і т. п.) це означає цілковитий крах.

Сьогодні існує чимало стратегій, покликаних знизити загрозу втрати даних через ламання дискових приводів. Усі вони, зазвичай, передбачають використання тієї чи іншої моделі надлишку і позначаються спільним терміном RAID (від *Redundant Array of Independent Disks – резервований масив незалежних дисків*).

7.1. Модель відмови дискових пристроїв

Розглянемо статистику цілковитої відмови дискових приводів. Проста одиниця вимірювання частоти виникнення цих явищ відома як *середній час напрацювання на відмову* (*mean time to failure*). Це проміжок часу, протягом якого 50% дискових пристроїв певної моделі цілковито вийде з ладу. Для сучасних моделей приводів магнітних дисків середній час напрацювання на відмову становить приблизно 10 років.

Щоб спростити сприйняття цього параметра, можна вважати, що одна десята всіх дискових пристроїв стає непридатною протягом одного року, і в цьому випадку відсоток *працездатності* (*survival*) задають як експоненту спадаючої функції. На практиці крива працездатності більше подібна на ту, яку зображено на рис. 14.

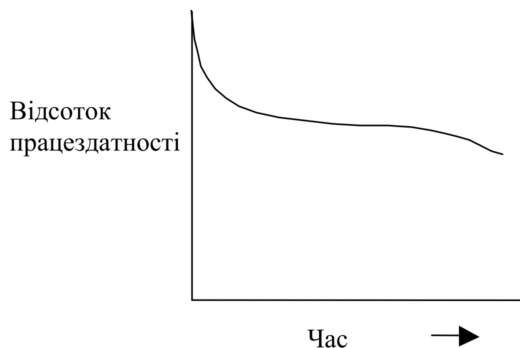


Рис. 14. Крива працездатності дискових пристроїв

Як і для більшості різновидів електронного обладнання, значна частина відмов дискових пристроїв трапляється на самому початку періоду їхньої експлуатації внаслідок незначних технологічних дефектів. Здебільшого, вироби з тими чи іншими порушеннями відбраковують ще на стадії технічного контролю і тестування. Проте окремі недоліки проявляються не одразу, а через деякий проміжок часу. Втішає одне: диск, який не вийшов з ладу на початку експлуатації, найімовірніше прослужить успішно ще багато років. Однак ближче до кінця життєвого циклу пристрою починають проявлятися негативні впливи таких чинників, як старіння матеріалів, стирання частин, які труться, і накопичення найдрібніших частинок пороху, що збільшує ймовірність виникнення несправності.

Середній час напрацювання на відмову – це, однак, не те саме, що середній час виникнення події втрати даних, оскільки використання схем RAID дає змогу зберігати інформацію цілою навіть за умови виходу дисків з ладу. Всі схеми RAID передбачають використання одного або декількох *дисків даних* у сукупності з одним або декількома *резервними дисками*, призначеними для зберігання службової інформації, склад якої цілковито визначається вмістом дисків даних. За умови поломки диска даних або резервного диска для відновлення інформації використовують вміст решти дисків масиву, так що непоправної втрати даних вдається уникнути.

7.2. Дзеркальні диски як засіб резервування

Простий підхід до розв’язання задачі підвищення ступеня надійності зберігання інформації полягає у використанні резервних дисків, які містять дзеркальні копії дисків даних. У цьому випадку диски даних і резервні диски цілковито взаємозамінні. Технологія використання дзеркальних дисків як засобу запобігання втрати даних, яку часто називають схемою RAID рівня 1, забезпечує значно вище значення середнього часу виникнення події втрати даних порівняно з середнім часом напрацювання на відмову, і наступний приклад це ілюструє. Зауважимо, що єдиною ситуацією, яка може спричинити до втрати даних при використанні дзеркальних дисків і деяких інших варіантів схем RAID, може

слугувати поломка другого диска масиву у момент, коли відбувається відновлення даних першого поламаного диска.

Приклад 18. Припустимо, що кожен диск характеризується 10-річним середнім часом напрацювання на відмову, отож імовірність його поломки у будь-якому році – 10%. Якщо диски дзеркальні, то при виході з ладу одного з них достатньо здійснити заміну диска та скопіювати дані з працюючого диска, і систему буде відновлено до її попереднього стану.

Єдина неприємність, яка може статися (хоча ймовірність цього дуже мала), стосується поломки диска з повноцінними даними у момент їхнього копіювання на новий диск. У цьому випадку інформацію втрачають цілковито без будь-якої можливості її відновлення.

Яка ймовірність саме такого перебігу подій? Припустимо, що процес заміни поламаного диска триває 3 години, тобто 1/8 доби, або 1/2 920 року. Оскільки, як ми вважаємо, цілковита відмова диска трапляється в середньому один раз у 10 років, то ймовірність того, що дзеркальний диск вийде з ладу в момент копіювання даних, оцінюється величиною *(1/10)*(1/2 920)=1/29 200*. Якщо середній час напрацювання на відмову одного диска є 10 років, то один з двох дисків може поламатися в середньому один раз за 5 років. У цьому випадку тільки одна з 29 200 подібних відмов може спричинити до втрати даних. Отже, середній час напрацювання на відмову, який спричинить до втрати даних у системі з двома дзеркальними дисками, становить 5*29 200 = = 146 000 років.

7.3. Блоки парності

Хоча технологія використання дзеркальних дисків є ефективним засобом пониження ймовірності втрати даних у зв’язку з цілковитою відмовою одного з пристроїв, її недоліком є те, що для дублювання кожного диска даних необхідний окремий резервний диск. Інший підхід, який називають схемою RAID рівня 4, передбачає використання тільки одного резервного службового диска незалежно від кількості дисків даних.

Вважатимемо, що всі диски ідентичні, отож їхні блоки можна умовно пронумерувати від 1-го до n. Зрозуміло, що всі блоки всіх дисків містять однакову кількість бітів. Наприклад, блоки розміром 16 384 байт, які використовували при звертанні до диска «Soaring 747», містять по 8 ∗ 16 384=131 072 бітів. Блок з номером i резервного диска складається з бітів парності для i-х блоків усіх дисків даних. Тобто послідовність j-х бітів i-х блоків усіх дисків, у тім числі резервного, повинна містити парну кількість одиничок, і це правило може бути виконане вибором відповідного (0 або 1) значення j-го біта i-го блока резервного диска. В основі цього правила лежить операція обчислення суми за модулем 2 (Додаток В). Сума за модулем 2 значень бітів дорівнює 0, якщо серед цих значень є парна кількість одиничок, і дорівнює 1, якщо кількість одиниць непарна.

Приклад 19. Розглянемо надзвичайно спрощений випадок, коли кожен блок містить всього 1 байт (8 бітів). Припустимо, що масив складається з трьох дисків даних (називатимемо їх «диск 1», «диск 2», «диск 3») і одного резервного диска («диск 4»). Зосередимо увагу, наприклад, на перших блоках усіх дисків. Якщо в них зберігаються бітові послідовності

диск 1 : 11110000,
диск 2 : 10101010,
диск 3 : 00111000,

тоді перший блок резервного диска повинен містити відповідні біти парності

диск 4 : 01100010.

Зверніть увагу, що всі чотири 8-бітові послідовності, які відповідають кожній позиції блоків трьох дисків даних і резервного диска, містять парну кількість одиниць: дві одиниці на позиціях 1, 2, 4, 5 і 7, чотири – на позиції 3 і жодної – на позиціях 6 і 8.

Зчитування. Операція зчитування блоків з дисків даних масиву RAID рівня 4 нічим не відрізняється від аналогічних процедур, які застосовують до звичайних дисків. Зчитувати вміст резервного диска, зазвичай, недоцільно, хоча формально не заборонено. За певних обставин деяку вигоду можна отримати від одночасного виконання запитів на читання, адресованих одному і

тому ж диску даних. Хоча збіг умов, за яких це стає вигідним, виникає доволі рідко.

Приклад 20. Припустимо, що під час читання деякого блока з першого диска даних надходить запит на читання іншого блока, наприклад, блока 1 з того ж диска. Якщо діяти в традиційній манері, перед опрацюванням другого запиту нам необхідно дочекатися завершення обслуговування першого. Якщо жоден з решти дисків масиву в цей момент не зайнятий, то цілком правомірно здійснити читання блока 1 з кожного диска, окрім першого, а потім обчислити вміст шуканого блока 1 першого диска за допомогою операції сумування за модулем 2. Якщо, зокрема, диски відповідають умовам прикладу 19, то можна здійснити читання блоків другого, третього і резервного дисків:

диск 2 : 10101010;
диск 3 : 00111000;
диск 4 : 01100010.

Потім, застосовуючи операцію додавання за модулем 2, отримують послідовність бітів, яка зберігається в блоці першого диска:

диск 1 : 11110000.

Записування. Під час виконання операції записування нового блока на диск даних необхідно змінити не тільки цей блок, але й відповідний блок резервного диска, щоб він знову містив правильні біти парності, які відповідають бітам однойменних блоків усіх дисків даних. Тривіальний і прямолінійний підхід полягає у зчитуванні відповідних блоків n дисків даних, обчисленні суми за модулем 2 для кожного розряду блоків і збереженні отриманого результату у блоці резервного диска. У цьому випадку доведеться виконати операції запису вихідного блока, читання $n-1$ блоків з решти дисків даних і запису блока на резервний диск, тобто разом $n+1$ операцій введення/виведення.

Ефективніший спосіб передбачає маніпуляції зі старим і новим вмістом i-го блока, який записується, і блока резервного диска. Якщо обчислити суму за модулем 2 старого і нового значень блока, що записується, то ми зможемо визначити, в яких послідовностях бітів для кожної позиції i-х блоків дисків даних змінилася кількість

одиниць. Оскільки кількості можуть змінюватися тільки на 1, парна кількість одиниць стає непарною. Якщо змінити вміст відповідних бітів *i*-го блока резервного диска, то кількість одиниць в усіх послідовностях знову стане парною. Для виконання цих дій достатньо виконати чотири дискові операції:

1) зчитати старе значення блока даних, який необхідно записати;
2) зчитати значення відповідного блока резервного диска;
3) записати новий блок даних;
4) виконати обчислення і зберегти змінений блок резервного диска.

Приклад 21. Нехай перші блоки трьох дисків виглядають так, як у прикладі 19:

диск 1 : 11110000;
диск 2 : 10101010;
диск 3 : 00111000.

Припустимо, що значення блока диска 2 змінюється з 10101010 на 11001100. Додавання за модулем 2 старого і нового значень блока дає 01100110, що засвідчує, що в першому блоці резервного диска необхідно змінити значення бітів у позиціях 2, 3, 6, 7. Блок резервного диска необхідно зчитати: 01100010 і додати за модулем 2 з отриманою сумою 01100110, що дає в підсумку 00000100. Після запису блока на другий диск даних і модифікації блока резервного диска блоки усіх дисків набудуть значень:

диск 1 : 11110000;
диск 2 : 11001100;
диск 3 : 00111000;
диск 4 : 00000100.

Зауважимо, що всі послідовності бітів для кожної позиції блоків знову міститимуть парну кількість одиниць.

Відновлення даних після відмови диска. Розглянемо дії, завдяки яким можна домогтися відновлення даних після цілковитої відмови одного з дисків масиву RAID рівня 4. Якщо вийшов з ладу резервний диск, то достатньо встановити новий пристрій, послідовно зчитати однойменні блоки всіх дисків, обчислити контрольні суми для бітових послідовностей кожної позиції і зберегти отри-

мані результати як блоки резервного диска. Якщо трапилась аварія з одним з дисків даних, то достатньо замінити його новим й обчислити дані на основі інформації решти дисків. Правило обчислення втрачених даних дуже просте і діє незалежно від того, який з дисків (резервний чи диск даних) став непридатним. Його формулюють так:

- значення біта блока одного диска дорівнює сумі за модулем 2 бітів, розміщених на тій же позиції, усієї решти дисків.

Якщо сума за модулем 2 дорівнює 1 (тобто кількість одиниць непарна), то і відновлюваному бітові треба присвоїти значення 1, і навпаки.

Приклад 22. Розглянемо той же масив дисків, що і в прикладі 19, і припустимо, що з ладу вийшов диск 2. Розглянемо, як виглядають перші блоки всіх чотирьох дисків:

диск 1 : 11110000;
диск 2 : ????????;
диск 3 : 00111000;
диск 4 : 01100010.

Обчислюючи суми за модулем 2 бітів кожної позиції, отримаємо блок другого диска в його первинному вигляді: 10101010.

7.4. Масиви RAID рівня 5

Схема RAID рівня 4 ефективно забезпечує збереження даних доти, доки водночас не вийде з ладу *декілька* дисків масиву. Вузьким місцем моделі RAID рівня 4 також є запис нового блока даних. Кожного разу доводиться зчитувати і зберігати блок резервного диска. Якщо кількість дисків даних дорівнює n, то кількість операцій запису на резервний диск в n разів перевищуватиме середню кількість операцій запису на будь-який з дисків даних.

Як проілюстровано у прикладі 22, правило відновлення даних однакове як для всіх дисків даних, так і для резервного диска, і зводиться до обчислення сум за модулем 2 бітових послідовностей. Отже, вважати один диск службовим, а інші – «звичайними» не обов’язково. Кожен диск може виконувати роль резервного стосовно до певних наборів блоків. Саме цю ідею покладено в основу схеми дискового масиву RAID рівня 5.

Наприклад, якщо є набір з $n+1$ диска, які перенумеровано від 0 до n, то можна інтерпретувати i-й циліндр j-го диска як резервний у тому випадку, коли j є залишком від ділення i на $n+1$.

Приклад 23. Розглянемо масив із 4-х дисків і покладемо n=3. Диск з номером 0 може виконувати функцію резервного для циліндрів з номерами 4, 8, 12 і так далі, оскільки залишок від ділення цих номерів на загальну кількість дисків (4) дорівнює нулю. Аналогічно, диск 1 може бути резервним для циліндрів 1, 5, 9 і так далі, диск 2 – для циліндрів 2, 6, 10 і так далі, диск 3 – для циліндрів 3, 7, 11 і так далі Як наслідок – коефіцієнт завантаження усіх дисків виявиться однаковим. Якщо операція запису всіх блоків виконується з однаковою частотою, то ймовірність того, що операцію адресовано певному дискові, становить 1/4. Якщо ж операцію адресовано іншому дискові, то з імовірністю 1/3 він буде резервним для блока, що зберігається. Отже, кожен з чотирьох дисків братиме участь у записуванні з імовірністю 1/4+3/4∗1/3=1/2.

7.5. Відновлення даних після відмови декількох дисків

Технології відновлення інформації після відмови декількох дисків масиву (резервних або дисків даних) реалізовано у схемі RAID найвищого рівня – *RAID рівня 6*. Вони ґрунтуються на теорії *кодів з виправлення помилок* (*error-correcting codes*) і дають змогу подолати наслідки аварії довільного ступеня важкості – лише б кількість резервних дисків була достатньою.

Розглянемо короткий приклад, що стосується одночасної відмови двох дисків і використання стратегії відновлення даних на основі простого коду з виправлення помилок, відомого як *код Хеммінга* (*Hamming code*).

Звернемось до масиву, який складається з семи дисків, перенумерованих від 1 до 7. Перші чотири диски є дисками даних, а диски 5-7 виконують функцію резервних. Взаємозв’язки між дисками даних і резервними дисками масиву описано бінарною матрицею 3∗7, представленою на рис. 15. Звернемо увагу на наступне:

а) матриця містить всі можливі 3-бітові послідовності одиниць і нулів, окрім набору з трьох нулів;

б) стовпці, які відповідають резервним дискам, містять по одній одиниці;
в) стовпці, які відповідають дискам даних, містять по дві одиниці.

Зміст кожного з трьох рядків матриці такий. Якщо поглянути на біти, які відповідають кожному з семи дисків масиву, і залишити тільки диски, яким у рядку відповідають одинички, то сума за модулем 2 значень бітів для цих дисків дорівнюватиме нулю. Іншими словами: диски, яким відповідають одинички в кожному з рядків матриці, вважаються об'єднаними в масив RAID рівня 4. Отож для обчислення бітів одного з резервних дисків досить знайти рядок матриці, в якому цей диск позначено одиницею, і здійснити додавання за модулем 2 значень відповідних бітів інших дисків, позначених одиницями в тому ж рядку.

	Диски даних	Резервні диски
Номер диска	1 2 3 4	5 6 7
	1 1 1 0	1 0 0
	1 1 0 1	0 1 0
	1 0 1 1	0 0 1

Рис. 15. Матриця резервування для системи, яка передбачає відновлення даних після одночасної відмови двох дисків

Якщо звернутися до матриці на рис. 15, то згадане правило означає:

1) біти диска 5 – це сума (за модулем 2) відповідних бітів дисків 1, 2 і 3;
2) біти диска 6 – це сума (за модулем 2) відповідних бітів дисків 1, 2 і 4;
3) біти диска 7 – це сума (за модулем 2) відповідних бітів дисків 1, 3 і 4.

Тож розглянемо, як конкретний набір бітів матриці вказує на простий спосіб відновлення інформації після одночасної відмови будь-яких двох дисків.

Зчитування. Дані з будь-якого диска масиву зчитують звичайним способом. Зчитувати інформацію з резервних дисків з метою використання у прикладних задачах недоцільно, хоча формально це не заборонено.

Записування. Способи запису аналогічні до тих, які згадували у пункті 7.3, однак у цьому випадку операція запису може стосуватися декількох резервних дисків. Щоб записати блок на будь-який з дисків даних, необхідно додати за модулем *2* попереднє і нове значення блока, який оновлюється, а потім отриманий результат додати (за модулем 2) до значень відповідних блоків усіх резервних дисків, позначених одиничками у рядках, в яких позначено одиницею диск, що модифікується.

Приклад 24. Знову розглянемо масив RAID рівня 6, який складається з семи дисків з матрицею резервування (рис. 15) і покладемо, що блоки мають довжину 8 бітів. Припустимо також, що перші блоки всіх семи дисків виглядають так, як проілюстровано на рис. 16. Зауважте, що блок диска 5 є сумою за модулем 2 блоків перших трьох дисків, рядок 6 є результатом додавання за модулем 2 рядків 1, 2 і 4, а останній рядок – це сума за модулем 2 рядків 1, 3 і 4.

Диск	Вміст
1)	11110000
2)	10101010
3)	00111000
4)	01000001
5)	01100010
6)	00011011
7)	10001001

Рис. 16. Значення перших блоків усіх дисків

Нехай новим значенням першого блока диска 2 має стати байт 00001111. Виконавши додавання за модулем 2 цієї бітової послідовності з попереднім вмістом блока 10101010, отримаємо 10100101. Якщо звернутися до стовпця матриці на рис. 15, який

відповідає диска 2, то можна побачити, що він містить одиниці в перших двох рядках. Оскільки в рядках 1 і 2 одиницями позначені резервні диски 5 і 6, то треба додати за модулем 2 вміст перших блоків дисків 5 і 6 з обчисленим вище байтом 10100101. Результат проілюстровано на рис. 17.

Диск	Вміст
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	11000111
6)	10111110
7)	10001001

Рис. 17. Значення перших блоків усіх дисків після оновлення 2-го і відповідних резервних дисків

Зверніть увагу, що нові значення всіх блоків знову задовольняють вимогам матриці з рис. 15: суми за модулем 2 значень блоків дисків, позначених одиницею у кожному рядку матриці, дорівнюють нулю.

Відновлення даних після одночасної відмови двох дисків. Тепер розглянемо, як схема резервування даних, розглянута вище, дає змогу подолати наслідки одночасної відмови двох дисків. Позначимо диски, що вийшли з ладу, як a і b. Оскільки усі стовпці матриці на рис. 15 різні, то необхідно знайти деякий рядок r, в якому вміст комірок стовпців a *і* b не збігається. Припустимо, що на перетині рядка r зі стовпцем a є 0, а в комірці з координатами r і b – 1.

Для обчислення втраченого значення блока диска b необхідно попарно додавати за модулем 2 біти блоків усіх дисків, крім b, які містять одиниці в рядку r матриці. Зауважимо, що серед цих дисків диска a немає, отож усі блоки будуть доступними. Оскільки кожен стовпець матриці містить одиниці (хоча б в одному рядку), то можна скористатися одним з цих рядків для обчислення блока

диска *a*, обчислюючи суму за модулем 2 значень блоків тих дисків (окрім *a*), які в цьому рядку позначено одиницею.

Приклад 25. Припустимо, що майже в один і той же момент часу диски 2 і 5 вийшли з ладу. Поглянувши на матрицю, зображену на рис. 15, визначимо, що значення у комірках стовпців, що відповідають дискам 2 і 5, розрізняються в рядку 2, де диск 2 позначено одиницею, а диск 5 – нулем. Отож для відновлення блока диска 2 можна додати за модулем 2 блоки трьох інших дисків, позначених одиницями в рядку 2 – дисків 1, 4 і 6. Зауважимо, що жоден з цих дисків не є пошкодженим.

Якщо припустити, що перші блоки всіх дисків спочатку виглядали так, як проілюстровано на рис. 17, то ситуацію після відмови дисків 2 і 5 можна схематично подати у формі таблиці (рис. 18):

Диск	Вміст
1)	11110000
2)	????????
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

Рис. 18. Значення перших блоків усіх дисків після відмови дисків 2 і 5

Якщо додати за модулем 2 значення блоків дисків 1, 4 і 6 (11110000, 01000001 і 10111110, відповідно), то отримаємо відсутній байт 00001111 диска 2. Очевидно, що він збігається з тим, який міститься у другому рядку на рис. 17. Тепер ситуація виглядає так, як на рис. 19.

Зауважимо, що диск 5 позначено одиницею в рядку 1 матриці на рис. 15. Отож для відновлення блока даних диска 5 необхідно додати за модулем 2 значення блоків трьох інших дисків, які позначено одиницею в рядку 1 – дисків 1, 2 і 3. Шукана сума дорівнює 11000111. Знову очевидно, що вона збігається з вихідним вмістом блока 5-го диска, проілюстрованого на рис. 17.

Диск	Вміст
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

Рис. 19. Значення перших блоків усіх дисків після відновлення диска 2

Додаткові зауваження до схеми RAID рівня 6:

1. Цілком можливо об'єднати ідеї, покладені в основу схем RAID рівнів 5 і 6, щоб розподілити функції резервування за номерами блоків або циліндрів з метою зменшення навантаження на виокремлені резервні диски.
2. Схема RAID 6-го рівня не є обмеженою чотирма дисками даних. Загальна кількість дисків може бути на одиницю меншою потрібного ступеня числа 2, наприклад, 2^k-1, де k дисків є резервними, а решта 2^k-k-1 – це диски даних. Ступінь резервування зростає приблизно як логарифм від кількості дисків даних. Для довільного k не важко побудувати матрицю, подібну до представленої на рис. 15, стовпці якої містять усі різні k-бітові послідовності нулів і одиниць, за винятком набору з k нулів. При цьому стовпці з єдиною одиницею відповідають резервним дискам, а всі інші – дискам даних.

Вправи для опрацювання

Вправа 13. Припустимо, що використовується схема створення дзеркальних дисків, аналогічна до наведеної в прикладі 18, причому ймовірність поломки кожного диска у будь-якому році дорівнює 4%, а на заміну поламаного диска потрібно 8 годин. Який у цьому випадку середній час напрацювання на відмову, що спричинить до втрати даних?

Вправа 14. Нехай диск може вийти з ладу протягом певного року з імовірністю F і на його заміну необхідно H годин. Визначити:

1) середній час виникнення події втрати даних, записаний як функція від F і H, якщо використовується механізм дзеркальних дисків;

2) середній час виникнення події втрати даних, записаний як функція від F, H і N, якщо використовується масив RAID рівня 4 або 5, що містить N дисків.

Вправа 15. Припустимо, що група дзеркальних дисків складається з трьох пристроїв, тобто кожен з них містить ідентичну інформацію. Якщо значення ймовірності виходу одного диска з ладу в будь-якому році дорівнює F, а на його заміну необхідно H годин, то який середній час виникнення події втрати даних?

Вправа 16. Розглянемо масив RAID рівня 4 з чотирьох дисків даних і одного резервного диска. Припустимо, що кожен блок будь-якого диска складається з одного байта. Опишіть механізм отримання даних для резервного диска і визначте вміст блока резервного диска, якщо відповідні блоки дисків даних такі:

а) 01010110, 11000000, 00111011, 11111011;
б) 11110000, 11111000, 00111111, 00000001.

Вправа 17. Розглянемо масив RAID рівня 4 з чотирьох дисків даних і одного резервного диска. Припустимо, що вийшов з ладу диск 1. Виконайте відновлення вмісту блока цього диска, враховуючи такі вимоги:

1) диски 2-4 містять блоки 01010110, 11000000 і 00111011, відповідно, а резервний диск – блок 11111011;
2) диски 2-4 містять блоки 11110000, 11111000 і 00111111, відповідно, а резервний диск – блок 00000001.

Вправа 18. Припустимо, що вміст блока першого диска масиву, розглянутого у прикладі 24, змінено на 10101010. Як треба модифікувати блоки інших дисків?

Вправа 19. Дано масив RAID рівня 6, утворений відповідно до схеми, розглянутої у прикладі 24. Блоки чотирьох дисків даних

масиву містять байти 00111100, 11000111, 01010101 і 10000100, відповідно. Визначити:

1) як повинні виглядати відповідні блоки резервних дисків;
2) якщо значення блока третього диска змінено на 10000000, то які дії для модифікації вмісту решти дисків треба виконати?

Вправа 20. Опишіть порядок дій, які необхідно виконати з метою відновлення даних у масиві RAID рівня 6 з сімома дисками, розглянутому в прикладі 24, при виході з ладу таких дисків:

а) 1 і 7;
б) 1 і 4;
в) 3 і 6.

Вправа 21. Запропонуйте схему масиву RAID рівня 6 з п’ятнадцятьма дисками, чотири з яких є резервними, узагальнивши матрицю Хеммінга для 7-ми дисків, зображену на рис. 15.

Вправа 22. Запропонуйте схему масиву RAID рівня 6 з десятьма дисками таку, яка даватиме змогу відновлювати інформацію за одночасного виходу з ладу трьох дисків.

ПРЕДСТАВЛЕННЯ ЕЛЕМЕНТІВ ДАНИХ НА ВТОРИННИХ ПРИСТРОЯХ ЗБЕРІГАННЯ

Обговоримо модель *блоків* даних, які розміщуються на носіях вторинних пристроїв зберігання, проте вже на високому структурному рівні. Розпочнемо з визначення того, як у вторинних сховищах представляються відношення або множини об'єктів.

- *Атрибути* відображають як байтові послідовності постійної або змінної довжини, які називають *полями* (*fields*).
- Поля знову ж об'єднуються в набори даних постійної чи змінної довжини, які позначають терміном *запис* (*record*) і відповідають кортежам або об'єктам.
- *Записи* зберігаються у фізичних *блоках (blocks)*. Для формування блоків використовують різні *структури даних*, корисні, передусім, у тих ситуаціях, коли зміст даних змінюється.
- Набори записів, які відповідають відношенням або екземплярам класів, зберігаються як колекції блоків, які називають *файлами*[1] (*files*). З метою забезпечення можливості ефективної обробки запитів і команд модифікації таких колекцій файли доповнюють однією або декількома структурами *індексів* (*index*).

Тема 8. Елементи даних і поля

Спочатку проілюструємо, як значення атрибутів – відправні елементи інформації, які зберігаються в базах даних, що керуються реляційними або об'єктно-орієнтованими системами, – представляють у формі полів. Зупинимося також на способах об'єднання полів у більші структури – записи, блоки і файли.

[1] Термін *файл* у контексті СКБД має ширше поняття, ніж однойменний термін в операційних системах. Хоча файлом бази даних може бути і неструктурований потік байтів, найчастіше такі файли складаються з наборів блоків, організованих у строго визначеному порядку, які допускають індексування і застосування ефективних спеціалізованих методів доступу до їхнього змісту.

8.1. Представлення елементів реляційних баз даних

Припустимо, що в контексті реляційної бази даних засобами SQL-команди *CREATE TABLE*, яку представлено на рис. 20, оголошено відношення з іменем *MovieStar*. Задача структурування і зберігання відношення, описаного цією командою, покладається на відповідну СКБД. Оскільки відношення є множиною кортежів, а кортежі за означенням подібні із записами або «структурами» (у розумінні, яке прийнято в мові програмування C++), можна припустити, що кожен кортеж буде збережено на диску як окремий запис. Запис займає деякий дисковий блок або його частину, а в середині запису значенню кожного компонента кортежу відповідає певне поле.

```
1)      CREATE TABLE Movie Star (
2)           name CHAR(30) PRIMARY KEY,
3)           address VARCHAR (255),
4)           gender CHAR(1),
5)           birthdate DATE
        );
```

Рис. 20. Приклад відношення, оголошеного мовою SQL

Хоча на перший погляд все виглядає просто і прозоро, насправді тут не все так просто, отож наберемось терпіння і спробуємо відповісти на кілька запитань.

1. Як представляють у формі полів значення типів даних SQL?
2. Як кортежі зберігаються у формі записів?
3. Як набори записів або кортежів відображаються у фізичних блоках пам'яті?
4. Як відношення оформляються і зберігаються у вигляді колекцій блоків?
5. Що робити, якщо кортежам відповідають записи різної довжини або записи точно не поміщаються в межах блоків?

6. Що відбувається, коли після оновлення вмісту деякого поля розмір запису змінюється? Як відшукати необхідний простір усередині блока, передусім у тих випадках, коли довжина запису збільшується?

Першій проблемі присвячено цю тему. Два наступні питання висвітлено у темі 9, останні – у темах 11 і 12, відповідно. Четверта тема, яка зачіпає такі способи представлення відношень, які забезпечують можливість ефективного доступу до кортежів цих відношень, вивчатимемо у розділі 3.

Заслуговують уваги і методи низького рівня представлення даних деяких спеціальних різновидностей, що активно використовуються в сучасних об'єктно-реляційних і об'єктно-орієнтованих системах, а саме – ідентифікаторів об'єктів (або покажчиків на записи) та об'єктів типу «BLOB» (наприклад, відеофрагментів формату MPEG об'ємом у декілька гігабайтів). Цю інформацію буде викладено у темах 10 і 11.

8.2. Представлення об'єктів

У деякому наближенні об'єкт можна сприймати як кортеж, а його поля – як компоненти атрибутів. Кортежі в об'єктно-реляційних базах даних вельми нагадують кортежі «звичайних» (реляційних) баз даних. Однак існує кілька цікавих обставин, про які ми не згадували у пункті 8.1.

1. Об'єкт можна асоціювати з функціями, які називають *методами*. Код методів є частиною схеми класу, до якого належить об'єкт.
2. Кожен об'єкт, зазвичай, володіє власним *ідентифікаційним номером* (*object identity* – OID), який можна трактувати, як певну адресу в деякому глобальному адресному просторі, що однозначно визначає цей об'єкт. Об'єкти нерідко зв'язують з іншими об'єктами. Подібні зв'язки описують покажчиками або списками покажчиків.

Код методів зберігається у схемі, отож належить базі даних загалом, а не окремим об'єктам. З метою забезпечення можливості звертання до методу, однак, необхідно, щоб запис, який представ-

ляє об'єкт, містив поле, що засвідчує його приналежність до певного класу.

Способи представлення адрес – чи це є ідентифікаційний номер об'єкта чи посилання на інший об'єкт – обговорюватимемо у темі 10. Зв'язки – як їх розглядають у мові ODL (*Object Definition Language – мова визначення об'єктів*), – належать об'єктові, й інформацію про них необхідно якось зберігати в базі даних. Оскільки кількість взаємозв'язаних об'єктів, які підлягають створенню, наперед невідома (щонайменше, у тих випадках, коли йдеться про об'єкти, з'єднані зв'язками типу «багато до багатьох» або які належать до категорії «багато» зв'язків вигляду «багато до одного»), то для опису зв'язків випадає користуватися записами змінної довжини (детальніше про це – у темі 11).

8.3. Представлення елементів даних полями

Розпочнемо з вивчення способів зберігання елементів даних, які є основними типами SQL (Додаток Г). Їх представляють як поля записів. Важливо розуміти, що всі дані описуються послідовностями байтів. Наприклад, значення атрибута типу *INTEGER,* зазвичай, відображається у двох або чотирьох байтах, а для типу *FLOAT* необхідно від 4-х до 8-ми байтів. Цілі і дійсні числа представлено наборами бітів, які інтерпретуються апаратним забезпеченням комп'ютера так, щоб зробити можливим виконання традиційних арифметичних операцій.

Символьні рядки постійної довжини. Найпростіший різновид рядків символів описується в SQL – типом *CHAR(n)*, який визначає множину рядків постійної довжини *n*. Поле, що відповідає атрибуту подібного типу, є масивом з *n* байтів (це справедливо для 8-бітових наборів символів ASCII, адже символ у форматі Unicode представляється як 16 бітів або 2 байта). Якщо значенням атрибута є коротший рядок, то в кожну «зайву» позицію поля заноситься спеціальний *незначущий* (*pad*) символ, 8-бітовий код якого не збігається з кодом будь-якого з тих символів, які дозволено включати в рядки SQL.

Приклад 26. Якщо оголошення атрибута *A* засвідчує його приналежність до типу *CHAR(5)*, то поля, які відповідають

атрибутові, в усіх кортежах становитимуть масиви з 5-ти символів. Якщо компонент атрибута *A* в одному з кортежів повинен містити '*cat'*, то масив виглядатиме так

c a t ⊥ ⊥

Тут ⊥ представляє незначущий символ-заповнювач, код якого заноситься у четвертий і п'ятий байти-елементи масиву. Зауважимо, що символи одинарних лапок, які використовують для позначення рядків у SQL-програмах, до значення рядка не зачисляють, у відповідному полі їх не зберігають.

Символьні рядки змінної довжини. Інколи довжина рядкових значень компонента певного атрибута відношення може змінюватися у широких межах. У таких випадках атрибути, зазвичай, відносяться до типу *VARCHAR(n).* Загалом, при реалізації подібних типів під значення можна відводити n+1 байт незалежно від справжнього розміру рядка. Отож SQL-тип *VARCHAR* насправді є полем фіксованої довжини, хоча розмір вмісту може змінюватися. Особливості полів і записів даних змінної довжини ми детально розглянемо у темі 11.

Згадаємо коротко два загальних способи представлення рядків типу *VARCHAR.*

1. *Довжина плюс вміст.* Система організує масив з n+1 байта. У першому байті як 8-бітове ціле число зберігається кількість байтів у рядку. Довжина рядка не може перевищувати n, а величина n, також не повинна бути більшою від 255 – інакше довжину рядка не можна представити за допомогою одного байта. Другий і наступні байти відображають символи рядка. Якщо рядок складається з меншої кількості символів, решта байтів масиву не може інтерпретуватися як частина вмісту, отож нею ігнорують.
2. *Рядки, які закінчуються символом null.* У цьому випадку також формується масив з n+1 байта. Масив заповнюється символами рядка, а в елемент, розташований за останнім байтом вмісту рядка, заноситься спеціальний символ *null*, який не збігається з жодним із символів, що дозволено

використовувати всередині рядків SQL. Як і в попередньому випадку, рештою байтів масиву ігнорують.

Приклад 27. Припустимо, що оголошення атрибута *A* містить тип *VARCHAR (10).* У записах, які представляють кортежі відношення з атрибутом *A*, створюються поля-масиви довжиною 11 байтів для зберігання компонентів атрибута *A*. Нехай значенням одного з компонентів є рядок ‘cat’. Тоді, згідно з першим підходом, у початковий байт масиву заноситься число 3, яке задає довжину рядка, а в наступні 3 байти – символи рядка. Значеннями решти семи байтів ігнорують. У цьому випадку масив набуде вигляду:

3 c a t

Зауважимо, що «3» – це 8-бітове ціле число (тобто 00000011), а не символ ‘3’.

Керуючись другим підходом, система заповнить три перших байти масиву символами рядка, а в четвертий занесе символ *null*:

c a t ⊥

Значення дати і часу. Значення дати, зазвичай, представляється як символьні рядки постійної довжини і принципово нічим не відрізняється від звичайних рядків символів подібного формату. Часові величини передбачають аналогічне представлення. У конкретних реалізаціях SQL значенням дати і часу відповідають спеціальні типи даних *DATE*(‘1948-05-14’) *і TIME* (‘15:00:03.1415’). Формат значень *TIME* належить до категорії рядків довільної довжини, оскільки він дає змогу зберігати значення часу з точністю до дробових часток секунди. Отже, існує два альтернативні засоби опису значень часу:

1. Система здатна обмежити точність представлення часових величин так, ніби вони належать до типу *VARCHAR(n)*, де *n* – максимально допустима довжина рядка з описом значення часу.
2. Часові значення можуть зберігатися і використовуватися як рядки дійсно змінної довжини (за деталями звертайтесь до теми 11).

Бітові послідовності. Послідовності бітів, описані значеннями SQL-типу *BIT(n)*, ділять на групи з 8-ми бітів у кожній, які

упаковуються в байти. Якщо *n* не ділиться на 8 без остачі, то зайвими бітами останнього байта ігнорують. Наприклад, бітову послідовність 010111110011 легко описати двома байтами – 01011111 і 00110000. Останні чотири нулі другого байта не є частиною значення поля.

У частковому випадку байт можна використовувати для представлення логічної величини, тобто вмісту єдиного біта – 10000000 (*true*) і 00000000 (*false*). У деяких ситуаціях вигіднішим є представлення, коли величини *true* і *false* відрізняються в усіх бітах байта – відповідно, 11111111 і 00000000.

Перелічені типи. Інколи зручне використання мають такі атрибути, вміст яких вибирають із фіксованих множин значень. Подібним значенням присвоюють символьні імена. Тип, в оголошенні якого зазначають набір символьних імен, називають *переліченим* (*enumerative type*). Прикладом перелічених типів слугують множини назв днів тижня {SUN, MON, TUE, WED, THU, FRI, SAT} або кольори {RED, GREEN, BLUE, YELLOW}.

Значення перелічених типів можна представити цілочисловими кодами з використанням необхідної кількості байтів. Наприклад, для опису значення RED можна взяти число 0, значення GREEN – 1, BLUE – 2; YELLOW – 3. Для представлення подібної множини цілих чисел достатньо двох бітів – відповідно, 00, 01, 10, 11. Зрештою, навіть у тих випадках, коли множина значень переліченого типу мала, то зручніше використовувати цілі байти, а не їхні частини. За такого підходу значенню YELLOW відповідатиме байт 00000011. Загалом за допомогою одного байта можна представити до 256-ти різних значень переліченого типу. За необхідності беруть більше байтів (наприклад, цілочисловий тип SHORTINT).

Тема 9. Записи

Поговоримо про способи формування *записів* (*record*), які об'єднують у себе групи полів. Подібні питання розглядатимемо також у темі 11, присвяченій полям і записам змінної довжини.

Загалом кожному типу запису, використаному в контексті бази даних, необхідно ставити у відповідність *схему*, яка також

зберігається. Схема містить опис імен і типів даних полів, а також величини їхнього *зміщення* (*offset*) від початку запису. Інформацію схеми використовує система за необхідності звертання до компонентів запису.

9.1. Конструювання записів постійної довжини

Кортежі відношень представляються записами, складеними з полів різних типів. У найпростішому випадку, коли всі поля мають постійну довжину, то запис можна отримати простим зчепленням полів.

Приклад 28. Звернемось до оголошення відношення *MovieStar*, наведеного на рис. 20. Воно налічує чотири поля:

1) *name* – 30-байтовий рядок символів;
2) *address* – символьний рядок типу *VARCHAR(255)*, представлений масивом із 256-ти байтів;
3) *gender* – один байт, який містить код одного з двох допустимих символів – ‘F’ або ‘M’;
4) *birthdate* – величина типу DATE (вважають, що тут використано 10-байтове представлення значень дат SQL).

Отже, загальна довжина запису типу *MovieStar* становить 30+256+1+10=297 байтів. Запис схематично зображено на рис. 21. У нижній частині рисунка подано значення зміщень – кількість байтів від початку запису – для кожного поля. Зміщення поля *name* дорівнює 0 (перший байт запису має номер 0), початок поля *address* розташований від початку запису на 30 байтів, *gender* – на 286 байтів, а *birthdate* – на 287 байтів.

Деякі архітектури комп’ютерів підтримують ефективніші алгоритми зчитування і запису елементів даних, якщо останні розміщуються в ОП за адресами з номерами, кратними 4 (або 8, якщо комп’ютер оснащений 64-бітовим процесором). Для певних типів даних, таких як цілі числа, вимога розміщення за адресами, кратними 4, може виявитися обов’язковою, а для інших (наприклад, чисел з плаваючою комою), можливо, такою ж умовою може бути «прив’язка» до адрес з множником 8.

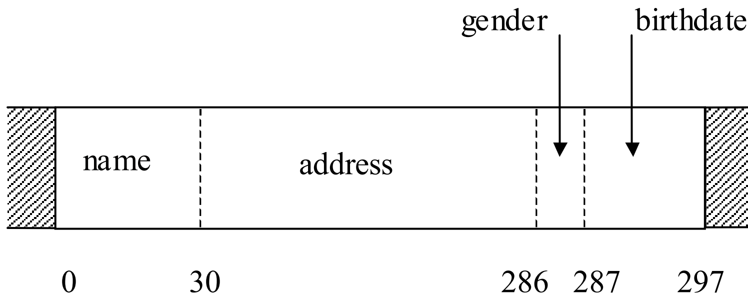


Рис. 21. Запис типу MovieStar

Хоча кортежі відношення постійно зберігаються на диску, а не в ОП, не враховувати подібні обставини не можна, оскільки під час зчитування дискового блока в ОП перший байт блока буде чітко розташований за адресою, кратною 4, або яка відповідає більшій степені числа 2 (наприклад, 2^{12}, якщо блоки і сторінки мають довжину, рівну 4 096=2^{12}). Отже, вимогу розміщення полів даних за строго визначеними адресами пам'яті, кратними 4, 8 і так далі, можна трактувати як умову зміщення цих полів від початку блока на відповідну величину.

Для спрощення припустимо, що всі поля даних необхідно розташувати за адресами ОП, кратними 4. Щоб виконати цю вимогу, достатньо дотримуватись таких умов:

а) кожен запис зміщений від початку блока на величину, кратну 4;
б) значення зміщення кожного поля від початку запису також кратне 4.

Іншими словами, ми заокруглюємо довжину всіх полів і записів до найближчого значення, кратного 4.

Приклад 29. Припустимо, що кортежі відношення *MovieStar* повинні представлятися в пам'яті за вказаною вище вимогою, тобто кожне поле запису необхідно розпочинати з байта, номер якого є множником 4. Тоді значення зміщення полів від початку запису будуть такими: 0, 32, 288, і 292. Загальна довжина запису становитиме 304 байти. Формат запису проілюстровано на рис. 22.

Довжина першого поля, *name*, становить 30 байтів, однак друге поле *address*, не повинно розпочинатися безпосередньо після першого – його необхідно змістити до найближчої позначки, кратної 4, а це – значення 32. Розмір поля *address* дорівнює 256, отож зміщення третього поля, *gender*, становитиме 288. Поле *gender* потребує лише одного байта, проте відвести для його зберігання менше, ніж 4 байти, не можна, отож зміщення наступного поля, *birthdate*, становитиме 292. Поле *birthdate* має довжину 10 байтів і завершується елементом під номером 301, отож загальна довжина запису повинна становити 302 байти (знову нагадаємо, що перший байт запису має номер 0), однак оскільки необхідно, щоб значення довжини запису також було кратне 4, то запис доповнюють двома «зайвими» байтами. Доцільно долучити ці два байти до поля *birthdate*, щоб їх неможливо було випадково використати для інших цілей.

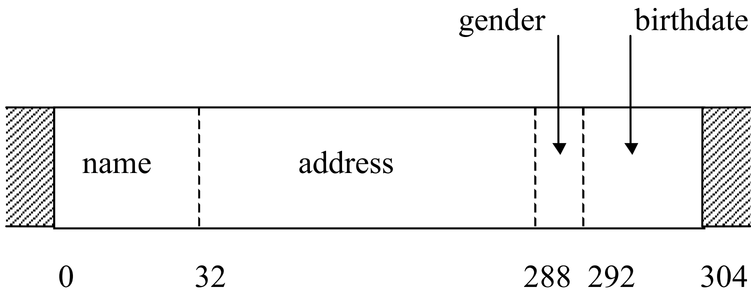


Рис. 22. Розташування полів кортежів MovieStar зі зміщенням, кратним 4

9.2. Заголовки записів

Існує ще одна проблема, яка трапляється при проектуванні структури запису. Часто у записі необхідно зберігати порції додаткової інформації, яка не належить до жодного з полів, наприклад:

1) дані про схему запису або покажчик на те місце, де СКБД зберігає схему запису цього типу;
2) відомості про загальну довжину запису;

3) дані про час останнього звертання до запису з метою його читання або модифікації.

Отож у багатьох випадках до структури запису долучають *заголовок (record header)*, який складається, зазвичай, з невеликої кількості байтів з додатковими даними того чи іншого виду.

СКБД, нагадаємо, зберігає і підтримує в актуальному стані інформацію схеми відношення, яка насправді відображає зміст відповідної команди *CREATE TABLE*:

1) перелік назв атрибутів;
2) список типів атрибутів;
3) порядок розташування компонентів атрибутів у кортежі;
4) обмеження, які стосуються окремих атрибутів і/або відношення загалом (наприклад, відомості про первинний ключ або обмеження приналежності значень деякому припустимому діапазону чи множині).

Очевидно, що розташовувати подібну інформацію у кожному записі кортежу відношення не варто. Достатньо долучити до запису покажчик на місце, де система зможе легко знайти все, що їй необхідно. З іншого боку, хоча довжину кортежу легко отримати зі схеми відношення, іноді зручніше зберігати це значення безпосередньо у записі з метою уникнення звертання до схеми (і виконання додаткових операцій введення/виведення), а також з метою швидкого пошуку початку наступного запису.

Приклад 30. Доповнимо структуру запису, розглянутого у прикладі 29, заголовком довжиною 12 байтів. Перші чотири байти заголовка призначено для зберігання покажчика, який задає величину зміщення в області файла бази даних, де представлено інформацію щодо схеми відношення. Наступні чотири байти представляють цілочислове значення довжини запису, а решта чотири байти – цілочислову величину, яка визначає момент вставки чи оновлення кортежу. Тепер довжина запису становить 316 байтів. Структуру запису представлено на рис. 23.

9.3. Групування записів постійної довжини в блоки

Записи, які представляють кортежі відношення, зберігаються у дискових блоках і переміщаються в ОП (у складі відповідних

блоків) за необхідності їхнього читання чи оновлення. Структуру блока, який містить записи відношення, наведено на рис. 24.

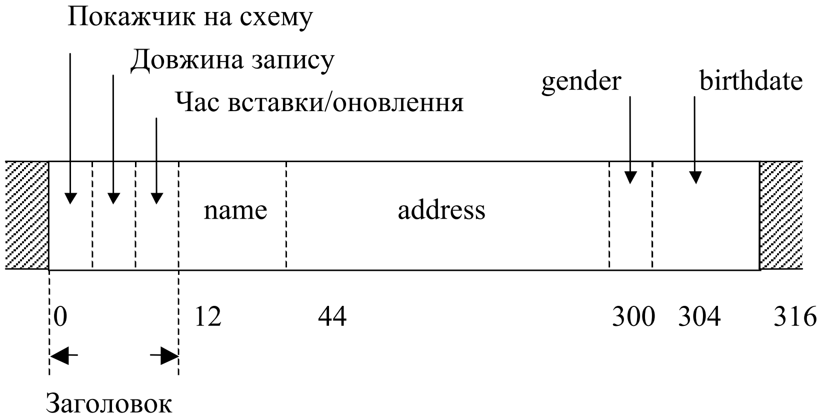


Рис. 23. Запис типу *MovieStar* із заголовком

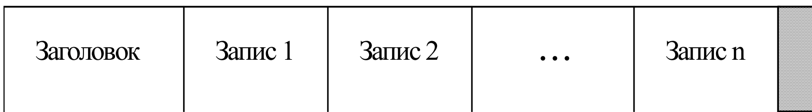


Рис. 24. Структура типового блока записів

Блок може налічувати необов’язковий заголовок, який містить таку інформацію:

1) посилання на один чи декілька блоків, які є частиною мережі блоків; цю частину використовують для створення структур індексів, що полегшують доступ до кортежів відношення (за детальною інформацією звертайтесь до розділу 3);
2) відомості про функцію, яку виконує блок в складі подібної мережі блоків;
3) дані про те, кортежі якого відношення представлено записами біжучого блока;
4) таблицю зі значеннями зміщень кожного запису від початку блока;

5) «ідентифікатор» блока (детальніше – у темі 10);
6) значення часу останнього звертання до блока і/або його модифікації.

У найпростішому випадку блок містить записи, які відповідають кортежам одного відношення, і формат тих записів фіксований. Отож одразу за заголовком блок налічує таку кількість записів, яку вдається помістити. Решту простору блока не використовують.

Приклад 31. Нехай необхідно упакувати в блоки розміром 4 096 байтів вміст відношення *MovieStar*. Кожен кортеж відношення представлено записом, структуру якого проілюстровано на рис. 23. Довжина запису становить 316 байтів. Під заголовок блока відведено 12 байтів, 4 084 байтів передбачено для розміщення 12-ти записів, і 292 байти залишаються вільними.

Вправи для опрацювання

Вправа 23. Припустимо, що запис складається з полів, перелічених у потрібному порядку: рядок із 15-ти символів, ціле розміром 2 байти, рядок дати формату SQL (10 байтів) і рядок часу формату SQL (8 байтів). Яка довжина запису, якщо:

1) зміщення полів довільні;
2) значення зміщення кожного поля кратні 4;
3) значення зміщення кожного поля кратні 8.

Вправа 24. Виконайте завдання з вправи 23 за умови, що список полів такий: дійсне число розміром 8 байтів, рядок із 17-ти символів, окремий байт і дата у форматі SQL.

Вправа 25. Виконайте завдання з вправи 23, вважаючи, що список полів такий самий, однак кожен запис має заголовок, який складається з двох 4-байтових покажчиків і символу.

Вправа 26. Виконайте завдання з вправи 24 з урахуванням того, що кожен запис має заголовок, який налічує 8-байтовий покажчик і десять 2-байтових цілочислових значень.

Вправа 27. Припустимо, що структура запису відповідає умові з вправи 23 і необхідно упакувати в блоки розміром 4 096 байтів

максимально можливу кількість записів, враховуючи, що заголовок блока містить десять 4-байтових цілих чисел. Скільки записів вдасться помістити в блок у кожній з трьох ситуацій, що стосуються вирівнювання полів щодо початку запису?

Вправа 28. Виконайте завдання з вправи 27 стосовно структури запису, зазначеної у вправі 26, та з урахуванням того, що розмір блока дорівнює 16 384 байти і блок містить заголовок, який налічує три 4-байтових цілих числа і таблицю зміщення, що містить по 2 байти даних з розрахунку на кожен запис блока.

Тема 10. Представлення адрес записів і блоків

Ознайомимось зі способами відображення адрес, покажчиків і посилань на записи і блоки, оскільки вони доволі часто відіграють роль складових частин інших записів. Існують також інші причини, які зумовлюють необхідність розуміння методів опису адрес у вторинних сховищах даних. Згодом (у розділі 3), розповідаючи про ефективні структури даних представлення відношень і файлів, ми наведемо кілька важливих прикладів використання адрес записів і блоків.

Після завантаження блока в буфер ОП адреси блока та окремих записів усередині блока можна сприймати як відповідні адреси перших байтів блока і запису у віртуальній пам'яті. Проте, якщо блок ще не зчитаний і перебуває на диску вторинного пристрою зберігання, то він не є частиною простору адрес віртуальної пам'яті. У цьому випадку розташування блока в базі даних, якою керує СКБД, описується певною послідовністю байтів, що містить відомості про ідентифікатор дискового пристрою, номер циліндра і так далі, а запис усередині блока визначається зміщенням його першого байта щодо початку блока.

Цю тему ми розпочнемо з обговорення особливостей адресних просторів у тому розумінні, як їх представляють в архітектурі «клієнт/сервер», потім наведемо різні способи опису адрес та опишемо методи «підміни» покажчиків, тобто їхнє перетворення під час передачі інформації з сервера бази даних прикладним програмам-клієнтам.

10.1. Системи «клієнт/сервер»

Типова СКБД функціонує як процес *сервера*, який обслуговує запити на отримання даних із вторинних пристроїв зберігання, що надходять від процесів *клієнтів* – застосувань-користувачів інформації з бази даних. Процеси сервера і клієнтів можуть виконуватися як на одному комп'ютері, так і в рамках розподіленої мережі комп'ютерів.

Клієнтські застосування, зазвичай, використовують 32-розрядний віртуальний адресний простір, який містить понад 4 млрд різних адрес. ОС або СКБД ухвалює рішення про те, які частини адресного простору в той чи інший момент часу необхідно розмістити в ОП, а апаратне забезпечення несе відповідальність за встановлення взаємно однозначної відповідності між віртуальними адресами і фізичними адресами ОП. Не вдаватимемось у деталі перетворення віртуальних адрес у фізичні і сприйматимемо адресний простір клієнта так, наче його цілковито розташовано в ОП.

Інформація сервера «перебуває» в адресному просторі бази даних. Адреси в такому просторі вказують на блоки і, можливо, на зміщення всередині блоків. Існує декілька способів представлення подібних адрес.

1. *Фізичні адреси* – набори байтів, які дають змогу визначити розташування блока чи запису на носії вторинного пристрою зберігання. Фізична адреса налічує такі компоненти:
   а) адресу комп'ютера, до якого підключено пристрій зберігання (якщо база даних розподілена між декількома комп'ютерами);
   б) ідентифікатор (дискового) пристрою зберігання, на якому розміщений шуканий блок даних;
   в) номер циліндра диска;
   г) номер доріжки циліндра (якщо пристрій містить пластини з декількома поверхнями);
   д) номер блока в межах доріжки;
   е) зміщення першого байта запису від початку блока (якщо адресується запис).

2. *Логічні адреси.* Кожен блок або запис мають логічну адресу – довільну послідовність байтів деякої фіксованої довжини. Логічні адреси транслюються у фізичні за допомогою *таблиці відповідності* (*map table*), яка зберігається у певному місці диска. Таблицю відповідності схематично зображено на рис. 25.

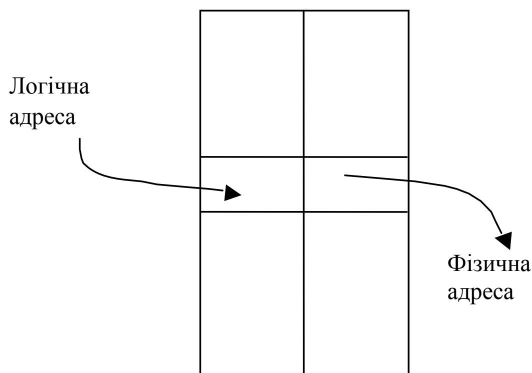


Рис. 25. Таблиця відповідності

Зауважимо, що фізичні адреси відзначаються великими розмірами. Вісім байтів – це той мінімум, за якого в адресу вдається включити хоча б деякі з перелічених вище компонентів. У деяких системах під кожну фізичну адресу відведено 16 байтів. Уявимо, для прикладу, базу даних об'єктів, спроектовану так, щоб забезпечити можливість використання протягом 100 років. Припустимо, що в майбутньому система доросте до таких розмірів, що її доведеться розподіляти на мільйон комп'ютерів, кожен із яких володіє розширеним адресним простором і настільки продуктивний, що здатен створювати по одному об'єкту протягом кожної наносекунди. Така система зможе зберігати до 2^{77} об'єктів, і для представлення кожної з адрес необхідно буде не менше, ніж десять байтів. Оскільки в кожній адресі доведеться подати посилання на комп'ютер, ідентифікатор пристрою зберігання і, ймовірно, ще чимало компонентів, то розмір адреси значно перевищить вихідну 10-байтову межу.

10.2. Логічні і структуровані адреси

Закономірним є питання, яке призначення логічних адрес. Всю необхідну інформацію про фізичні адреси можна взяти із таблиці відповідностей. Щоб простежити за логічними посиланнями, які вказують на записи, достатньо звернутися до таблиці відповідностей і знайти необхідні фізичні адреси. Окрім того, рівень опосередкованості, який закладено у таблиці відповідностей, забезпечує значні переваги в гнучкості. Наприклад, під час використання різних форм організації даних необхідно здійснювати переміщення записів – як всередині блока, так і з одного блока в інший. За використання таблиці відповідностей зовнішні покажчики посилаються тільки на її комірки. І все, що необхідно зробити для переміщення чи вилучення запису, – це змінити певний елемент таблиці відповідностей.

Застосовують і схеми представлення *структурованих адрес* (*structured address*), які пропонують ті чи інші можливості суміщення фізичних і логічних адрес. Наприклад, цілком правдоподібною є ситуація, коли для посилання на блок (але не на зміщення всередині блока) використовують фізичну адресу, яка супроводжується ключовим значенням шуканого запису. Тоді блок відшукують за «фізичною» частиною адреси, а потім для отримання необхідного запису аналізується задане ключове значення.

Очевидно, для перегляду записів усередині блока необхідно володіти достатньою інформацією, яка дає змогу визначити розташування записів і розрізняти їх. У найпростішому випадку записи мають постійну довжину і володіють ключовими полями, розташованими зі строго визначеним зміщенням. Тоді залишається звернутися до елемента заголовка блока, який містить значення лічильника записів, що належать блокові, – і ми чітко визначимо, де шукати ключові поля, вміст одного з яких задано як частину адреси. Проте блоки можна формувати і багатьма іншими способами (детальнішу інформацію подамо нижче), які даватимуть змогу виокремлювати записи всередині блока.

Подібний і цілком зручний варіант поєднання фізичних і логічних адрес передбачає зберігання у кожному блоці спеціальної *таблиці зміщень* (*offset table*), яка містить значення зміщення від

початку блока для кожного запису, що належить блокові. Схему блока з таблицею зміщень подано на рис. 26. Зверніть увагу, що таблиця зміщень нарощується від початку блока, а записи заносимо в блок у зворотному напрямі. Такий підхід найзручніший, якщо довжина записів не однакова.

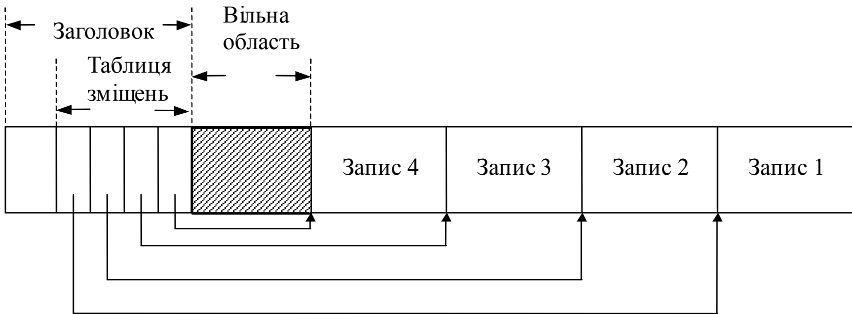


Рис. 26. Схема блока з таблицею зміщення записів

У цьому випадку адреса запису визначається фізичною адресою блока і значенням відповідного елемента таблиці зміщень. Подібний рівень опосередкованості значень адрес є зручніший порівняно з моделлю логічних адрес і усуває необхідність використання глобальної таблиці відповідностей.

- При зміні положення запису всередині блока достатньо модифікувати вміст елемента таблиці зміщень; значення зовнішніх покажчиків на запис виправлення не потребують.
- Якщо розмірність елементів таблиці зміщень достатня для зберігання «довгих» адрес, то запис можна перемістити навіть в інший блок.
- При видаленні запису існує можливість позначити відповідний елемент таблиці зміщень спеціальною позначкою – *«обеліском»* (*tombstone*), який засвідчуватиме факт вилучення запису. Якщо запис вилучено, то ідучи за значеннями покажчиків, система відшукає позначку-«обеліск» і буде змушена присвоїти покажчикам нульові значення або

змінити структури даних так, щоб вилучити посилання на неіснуючий запис.

10.3. Підміна покажчиків

Часто структура записів налічує покажчики або адреси. Подібну ситуацію не можна назвати типовою для записів, які представляють кортежі відношень, однак вона достатньо поширена, якщо йдеться про об’єкти. Окрім того, сучасні об’єктно-реляційні системи баз даних дають змогу використовувати атрибути типу покажчика (які називають посиланнями), так що необхідність опису покажчиків у контексті записів виникає навіть у реляційному середовищі. Зрештою, на основі блоків, взаємозв’язаних між собою за допомогою покажчиків, формуються структури індексів (детальніше про це у розділі 3). Отож обговоримо проблему керування покажчиками при переміщенні блоків із вторинних сховищ даних в ОП, і навпаки.

Як ми згадували раніше, кожен блок, запис, об’єкт чи інший адресований елемент даних володіє адресами двох типів.

1. *Адреса в базі даних* – адреса у просторі адрес сервера бази даних, яка містить, зазвичай, близько восьми байтів і задає місцерозташування елемента даних у вторинному сховищі.
2. *Адреса пам’яті* – адреса у віртуальному адресному просторі після буферизації елемента у віртуальній пам’яті, яка налічує, зазвичай, чотири байти.

Якщо елемент розташовано у базі даних, посилатися на нього необхідно, використовуючи адресу в базі даних. Якщо ж елемент зчитано в ОП, то він допускає можливість звертання до нього як за адресою в базі даних, так і за адресою пам’яті. Застосовуючи для посилання на елемент, який є у віртуальній пам’яті, покажчик, зручніше використовувати адресу пам’яті, оскільки переміщатися за значенням такого покажчика вдається за допомогою єдиної машинної інструкції.

Використання адрес у базі даних, навпаки, вимагає значних обчислювальних витрат, які полягають у використанні таблиці трансляції. За допомогою цієї таблиці «базові» адреси всіх елементів, які перебувають у цей час у віртуальній пам’яті, транслюються в адреси пам’яті. Таблиця трансляції, яку схематично зображено на

рис. 27, нагадує таблицю відповідностей, що виконує перетворення логічних адрес у фізичні. Зауважимо, однак, що:

а) і логічні, і фізичні адреси є формами представлення адрес у базі даних; з іншого боку, адреси пам'яті, перелічені в таблиці трансляції, вказують на копії об'єктів у віртуальній пам'яті;

б) таблиця відповідностей містить рядки для всіх адресованих елементів бази даних, а в таблиці трансляції згадуються адреси тільки тих елементів, які завантажено у віртуальну пам'ять.

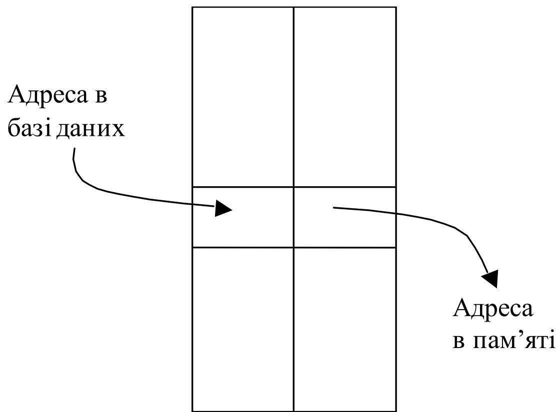


Рис. 27. Таблиця трансляції для перетворення адрес у базі даних в адреси пам'яті

З метою зменшення накладних витрат, які накопичуються унаслідок неперервного повторення процедур трансляції адрес у базі даних в адреси пам'яті, розроблено кілька прийомів, які в сукупності прийнято називати *підміною покажчиків* (*pointer swizzling*). Ідея полягає в тому, що під час переміщення блока даних із вторинного сховища в ОП покажчики всередині блока «підміняються», тобто їхній вміст перетворюється з простору адрес бази даних у віртуальний адресний простір. Покажчик у цьому випадку повинен містити такі компоненти:

1) біт, який засвідчує, що покажчик містить адресу в базі даних або (підмінену) адресу пам’яті;
2) адресу в базі даних або адресу пам’яті – те, що необхідно в конкретному випадку; для зберігання адреси відводиться однакове місце незалежно від її типу (очевидно, що, використовуючи адресу пам’яті, яка, зазвичай, коротша від адреси в базі даних, частина простору залишається вільною).

Приклад 32. На рис. 28 проілюстровано просту ситуацію: блок 1 містить запис з покажчиками, перший з яких посилається на інший запис того ж блока, а другий – на запис у блоці 2. Під час копіювання блока 1 в ОП відбувається підміна покажчика, який адресує запис усередині цього блока, після чого він міститиме посилання на адресу пам’яті, за якою розташований шуканий запис.

Блок 2 залишається на диску, отож підмінити другий покажчик блока 1 не вдається. Він, як і раніше, повинен посилатися на адресу запису в базі даних. Якщо припустити, що згодом блок 2 також буде зчитано в ОП, то у цьому випадку, ймовірно, система отримає можливість підмінити і другий покажчик у записі блока 1 – все залежить від стратегії підміни, яку застосовують.

Існує декілька стратегій використання механізму підміни покажчиків.

Автоматична «підміна». Як тільки блок зчитується в ОП, в його записах відшукуються всі покажчики та адреси і вводяться в таблицю трансляції (якщо це не зроблено раніше). До них належать і покажчики, що посилаються з записів блока назовні, й адреси блока як такого і/або записів усередині блока. Необхідним є відповідний механізм, який дає змогу відшукати подібні покажчики у блоці.

1. Якщо блок містить записи з відомими схемами, то останні допоможуть відшукати у структурі записів будь-які елементи (зокрема, і покажчики).
2. Якщо блок використовується в структурі індексу (про цю ситуацію докладніше у розділі 3), то місце розташування покажчиків відоме наперед.

3. Заголовок блока може містити спеціальний список, який задає значення адрес розташування покажчиків усередині блока.

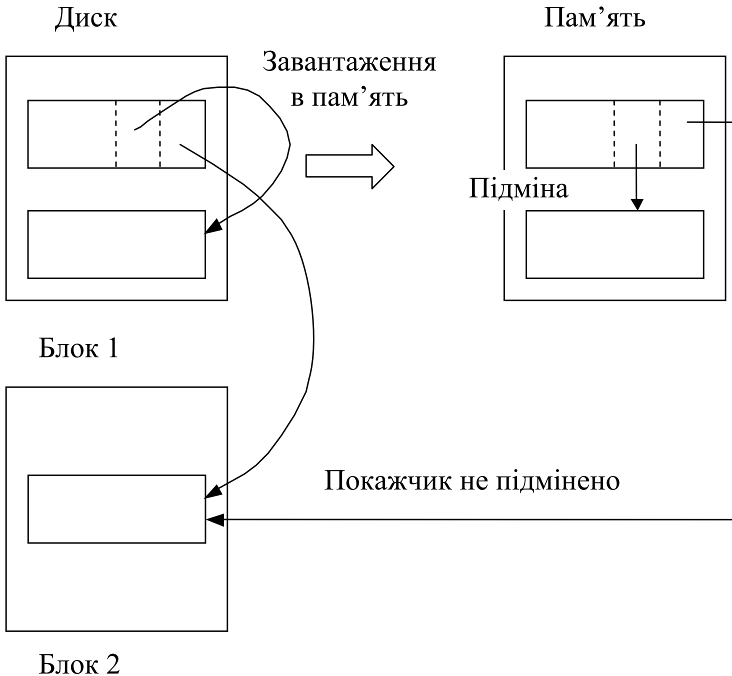


Рис. 28. Приклад використання механізму підміни покажчиків

При вставці у таблицю трансляції адрес блоків і/або його записів, щойно скопійованих у пам’ять, положення блока в буфері пам’яті вже відоме. Отож долучити до таблиці трансляції адреси пам’яті для завантажених елементів неважко. Можлива ситуація, коли під час спроби внесення в таблицю трансляції адреси *A* бази даних виявиться, що адреса вже присутня в таблиці, оскільки блок з адресою *A* завантажився раніше. У цьому випадку достатньо замінити вміст покажчика на *A* в блоці, який щойно зчитано, відповідною адресою пам’яті і присвоїти бітові «підміни» значення

«істина». Якщо ж адресу *A* ще не внесено в таблицю трансляції, то блок з цією адресою ще не зчитувався у пам'ять. Отож покажчик на *A* підмінити не можна і його адреса, як і раніше, належатиме до категорії адрес у базі даних.

Якщо у спробі слідування за покажчиком *P* з деякого блока виявиться, що цей покажчик ще не підмінено, тобто він посилається на адресу в базі даних, то необхідно перевірити, чи завантажено у пам'ять блок *B*, що містить елемент з адресою *P*. Якщо в таблиці трансляції адреси *P* немає, то необхідно скопіювати блок *B* у буфер ОП і підмінити покажчик *P*, змінюючи адресу в базі даних на рівнозначну адресу пам'яті.

«Підміна» за вимогою. Альтернативний підхід до використання механізму «підміни» покажчиків полягає в тому, що після початкового завантаження блока в ОП вміст покажчиків залишається без змін. У таблицю трансляції заносять елементи, які представляють адреси в базі даних і в пам'яті: блока, записів у блоці і покажчиків у записах. Якщо необхідно переміститися за адресою пам'яті, заданою деяким покажчиком *P*, то цей покажчик перетворюється так, як і у випадку використання автоматичної підміни.

Різниця між стратегіями автоматичної трансляції і підміни за вимогою полягає в тому, що перша виконує перетворення вмісту всіх покажчиків при завантаженні блока в ОП. Можлива економія, яка отримується за одночасної трансляції всіх покажчиків у блоці, ймовірно, зрівноважується тим, що не всі підмінені покажчики використовуватимуть у майбутньому. Іншими словами, йдеться про зайві затрати часу на перетворення, результати яких не використовують.

Заборона «підміни». Механізм підміни покажчиків можна не використовувати зовсім. У цій ситуації таблиця трансляції все ще необхідна, і переміститися за покажчиками можна в їхній «звичайній» (не «підміненій») формі. Подібний підхід має ту перевагу, що дає змогу уникнути записів, «закріплених» у пам'яті (детальніше про це – у пункті 10.5), і необхідності прийняття рішення щодо форми представлення покажчиків.

Програмне керування механізмом «підміни» покажчиків. У деяких випадках програміст, який створює застосування, може наперед знати про те, коли і які покажчики у блоці будуть потрібними, і потурбуватися про їхню примусову «підміну» в процесі завантаження блока в ОП або за спеціальною вимогою. Наприклад, якщо наперед відомо, що інформацію блока, який завантажується (припустимо, кореневого блока B-дерева), активно використовуватимуть, то покажчики, напевно, зручніше «підмінити» одразу. Проте покажчики у блоках, які після зчитування використовуються одноразово, ймовірно, є сенс не змінювати і транслювати виключно за необхідності.

10.4. Збереження блоків на диску

Якщо блок даних у пам'яті необхідно зберегти на диску, то вміст усіх покажчиків усередині блока треба перетворити в «зворотному» напрямі, тобто адреси пам'яті замінити відповідними адресами в базі даних. Таблицю трансляції, яка пов'язує адреси обох типів, можна використовувати і для відшукання адрес у базі даних за заданими адресами пам'яті. Проте було б недоцільним здійснювати перегляд таблиці трансляції цілковито під час необхідності виконання кожної операції, оберненої до «підміни» покажчиків. Отож таблиця трансляції має відповідні індекси, і задачу відшукання адреси пам'яті за заданою адресою *x* у базі даних можна звести до виконання запиту:

```
SELECT numAddr
FROM TranslationTable
WHERE dbAddr = x;
```

У цьому випадку доцільним рішенням було б створення індексу для атрибуту *dbAddr* у формі геш-таблиці (детальніше розглянемо у розділі 3), у якій функцію ключів виконують адреси в базі даних.

10.5. «Закріплені» записи і блоки

Вважають, що блок «закріплено» (*pinned*) у пам'яті, якщо його не можна безпечно зберегти на диску, не виконуючи додаткових дій. Заголовок блока може мати спеціальний біт, який демонструє, чи «закріплений» блок, чи ні. Обставини, за яких передбачають

«закріплення» блока в пам'яті, є різноманітними. Наприклад, це необхідність підтримки вимог, які виставляє підсистема відновлення даних. Іншою немаловажною підставою для «закріплення» деякого блока є наявність посилань на елементи цього блока з боку підмінених покажчиків інших блоків.

Якщо у блоці B_1 «підмінено» покажчик, який адресує деякий елемент даних блока B_2, то переміщення на диск блока B_2 і очищення буфера ОП треба виконувати зважено. Інакше наслідки можуть виявитися дуже трагічними. Якщо не вжити захисних заходів, то покажчик у B_1 (якщо раптом доведеться ним скористатися) приведе систему до буфера, в якому блок B_2 вже відсутній – наслідки можуть виявитися найтрагічнішими. Отож блок (подібний до B_2), що адресується ззовні, набуває статусу «закріпленого».

Під час спроби збереження блока пам'яті на диску необхідно не тільки транслювати зворотно «підмінені» покажчики, але й з'ясувати, чи блок не є «закріпленим». Якщо блок «закріплено», то необхідно або усунути причини, за яких блок потрапив у цей стан, або залишити блок на місці (очевидно, погоджуючись з тим, що буфер пам'яті, відведений під блок, не можна буде використовувати для іншої мети). Щоб позбавити блок від ознаки «закріплення», встановленої у зв'язку з наявністю зовнішніх «підмінених» покажчиків, що посилаються на елементи блока, необхідно здійснити обернене перетворення цих покажчиків. Отож у таблиці трансляції для кожної адреси в базі даних, якій відповідає певний елемент у пам'яті, треба зберігати й адреси пам'яті, в яких розташовано «підмінені» покажчики, що адресують цей елемент. З цією метою застосовують два підходи, за яких необхідно:

1. Підтримувати перелік посилань на адреси пам'яті у вигляді зв'язаного списку, який поставлений у відповідність тому або іншому елементові трансляції.
2. Якщо адреси пам'яті суттєво коротші від адрес у базі даних, то створити зв'язаний список у просторі, який використовують для розміщення покажчика. Тобто розмістити в області покажчика на адресу в базі даних:
   а) «підмінений» покажчик;

б) інший покажчик, як частину зв'язаного списку покажчиків, що посилаються на одну адресу пам'яті.

На рис. 29 проілюстровано спосіб зв'язування у список усіх покажчиків, які посилаються на адресу в пам'яті, починаючи з елемента таблиці трансляції, який ставить у відповідність адресі *x* у базі даних адресу *y*.

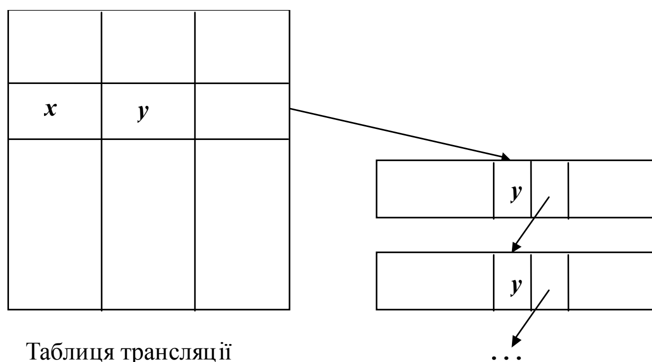


Рис. 29. Представлення зв'язаного списку покажчиків, що адресують одну адресу пам'яті

Вправи для опрацювання

Вправа 29. Розглянемо дисковий привід «Soaring 747», який має такі характеристики: 8 пластин або 16 поверхонь; 2^{14} або 16 384 доріжок на поверхні; в середньому 2^{7}, або 128 секторів на доріжці; 2^{12} або 4 096 байтів у секторі. Якщо представити фізичні адреси структурних елементів цих носіїв, виокремлюючи окремі байти для опису кожного циліндра, доріжки циліндра і блока всередині циліндра, то скільки байтів для цього необхідно? Зверніть увагу на максимально можливу кількість блоків на доріжці і врахуйте, що кількість секторів з розрахунку на одну доріжку змінюється.

Вправа 30. Виконайте завдання з вправи 29 стосовно параметрів диска «Soaring 777». Характеристики: десять поверхонь, по

10 000 доріжок на кожній; кожна доріжка містить 1 000 секторів місткістю 512 байтів; зазори займають 20% площі доріжки.

Вправа 31. Якщо поряд з описом адрес блоків необхідно представити адреси записів усередині блоків, то доведеться залучити додаткові байти. Вважаючи, що, як і у вправі 29, розглядається дисковий пристрій «Soaring 747», дайте відповідь, скільки байтів необхідно для адрес записів, якщо:

а) до складу фізичної адреси долучити номер байта всередині блока;

б) для посилання на записи використовувати структуровані адреси, вважаючи, що кожному запису відповідає ключ – 4-байтове ціле число.

Вправа 32. Для представлення адреси комп'ютера, згідно з вимогами протоколу *IP,* необхідно 4 байти. Припустимо, що адреса блока дискових даних у системі, відкритій для доступу з Internet, налічує IP-адресу комп'ютера, номер дискового пристрою з інтервалу 1-1 000 і адресу блока на конкретному пристрої (йдеться про «Soaring 747»). Наскільки довгою виявиться повна адреса блока?

Вправа 33. Припустимо, що процес автоматичної «підміни» покажчиків займає половину часу, необхідного у випадку, коли б кожен покажчик «підмінявся» окремо і незалежно від інших. Якщо ймовірність того, що покажчик у пам'яті буде використано хоча б один раз, дорівнює p, то за яких значень p процес автоматичної «підміни» виявиться ефективнішим порівняно з механізмом «підміни» за вимогою?

Тема 11. Елементи даних і записи змінної довжини

До цього часу ми користувалися спрощеними припущеннями про те, що кожен елемент даних має постійну довжину, всі записи описуються фіксованою схемою і кожен запис являє собою список полів постійного розміру. Проте на практиці не все так ідеально. Цілком правдоподібними є ситуації, коли доводиться мати справу з такими об'єктами та умовами:

1. *Елементи даних змінного розміру*. Наприклад, на рис. 20 наведено оголошення відношення *MovieStar* з атрибутом-полем *address*, довжина якого може досягати 255 байтів. Деякі рядки адрес дійсно можуть бути такими довгими, проте, здебільшого, їхній розмір становить, найімовірніше, 50 байтів або менше. Якщо відводити під кожну адресу стільки байтів, скільки справді необхідно, то нам вдасться більше, ніж удвічі, скоротити простір, необхідний для збереження вмісту відношення.
2. *Поле, яке повторюється*. За необхідності представлення в рамках запису, який відповідає деякому об'єктові, зв'язку типу «багато до багатьох» доцільно зберігати у записі таку кількість посилань, яка відповідає кількості об'єктів, що адресуються біжучим об'єктом.
3. *Записи змінного формату*. У певних ситуаціях попередня інформація щодо кількості екземплярів полів у записі або про номенклатуру полів наперед відсутня. Наприклад, деякі кіноактори водночас є і режисерами. Отож можливо, що доведеться доповнити записи з відомостями про акторів посиланнями на зняті ними кінофільми. Оскільки, здебільшого, актори все-таки не є ні режисерами, ні продюсерами/керівниками, то резервувати відповідне місце для подібної інформації у кожному записі про актора не доцільно.
4. *Поля типу «BLOB» («крупні двійкові об'єкти»).* Сучасні СКБД підтримують атрибути, значеннями яких можуть слугувати крупні порції даних. Наприклад, у деяких випадках доцільним виявиться включення до схеми відношення *MovieStar* атрибута *picture*, призначеного для збереження фотографії актора у форматі *GIF*. Запис з відомостями про кінофільм цілком здатний містити поле з даними об'ємом у декілька гігабайтів, яке представляє версію кінофільму в форматі *MPEG*. Подібні поля є настільки великими, що модель, яка передбачає можливість розміщення декількох записів у межах блока, підлягає кардинальному перегляду.

11.1.Записи з полями змінної довжини

Якщо одне або декілька полів запису мають змінну довжину, то запис повинен містити інформацію, необхідну і достатню для відшукання будь-якого з полів. Однією з простих та ефективних вважають схему, яка передбачає розміщення всіх полів постійного розміру на початку запису, а всіх решта полів – наприкінці. Заголовок запису повинен містити:

1) значення довжини запису;
2) адреси (зміщення) перших байтів усіх полів змінного розміру; якщо такі поля завжди розташовані у межах запису в одному і тому ж порядку, то перше з них не потребує покажчика – зрозуміло, що воно йде безпосередньо за групою полів постійної довжини.

Приклад 33. Розглянемо запис, який представляє відомості про ім'я (*name*), адресу (*address*), стать (*gender*) і дату народження (*birthdate*) актора. Вважатимемо, що *gender* і *birthdate* є полями постійної довжини – 1 і 12 байтів, відповідно. Для атрибутів *name* і *address*, однак, доцільно відводити поля такої довжини, яка необхідна у кожному конкретному випадку. На рис. 30 подано схему, яка ілюструє запис зазначеного формату. Якщо поле імені завжди розташовувати перед полем адреси, то можна обійтись без покажчика на перший байт імені, оскільки, згідно з прийнятою нами угодою, поля змінної довжини знаходяться безпосередньо після полів постійної частини запису.

✓ **Корисно знати.** *Представлення значення NULL.* Кортежі відношень часто налічують компоненти зі значеннями *NULL*. Ці значення зручно представляти у форматі запису, проілюстрованому на рис. 30. Якщо в полі, подібному до *address*, необхідно занести *NULL*, то достатньо розмістити у полі заголовка з покажчиком на *address* нульовий покажчик. Область під вміст *address* не буде потрібною і місце знадобиться тільки для зберігання покажчика у заголовку. Подібний підхід забезпечує суттєву економію в середньому – він має переваги навіть

```
у тих випадках, коли поле address є постійної
довжини, проте його вміст не визначено.
```

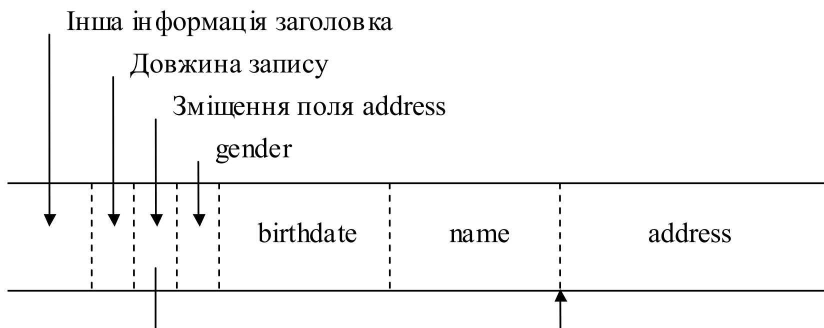


Рис. 30. Запис типу MovieStar з компонентами name і address, які реалізовані як поля змінної довжини

11.2. Записи з полями, які повторюються

Ситуація, подібна до розглянутої у пункті 11.1, повторюється і в тому випадку, коли запис містить змінну кількість екземплярів поля деякого типу *F*, однак саме це поле має фіксований розмір. Зручно об'єднати всі входження поля *F* в одну групу і долучити до заголовка запису покажчик, який адресує перший елемент групи. Для відшукання довільного екземпляра поля *F* достатньо виконати наступне. Позначимо кількість байтів, відведених для поля *F*, як L, а значення зміщення групи полів *F* – як *S*. Тоді зміщення екземпляра поля з номером k, де k=0, 1, 2, ..., дорівнює *S*+k**L*.

Приклад 34. Нехай необхідно змінити структуру запису типу *MovieStar* так, щоб зберегти поля *name* і *address* і забезпечити представлення інформації про кінофільми з участю актора. На рис. 31 наведено схему, яка ілюструє один зі способів компонування подібного запису.

Заголовок запису містить покажчики на перший байт поля *address* (ми вважаємо, що поле *name* розміщується одразу ж після заголовка, отож покажчика не потребує) і на перший із покажчиків,

що адресують записи типу *Movie*, – відоме значення довжини запису дає змогу визначити їхню кількість.

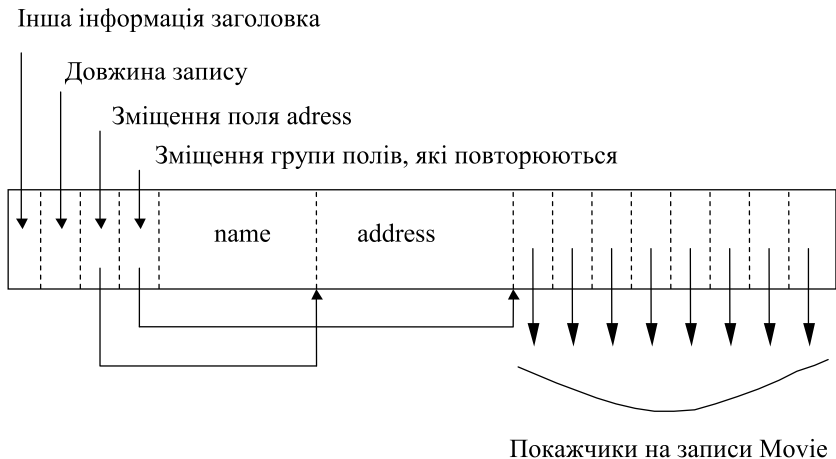


Рис. 31. Приклад запису з групою полів, які повторюються

Альтернативний підхід полягає у збереженні постійної частини запису в її попередній формі та переміщенні змінної порції даних – полів змінної довжини і/або групи полів, які повторюються, – в окремий блок. Вихідний запис повинен містити:

1) адреси кожного з «винесених» полів;
2) кількість подібних полів або, коли поля різні, значення їхніх довжин.

На рис. 32 проілюстровано інший варіант компонування запису, який розглянуто у прикладі 34: поля змінної довжини *name* і *address*, а також група полів покажчиків на записи *Movie* перенесено в окремий блок або декілька блоків.

Розглянутий прийом, який передбачає розподіл постійної і змінної частини запису у різних блоках, має і переваги, і недоліки:

- однакова довжина записів дає змогу пришвидшити процеси пошуку, зменшити розміри заголовків і спростити задачу переміщення записів усередині блоків і між різними блоками;

* зберігання змінних частин в окремих блоках, однак, спричинює збільшення кількості операцій введення/виведення, необхідних для перегляду вмісту кожного запису цілковито.

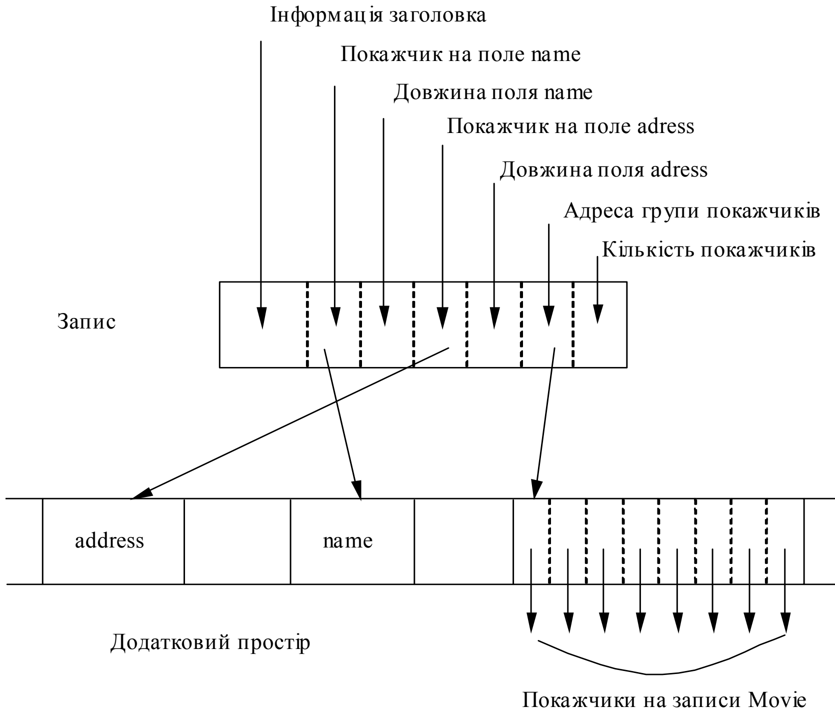


Рис. 32. Схема розподілу постійної і змінної частин у різних блоках

Компромісне рішення полягає в тому, щоб виокремити в рамках порції запису постійної довжини таку область, вмісту якої достатньо для представлення:

1) деякої «розумної» кількості копій полів, які повторюються;
2) покажчика, який задає адресу місця розміщення додаткових екземплярів полів;

3) лічильника додаткових екземплярів полів.

Якщо кількість полів виявиться меншою, ніж було передбачено, то деяку частину області просто не буде використано за призначенням. Якщо ж кількість додаткових полів зростатиме, то покажчику в головному записі присвоять відповідне ненульове значення, яке задаватиме адресу розміщення полів у зовнішньому блоці.

11.3. Записи змінного формату

Складнішою вважають ситуацію, яка стосується використання записів, структура яких не відповідає наперед визначеній постійній схемі: наприклад, розмірності полів і/або порядок їхнього розташування в межах запису не цілком визначаються оголошеннями відношення або класу, кортежі чи об'єкти яких представлено такими записами. У найпростішому варіанті запису змінного формату для опису поля передбачено використання послідовності *тегів* (*tagged fields*), кожен з яких призначено для зберігання:

1) відомостей про функціональне призначення поля, а саме:
   а) ім'я поля чи атрибута;
   б) найменування типу поля, якщо ці дані не можна отримати за іншими елементами схеми;
   в) значення довжини поля, якщо це значення безпосередньо не визначено інформацією про тип поля;
2) значення поля.

Зазначимо, принаймні, дві причини, які зумовлюють доцільність використання полів тегів.

1. *Розповсюдженість застосувань, які інтегрують інформацію.* Часто відношення конструюються з традиційних джерел інформації різноманітних типів. Наприклад, відомості про кінофільми та акторів, які надійшли у централізовану базу даних, можна отримати від різних постачальників інформації, одні з яких передбачають необхідність зберігання даних про дати народження акторів, другі цю вимогу ігнорують, треті надають відомості про адреси в довільному форматі, а четверті підтримують чітко оговорену структуру і т.д. Якщо полів небагато, ймовірно, зручніше заповнювати значеннями *NULL* ті з них, вміст яких не визначено. Якщо

ж кількість і номенклатура типів полів надзвичайно великі, то значення *NULL* доведеться використовувати надто часто. Доцільніше зекономити простір і скористатися структурою полів тегів, які представляють тільки такі значення, які відмінні від *NULL*.

2. *Використання записів вільного формату*. Якщо запис містить чималу кількість полів і багато з них часто не використовуються або перелічуються в довільному порядку, то структури полів тегів мають значну перевагу. Наприклад, в особових картках пацієнтів медичних закладів може міститися інформація про різні тести та аналізи, кількість різновидностей таких тестів налічується тисячами, проте тільки деякі з них використовують при обстеженні кожного конкретного пацієнта.

Приклад 35. Припустимо, що поряд зі звичайними відомостями про акторів є окремі порції інформації: в яких фільмах той чи інший актор був режисером; які імена колишніх дружин (чоловіків) акторів (актрис); яким бізнесом займаються актори (є вони власниками ресторанів або магазинів) і т.п. На рис. 33 зображено фрагмент гіпотетичного запису змінного формату з використанням тегів. Вважають, що імена полів і назви їхніх типів описуються однобайтовими кодами. На схемі проілюстровано два поля з відповідними тегами. Обидва поля належать до рядкового типу.

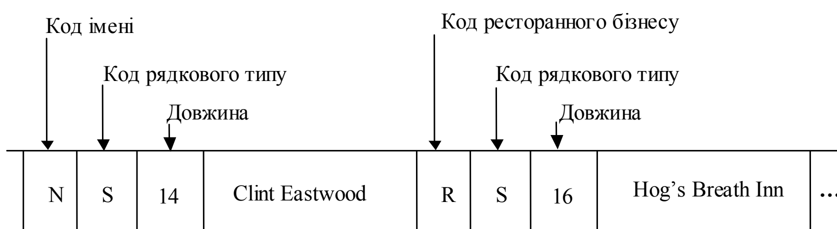


Рис. 33. Структура запису з полями тегів

11.4. Записи крупного об'єму

Розглянемо іншу проблему. Її значимість зростає відповідно до розповсюдження СКБД, що підтримують крупноформатні типи даних: ми розповімо про способи представлення значень, розмір яких заздалегідь перевищує об'єм блока. Типовими прикладами таких даних є фрагменти аудіо- чи відеозаписів. Часто елементи подібної інформації мають і змінну довжину. Якщо ж довжина фіксована, то все одно для представлення таких значень необхідні особливі прийоми. У цьому пункті ми зосередимо увагу на моделі *зв'язаних записів* – зручному механізмі керування записами, які неможливо помістити у блок. Питання представлення записів надвеликих розмірів, що вимірюються мегабайтами чи гігабайтами, розглянемо у пункті 11.5.

Модель зв'язаних записів має переваги і в тих випадках, коли розміри записів не перевищують об'єму блока, проте під час групування записів у блоки губляться чималі фрагменти простору пам'яті. У прикладі 31 в аналогічній ситуації, яка стосується розміщення записів у блоці, розмір області, що не використовується, становив близько 7% від загального об'єму блока. Якщо припустити, що довжина запису не надто перевищує половину розміру блока (у блок вдається помістити тільки один запис), то втрати будуть наближатися до позначки 50 %.

З метою подолання зазначених вище проблем бажано мати можливість розміщення записів у декількох блоках. Записи, які складаються з фрагментів, розташованих у декількох блоках, прийнято називати *зв'язаними* (*spanned records*).

Якщо записи передбачають зв'язування, то для опису кожного запису і його окремих фрагментів необхідно в їхніх заголовках подати додаткову інформацію:

1) заголовок кожного запису і фрагмент повинен зберігати окремий біт, який засвідчує, чи є порція даних фрагментом запису;
2) заголовок фрагмента необхідно доповнити бітами, які засвідчують, що фрагмент є першим (останнім) у ланцюгу зв'язаних фрагментів запису;

3) заголовок фрагмента запису повинен містити покажчики на попередній і/або наступний фрагменти.

Приклад 36. На рис. 34 проілюстровано схему розподілу трьох записів у двох блоках (розмір одного запису становить близько 60 % від об'єму блока). Заголовок фрагмента 2*a* запису 2 містить біт-індикатор фрагмента – біт, який зазначає, що фрагмент є першим, і покажчик на другий (2*b*) фрагмент запису 2. Аналогічно, заголовок фрагмента 2*b* містить біт-індикатор фрагмента, біт-ознаку заключного фрагмента і покажчик на перший (2*a*) фрагмент запису 2.

11.5. Об'єкти BLOB

Розглянемо способи представлення записів або полів надто великих розмірів. Прикладами подібних даних є графічні зображення різних фрагментів (*GIF* або *JPEG*), фрагменти відеозаписів (кодування *MPEG*), різноманітна аудіоінформація, сигнали радарів і т.п. Елементи такої інформації прийнято позначати абревіатурою *BLOB* (від *Binary Large Objects* – «*крупні двійкові об'єкти*»).

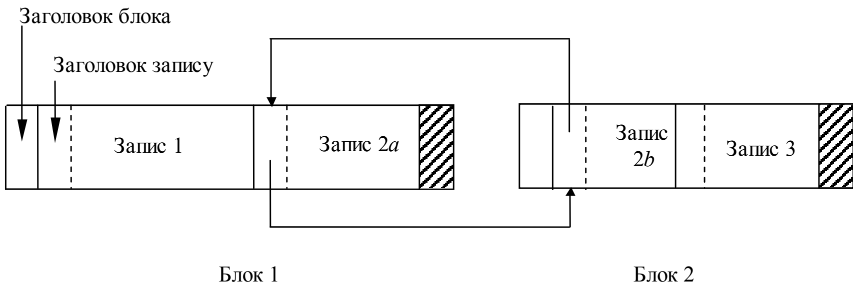


Рис. 34. Розподіл зв'язаних записів у блоках

Читання об'єктів BLOB. Природно вимагати, щоб блок даних, який очікує клієнт, система повертала повністю. Зі зростанням розмірів елементів даних такий підхід підлягає радикальному перегляду: можна вважати, що система повинна передавати водночас і цілковито тільки «невеликі» значення полів записів і

давати змогу клієнтові вимагати окремі блоки об'єкта BLOB по одному в кожний момент часу і незалежно від наявності блоків, що ще залишаються. Наприклад, якщо об'єкт BLOB представляє двогодинний кінофільм і клієнт посилає запит на його відтворення, то система повинна зчитувати і передавати клієнтові послідовні блоки об'єкта з достатньою швидкістю.

У багатьох випадках важливо, щоб клієнтові було надано можливість зчитування окремих порцій даних (наприклад, титрів кінофільму або фінальної частини аудіокліпа) незалежно від порядку їхнього розташування «усередині» об'єкта і без необхідності цілковитого завантаження об'єкта. Якщо система підтримує подібні операції, то їй, імовірно, необхідні структури індексних значень (наприклад, які вказують на початок кожного посекундного фрагмента фільму).

Вправи для опрацювання

Вправа 34. Припустимо, що картка пацієнта лікувального закладу налічує дату народження, номер полісу соціального страхування й ідентифікатор – під кожне поле фіксованого розміру відведено по 10 байтів. Запис містить також поля змінної довжини, призначені для зберігання рядків імені, адреси і відомостей з історії хвороби. Якщо кожен покажчик усередині запису і лічильник довжини займають по 4 байти, а вимоги стосовно вирівнювання полів не поставлено, то яким виявиться розмір запису без урахування області, необхідної для зберігання вмісту полів змінної довжини?

Вправа 35. Припустимо, що структура запису відповідає умовам вправи 34, проте розміри вмісту полів змінної довжини рівномірно розподілені в наступних інтервалах: ім'я – 10-50 байтів, адреса – 20-80 байтів, історія хвороби – 0-1 000 байтів. Визначити середню довжину запису з інформацією про пацієнта.

Вправа 36. Припустимо, що структура запису з відомостями про пацієнта лікувального закладу, розглянута у вправі 34, доповнена полями, що повторюються і відображають результати аналізів на наявність холестерину в крові. Кожне таке поле вимагає 16 байтів для представлення дати проведення і цілочислового результату

аналізу. Запропонуйте варіанти компонування картки пацієнта, якщо:

а) поля з результатами аналізів, які повторюються, містяться безпосередньо в самому записі;
б) результати зберігаються в окремих блоках, а запис налічує лише покажчики на них.

Вправа 37. Нехай у запис із структурою, запропонованою у вправі 34, необхідно додати поля з описом назви аналізів, дати їхнього проведення та отриманих результатів. Кожні три поля, які представляють дані про один аналіз, займають 40 байтів. Припустимо, що ймовірність проходження кожним пацієнтом будь-якого з тестів дорівнює p.

1. Нехай для зберігання покажчика і цілого значення потрібно по 4 байти, а всі дані аналізів зберігаються всередині запису як поля змінної довжини. Визначити середню кількість байтів, яку відведено для опису аналізів одного пацієнта.
2. Виконайте завдання п.1, якщо результати аналізів зберігаються поза записом, а запис містить відповідні покажчики.

Тема 12. Модифікація записів

Операції вставки, вилучення та оновлення записів даних створюють специфічні проблеми, які заслуговують особливої уваги. Проблеми набувають найбільш серйозного характеру, якщо змінюється довжина записів, хоча інколи виникають і тоді, коли записи і поля мають постійні розміри.

12.1. Вставка

Спочатку розглянемо операцію вставки нових кортежів у відношення (або, що майже те саме, об’єктів у біжучий екземпляр класу). Якщо записи, що представляють вміст відношення, зберігаються у довільному порядку, то достатньо просто знайти деякий блок з вільною областю, достатній для розміщення додаткових записів, або, якщо такого немає, запросити новий блок. Зазвичай, система баз даних підтримує деякий механізм ефективного пошуку всіх блоків, які містять кортежі певного відношення чи об’єкти зазначеного класу.

Проблеми збільшуються, якщо кортежі необхідно розташувати в деякому заданому порядку (такому, який забезпечує, наприклад, сортування відношення за значеннями компонентів первинного ключа). Існує чимало серйозних доказів на користь зберігання записів впорядкованими, оскільки в цьому випадку суттєво спрощується виконання запитів певних категорій. За необхідності вставки нового запису системі необхідно, передусім, визначити місцерозташування блока, в який доцільно помістити цей запис, сподіваючись, що блок містить потрібну вільну область. Оскільки щодо записів передбачений заданий порядок розташування, можливо, доведеться пересувати записи всередині блока, щоб звільнити простір достатнього об’єму.

Якщо необхідно здійснити зсув записів усередині блока, то структура, запропонована на рис. 26 і відтворена на рис. 35, буде дуже доречною. Нагадаємо, що заголовок блока може налічувати таблицю зміщень (рис. 35), яка містить покажчики на перші байти кожного запису всередині блока. Якщо покажчик посилається на запис за межами блока, то він повинен містити *структуровану адресу*, що складається з адреси іншого блока та адреси елемента заголовка цього блока, в якому зберігається значення зміщення необхідного запису.

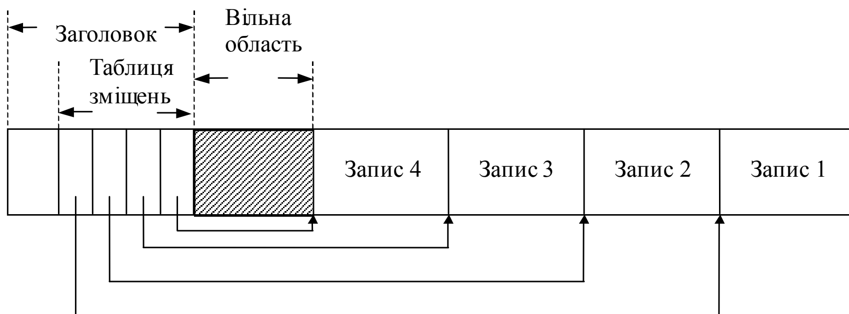


Рис. 35. Таблиця зміщень спрощує задачу пересування записів усередині блока

Якщо простору для нового запису в біжучому блоці достатньо, то записи пересуваються і покажчики в таблиці зміщень отримують оновлені адресні значення, після чого запис вставляється на

вивільнене місце блока і в таблицю зміщень додається новий покажчик, який посилається на адресу першого байта вставленого запису.

Якщо ж у блоці місця для нового запису немає, то підшукується позиція для його вставки за межами блока. Існують два головні підходи щодо розв'язання подібної задачі та кілька їхніх комбінованих варіантів.

1. *Знайти місце у найближчому блоці.* Наприклад, якщо, згідно з заданим порядком слідування записів, новий запис підлягає вставці у блок B_1, а достатнього вільного простору в ньому немає, то необхідно звернутися до наступного за порядком блока B_2. Якщо блок B_2 має вільний фрагмент потрібного розміру, то необхідно перемістити один або декілька записів з найвищими номерами з блока B_1 у B_2, пересунути записи в обох блоках і присвоїти покажчикам у таблиці зміщень B_1, які посилаються на записи, перенесені в блок B_2, «довгі» структуровані адреси відповідних елементів таблиці зміщень блока B_2. У цьому випадку, очевидно, розмір області таблиці зміщень блока B_1 зросте.
2. *Створити блок переповнення.* У заголовку кожного блока B необхідно передбачити місце для спеціального покажчика, що адресує додатковий *блок переповнення* (*overflow block*), призначений для зберігання записів, які повинні були б належати блокові B, однак у ньому не помістилися. Один блок переповнення, який зв'язаний з блоком B, здатний посилатися на наступний блок переповнення і т.д. Структуру проілюстровано на рис. 36.

12.2. Вилучення

Під час вилучення запису з'являється можливість повторного використання фрагмента простору блока, який звільняється. Якщо використовується структура блока, подана на рис. 35, і записи здатні пересуватися в межах блока, то системі вдасться легко оптимізувати розміщення записів, щоб очистити область усередині блока.

Якщо зсув записів виконувати не можна, то система повинна підтримувати деякий список фрагментів блока, що не

використовуються, щоб при вставлянні нового запису було зрозуміло, чи має блок достатньо вільного простору і, якщо відповідь позитивна, де саме розміщені доступні ділянки і наскільки вони великі. Список вільних областей не обов'язково зберігати в заголовку блока. Достатньо розмістити у ньому тільки головний елемент-покажчик зв'язаного списку, а решту елементів зберігати безпосередньо у вільних областях.

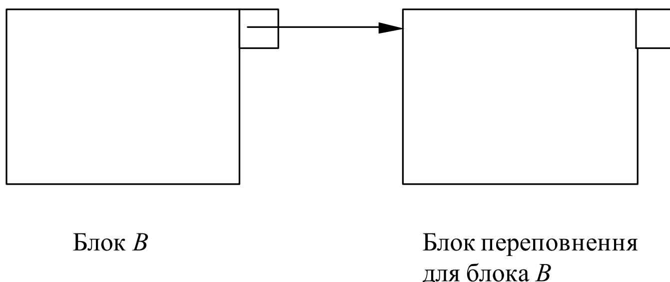


Рис. 36. Основний блок і відповідний блок переповнення

Вилучення запису інколи спричинює до того, що потреба у використанні блока переповнення зникає. Якщо запис видаляється з блока *B* або з довільного блока, що належить ланцюжку зв'язаних блоків переповнення, які належать до блока *B*, то система може підрахувати загальний об'єм областей усіх блоків ланцюга. Якщо для розташування записів вдається використати меншу кількість блоків, то система здатна прийняти рішення щодо реорганізації структури блоків.

Однак яку б зі схем представлення записів у блоках не було обрано, не можна забувати про одну проблему, що супроводжує операції вилучення записів. Цілком можливо, що існують зовнішні покажчики, які адресують запис, що вилучається. Якщо подібні покажчики дійсно існують, то було б неправильно залишити їх у невизначеному («підвішеному») стані або дозволити посилатися на новий запис, який, імовірно, з часом буде вставлено у блок на місце вилученого. Звичайна практика передбачає розташування спеціальної *ознаки-«обеліска»*(*tombstone*) на позиції, яку займав вилучений запис. Ознаку необхідно утримувати тут до моменту цілковитої

реконструкції бази даних. Де саме розташована ознака-«обеліск» – це залежить від природи покажчиків на запис. Якщо покажчики адресують деякі фіксовані позиції, на яких розташовані інші покажчики, що посилаються на запис, то ознаки-«обеліски» ставлять на названі фіксовані позиції. Розглянемо два приклади.

1. Якщо використовують схему блока з таблицею зміщень записів (див. п.10.2), то роль «обеліска» може виконувати нульовий покажчик у таблиці зміщень, оскільки зовнішні покажчики посилаються саме сюди, а не на адресу запису безпосередньо.
2. Якщо для трансляції логічних адрес записів у фізичні використовують таблицю відповідностей (див. рис. 25), то ознаку-«обеліск» як нульовий покажчик розташовують на місці фізичної адреси видаленого запису.

Якщо необхідно замінити ознакою-«обеліском» саме запис, а не посилання на нього, то доцільно на самому початку заголовка запису передбачити спеціальний біт-«обеліск», присвоюючи йому значення 0, якщо запис не видалений, і 1 – у протилежному випадку. Після вилучення запису біт повинен залишатися на місці, проте всі наступні байти вивільненої ділянки блока можна використати для вставки нового запису (рис. 37). Якщо поля запису підлягають вирівнюванню (див. пункт 9.1), то одним байтом, що містить біт-«обеліск», відбутися не вдасться – можливо, доведеться пожертвувати чотирма або більше байтами.

Якщо система простежить за вмістом зовнішнього покажчика, який адресує вилучений запис, то виявить «обеліск», який засвідчуватиме, що запис вилучений і зчитувати наступні байти просто немає змісту.

Рис. 37. Позначення ознакою-«обеліском»запису 1 після вилучення

12.3. Оновлення

Операція оновлення вмісту запису постійної довжини не має впливу на структуру зберігання, оскільки нові дані повинні займати той самий простір, що й попередні. Проте після зміни значень запису змінної довжини система матиме ті ж проблеми (хіба що за винятком необхідності використання ознак-«обелісків»), які виникають під час операції вставки нових і вилучення існуючих записів.

Якщо об’єм нового вмісту запису перевищує попередньо встановлені межі, то знадобиться розширення вільної області за рахунок зсуву записів усередині блока або створення і використання блока переповнення. Якщо фрагменти запису змінної довжини зберігаються в окремому блоці, як проілюстровано на рис. 32, системі, ймовірно, доведеться впорядковувати елементи цього блока або створити новий блок, призначений для зберігання змінної порції даних запису. Якщо ж, навпаки, об’єм вмісту запису зменшується, то система отримує ті ж можливості консолідації і повторного використання вільного простору блока чи вилучення блоків переповнення, що і при вилученні запису.

СТРУКТУРИ ІНДЕКСІВ

У попередньому розділі розглянуто способи зберігання полів і записів. Тепер вивчатимемо, як представляються відношення та екземпляри класів повністю. Якщо вважати, що достатньо обмежитися розподілом записів, які містять дані кортежів чи об'єктів екземпляра класу по довільних дискових блоках, то це глибока помилка. Уявімо собі процес обробки найпростішого запиту виду *SELECT * FROM R,* зверненого до відношення *R*, записи якого «розкидані» по всьому диску. Системі у подібній ситуації доведеться провіряти кожен блок даних у вторинному сховищі, сподіваючись, що заголовки блоків і записів містять достатню інформацію для того, щоб визначити, де початок кожного запису, де завершення і до якого відношення він належить.

Краще рішення полягає в резервуванні для кожного відношення певної кількості блоків, можливо, навіть цілих циліндрів диска. Тепер у процесі пошуку блоків вмісту відношення системі вдасться уникнути, передусім, необхідності сканування всього простору дискового сховища. Проте подібний спосіб організації даних уже не в стані спростити задачу обробки дещо складнішого запиту – наприклад, *SELECT * FROM R WHERE a=10.* У цьому випадку є важливими створення і вибір *індексів* відношення, що дають змогу пришвидшити пошук таких кортежів, певні компоненти яких задовольняють заданій умові. Як проілюстровано на рис. 38, індекс – це деяка структура даних, яка як «параметр» отримує задану «властивість» записів (зазвичай, значення одного або декількох полів) і «швидко» знаходить записи, які мають цю властивість.

Вдалий індекс дає змогу відшукати необхідний запис, звертаючись тільки до невеликої частини повної множини записів (в ідеалі безпосередньо до групи шуканих записів). Поля, за значеннями яких індекс конструюється, називають *ключем пошуку*, або просто «ключем», якщо із контексту зрозуміло, що мова йтиме про індекс.

Індекси можуть будуватись як найрізноманітніші структури даних. Розглянемо варіанти індексів:

1) прості індекси для впорядкованих файлів;
2) вторинні індекси для невпорядкованих файлів;
3) *B*-дерева – модель, яку застосовують з метою організації індексів для довільних файлів;
4) геш-таблиці – альтернативну структуру представлення індексів, яка широко використовується.

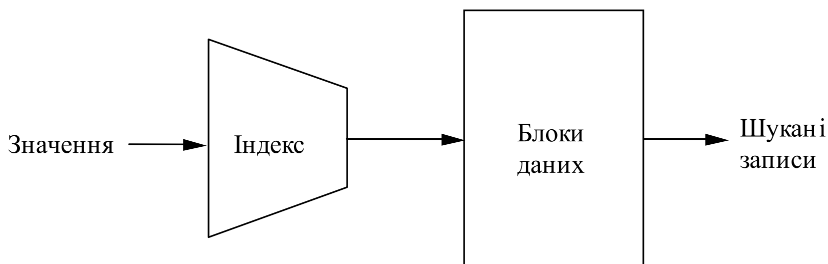


Рис. 38. Схема відшукання даних за допомогою індексу

Тема 13. Індекси для послідовних файлів

Ознайомимось з індексами з найпростішого варіанта структури: посортованому *файлові даних* (*data file)* відповідає *файл індексу (index file)*, сформований з пар виду *«ключ-покажчик»*. Значення *K* ключа пошуку в файлі індексу асоційоване з покажчиком, який посилається на запис файла даних, що володіє значенням *K*. Подібна індексна структура може бути *щільною (dense)* у тому розумінні, що *кожному* запису файла даних відповідає певний елемент файла індексу, або *розрідженою (sparse)* – файл індексу містить покажчики тільки на деякі записи файла даних (наприклад, один елемент з розрахунку на кожен блок даних).

13.1. Послідовні файли

Одна з найпростіших індексних структур ґрунтується на файлі, посортованому за значеннями атрибутів індексу. Подібний файл називається *послідовним (sequential file)*. Структура індексу для послідовного файла особливо корисна у тих випадках, коли ключ пошуку збігається з первинним ключем відношення. На рис.39

схематично зображено відношення, представлене як послідовний файл.

Кортежі відношення, яке зберігається у формі послідовного файла, посортовані за значеннями первинного ключа. Рисунок відповідає ситуації, коли ключами є цілочислові значення. Тут показано тільки ключові поля і висунуто доволі нетипове припущення про те, що кожен блок може містити тільки два записи. Наприклад, у першому блоці розміщені записи з ключами 10 і 20. Окрім того, у цьому і багатьох наступних прикладах як ключові значення використовують послідовні цілі числа, кратні 10, хоча, очевидно, що на практиці подібна вимога не ставиться – так само як і те, щоб в індексі і файлі даних були присутні всі значення, кратні 10, з визначеного інтервалу.

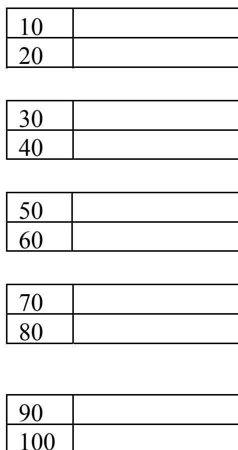
13.2. Щільні індекси

На основі посортованих записів файла даних можна створити *щільний індекс*, який являє собою послідовність блоків, що містять ключі записів даних і покажчики на ці записи (під покажчиками розуміють адреси – у тому розумінні, як говорилось у темі 10). Щільним називають індекс, який містить ключі для *кожного* запису файла даних. *Розріджені* індекси (ми розповімо про них у наступному пункті), навпаки, зазвичай, зберігають по одному ключовому значенню для кожного блока файла даних.

Записи файла щільного індексу впорядковують так само, як і записи файла даних. Оскільки ключі і покажчики займають суттєво менший простір, ніж повні записи, можна очікувати, що для зберігання індексу необхідно буде значно менше дискових блоків. Переваги індексного файла відчутні, передусім, у ситуації, коли його буде завантажено в ОП цілковито, а відповідний файл даних – ні. За допомогою подібного індексу вдається відшукати потрібний запис даних за заданим ключем шляхом єдиної операції дискового введення/виведення.

Приклад 37. На рис. 40 зображено початкові блоки файла індексу для послідовного файла, який наведено на рис. 39. Для зручності вважатимемо, що значення ключів кратні 10, хоча на практиці подібна закономірність трапляється, зазвичай, дуже рідко.

Вважатимемо також, що блоки індексного файла містять по чотири пари виду «*ключ-покажчик*». Зазначимо, що і ця гіпотеза далека від реальності – у типовому дисковому блоці можуть вміститися сотні подібних пар. Перший блок індексу містить покажчики на чотири початкові записи даних, другий блок – на наступні чотири записи і т.д. На практиці інколи доцільно заповнювати блоки індексного файла не цілком – докази на користь подібного рішення наведено в пункті 13.6.



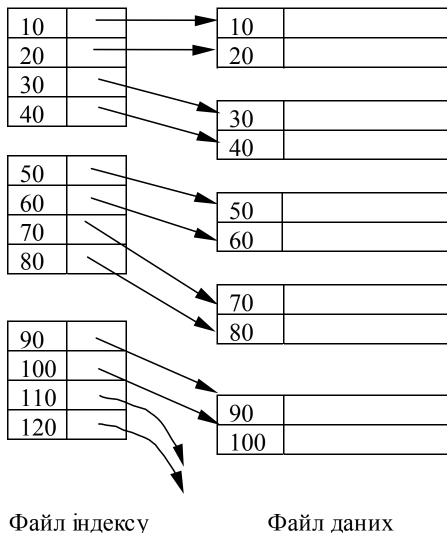


Рис. 39. Приклад послідовного файла

Рис. 40. Щільний індекс (зліва) для послідовного файла даних (справа)

Щільний індекс дає змогу системі успішно обробляти запити, в яких передбачено відшукання записів за заданим значенням ключа. Система переглядає блоки індексу в пошуках значення K ключа, а потім, коли запис індексу з ключем K знайдено, звертається за покажчиком, що адресує запис даних з тим же ключовим значенням. На перший погляд може видатися, що для відшукання запису індексу з ключем K системі доведеться переглядати кожен

блок індексу (або в середньому – половину блоків), однак існують чинники, які дають змогу значно збільшити продуктивність операцій пошуку за індексом.

1. Розмір файла індексу, зазвичай, суттєво менший від об'єму файла даних.
2. Оскільки ключі зберігаються упорядковано, то для відшукання значення K доцільно використовувати алгоритм бінарного пошуку. Якщо індекс містить n блоків, то системі доведеться звернутися тільки до $\log_2 n$ з них.
3. Індекс може бути таким малим, що його вдасться цілком завантажити в ОП і зберігати тут протягом необхідного часу. У цьому випадку для відшукання ключа пошуку K достатньо виконати кілька операцій в ОП, не звертаючись до диска.

Приклад 38. Нехай дано відношення, яке містить 1 000 000 кортежів. Відомо, що для зберігання кожних десяти кортежів необхідно один 4 096-байтовий блок, а загальний об'єм даних відношення становить приблизно 400 Мбайтів. Це надто багато, щоб завантажити інформацію в ОП цілковито. Припустимо, що довжина ключового поля становить 30 байтів, а розмір покажчика – 8 байтів. Тоді в один 4 096-байтовий блок (з урахуванням витрат на заголовок блока) вдається помістити 100 пар виду *«ключ-покажчик»*. Для побудови щільного індексу необхідно 10 000 блоків, або 40 Мбайтів. Такий індекс за певних умов можна завантажити у буфери ОП – все залежить від її загального об'єму і розмірів області, вільної від інших даних. Оскільки $\log_2(10\ 000)$ становить близько 13-ти, то для відшукання потрібного ключа доведеться виконати 13 або 14 звертань до блоків індексу. Оскільки при використанні будь-якого варіанта бінарного пошуку адресується незначна підмножина усіх блоків (наприклад, блок усередині списку, блоки на позиціях 1/4 і 3/4, на позиціях 1/8, 3/8, 5/8, 7/8 і т.д.), то навіть у випадку, якщо розмістити весь індекс у пам'яті неможливо, завантажити основні блоки все таки вдасться. Отож для отримання запису з довільним ключем необхідно значно менше, ніж 14 операцій дискового введення / виведення.

13.3. Розріджені індекси

Якщо щільний індекс виявиться надто великим, то інколи доцільно скористатися подібною структурою *розрідженого індексу*, яка дає змогу зменшити розмір файла ціною ймовірного збільшення проміжку часу, необхідного для відшукання запису за заданим значенням ключа. На рис. 41 проілюстровано розріджений індекс, який містить тільки по одному покажчику на кожен блок файла даних, а ключами індексу є ключові значення перших записів кожного блока даних.

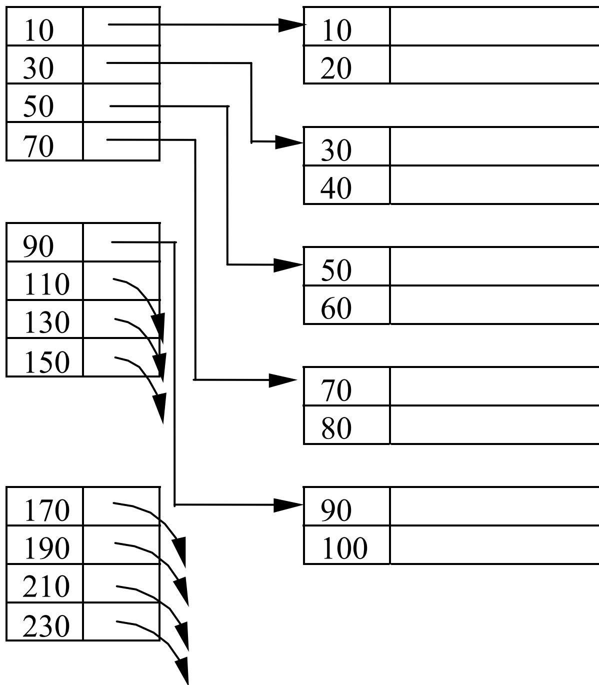


Рис. 41. Розріджений індекс для послідовного файла даних

Приклад 39. Вважатимемо, що файл даних є посортованим, значеннями ключа є цілі числа, кратні 10-ти, і кожен блок індексу містить по чотири пари виду *«ключ-покажчик»*. Отож перший блок

індексу містить елементи, які відповідають початковим записам перших чотирьох блоків даних, тобто 10, 30, 50 і 70. Аналогічно, другий індексний блок містить значення ключа для перших записів даних з номерами 5 – 8, а це 90, 110, 130 і 150. На рисунку зображено і третій блок індексу, покажчики якого посилаються на перші записи гіпотетичних блоків даних 9 – 12.

Приклад 40. Розріджений індекс, зазвичай, займає значно менше дискового простору, ніж щільний. Якщо знову звернутися до умови прикладу 38, де дані розміщувались у 100 000 блоків, а на кожен блок індексу припадало по 100 пар виду *«ключ-покажчик»*, і замість щільного індексу створити розріджений, то для зберігання останнього необхідно тільки 1 000 блоків і всього 4 Мбайти простору. Такий об'єм даних, імовірно, поміститься в буферах ОП.

З іншого боку, щільний індекс дає змогу системі обробляти запити виду «чи існує запис з ключовим значенням *K*?», не вимагаючи завантаження всього блока з шуканим записом. Гарантією існування запису з ключем *K* слугує значення *K,* присутнє у ключовому полі щільного індексу. Під час обробки цього ж запиту за участю розрідженого індексу системі доведеться виконати операцію дискового введення/виведення для завантаження блока, який (можливо!) містить запис з ключем *K*.

Щоб за допомогою розрідженого індексу відшукати запис даних з ключем *K*, треба знайти запис індексу з найбільшим ключовим значенням, меншим або рівним *K*. Оскільки файл індексу посортовано за ключем, то зазначений елемент індексу не важко виявити, застосовуючи модифікований варіант алгоритму бінарного пошуку. Покажчик, який відповідає знайденому ключу, адресує визначений блок даних. Звертаючись до цього блока, система повинна знайти в ньому запис з ключем *K*. Формати опису блока і запису даних покликані надавати відповідну інформацію, достатню для ідентифікації запису всередині блока та окремих елементів вмісту самого запису (за довідками звертайтесь до тем 9 і 10).

13.4. Багаторівневі індекси

Отож індекс може виявитися достатньо об'ємним. Навіть у тому випадку, якщо для відшукання елемента індексу використовують

алгоритм бінарного пошуку, то системі досить часто доводиться виконувати чималу кількість операцій дискового введення/виведення. З метою пришвидшення перегляду індексного файла можна створити додатковий індекс другого рівня «індекс для індексу».

На рис. 42 наведено запозичену з рис. 41 схему, доповнену індексом другого рівня (нехай ключем слугуватимуть числа, кратні 10). Можна також спробувати створити, за необхідності, індекси третього рівня, четвертого і т.д. Проте такому підходу властиві певні внутрішні обмеження. За необхідності конструювання індексів декількох рівнів перевага полягає у звертанні до структури B-дерева (детальніше – у темі 15).

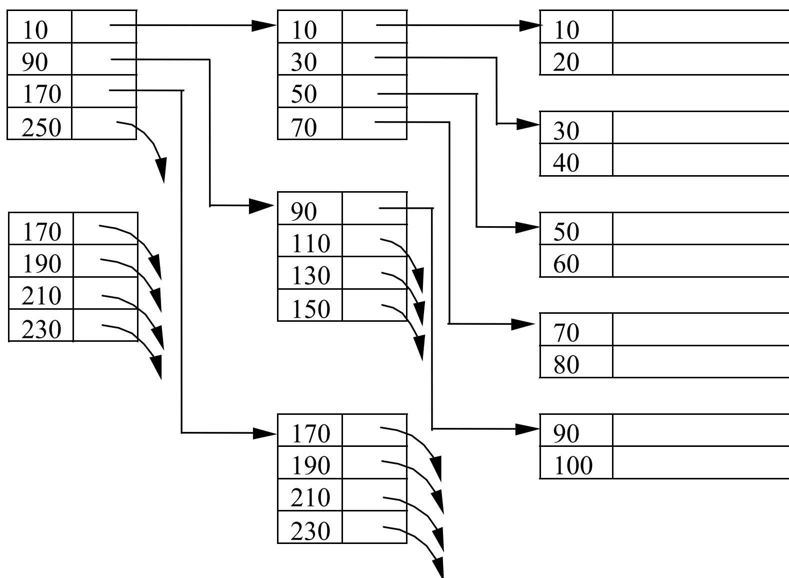


Рис. 42. Приклад дворівневого розрідженого індексу

У цьому випадку (рис. 42) індекс першого рівня є розрідженим, хоча для цього рівня можна було б створити і щільний індекс. Однак індекси усіх рівнів, окрім першого, зі зростанням номера рівня повинні бути розрідженішими, оскільки в протилежному

випадку система не отримає жодних переваг – індекс міститиме стільки ж елементів, скільки і «сусідній» індекс меншого рівня, і займатиме такий самий дисковий простір.

Приклад 41. Припустимо, що до розрідженого індексу першого рівня (приклад 40) долучають індекс другого рівня. Оскільки індекс першого рівня займає 1 000 блоків і дає змогу «упакувати» по 100 пар виду *«ключ-покажчик»* у кожен блок, то для індексу другого рівня необхідно буде 10 блоків. Цілком імовірно, що ці 10 блоків завантажать у буфери ОП водночас. Тоді для відшукання запису даних із заданим ключовим значенням *K* спочатку необхідно переглянути індекс другого рівня, щоб знайти у ньому запис з найбільшим ключовим значенням, яке менше або дорівнює *K*. Відповідний покажчик приведе до деякого блока *B* індексу першого рівня, звідки система, напевно, отримає доступ до шуканого запису даних, однак для початку потрібно зчитати блок *B* у ОП (якщо цього не зроблено раніше). Зауважимо, що поки-що це перша операція дискового введення/виведення, яку виконано. Тепер система зобов'язана переглянути блок *B* у пошуках найбільшого ключового значення, яке менше або дорівнює *K*, і елемент індексу з цим ключем безпосередньо вкаже на блок даних, що містить запис з ключовим значенням *K* – звичайно, якщо такий існує. Для зчитування знайденого блока даних необхідно виконати другу, і останню, операцію дискового введення/виведення.

13.5. Дублікати ключових значень

Досі ми вважали, що ключ пошуку, для якого створюється індекс, збігається з ключем відношення, тобто кожному значенню ключа відповідає не більше одного запису даних. Однак часто індекси конструюють для атрибутів, які не утворюють первинний ключ, отож цілком імовірна ситуація, за якої одне і те ж ключове значення присутнє в декількох записах. Якщо записи сортуються за ключами пошуку і порядок розташування записів з однаковими ключами довільний, то всі попередньо розглянуті підходи цілком придатні і для випадку, коли ключ пошуку не є ключем відношення.

Імовірно, найпростіший варіант розширення попередньої моделі полягає у створенні щільного індексу, в якому для кожного запису

файла даних зі значеннями ключа пошуку *K* існує елемент із ключем *K*. Іншими словами, у файлі індексу допускається присутність дублікатів значень ключа пошуку. Задача відшукання усіх записів даних, які містять заданий ключ пошуку *K*, є простою: ідентифікувати у файлі індексу перший екземпляр запису з ключем *K*, знайти усі інші екземпляри (вони повинні іти безпосередньо після першого) і переміститися за адресами покажчиків на записи даних з ключовим значенням *K*.

Дещо ефективніший варіант передбачає включення у файл щільного індексу по одному запису для кожного значення ключа пошуку *K*. Покажчик у цьому записі адресує перший із записів даних з ключовим значенням *K*. Щоб знайти решту записів даних з таким самим значенням ключа, достатньо зчитати наступні записи даних до моменту зміни вмісту ключового поля – оскільки файл даних є посортованим за ключовим полем, то записи з однаковими значеннями поля будуть зібраними в окремі групи. Ідею проілюстровано на рис. 43.

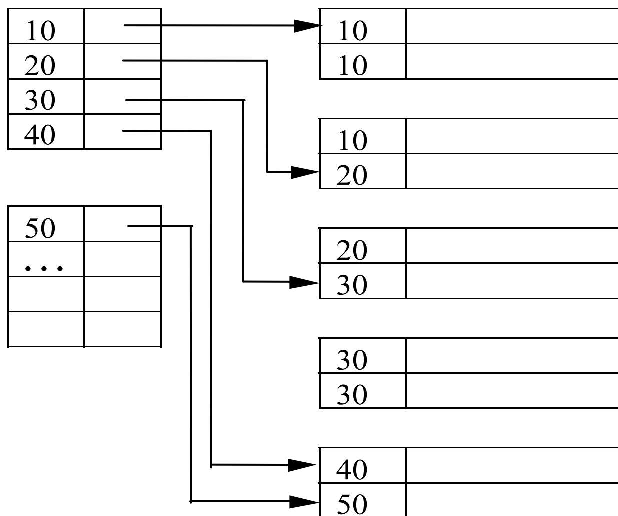


Рис. 43. Щільний індекс для файла з дублікатами значень ключа пошуку

Приклад 42. Припустимо, що необхідно відшукати такі записи файла даних, зображеного на рис. 43, які містять значення ключа пошуку рівного 20. Система виявить у файлі індексу елемент з ключем 20 і, стежачи за адресою покажчика, виявить перший запис даних з ключем пошуку 20. Потім необхідно переглянути наступні записи файла даних. Оскільки біжучим є останній запис другого блока, то система переходить до третього блока (вважатимемо, що блоки файла даних організовані у формі зв'язаного списку і заголовок кожного блока містить покажчик на наступний блок). Тут ключовим значенням 20 володіє перший запис, однак другий містить ключ 30. Отож процес пошуку завершується, оскільки записів з ключем 20 більше немає.

Отже, система виявить 2 записи зі значенням ключа пошуку, яке дорівнює 20.

На рис. 44 проілюстровано варіант розрідженого індексу для того ж файла даних, що і на рис. 43. Цей індекс виглядає як звичайно: він містить пари виду *«ключ-покажчик»*, що відповідають значенням ключа пошуку перших записів блоків даних.

Щоб знайти записи із заданим значенням *K* ключа пошуку в такій структурі, необхідно відшукати елемент індексу (назвемо його *E1*) з найбільшим номером, який має значення ключа, менше або рівне *K*. Потім необхідно переміщатися до початку індексу доти, доки не буде досягнуто першого елемента індексу чи елемента *E2* зі значенням ключа, строго меншим від *K* (у частковому випадку, коли *E1* вже містить значення ключа, яке строго менше від *K*, то *E2* збігається з *E1*). Блоки даних, які можуть містити записи зі значенням *K* ключа пошуку, адресуються елементами індексу з номерами з інтервалу від *E2* до *E1* включно).

Приклад 43. Припустимо, що у структурі, представленій на рис. 44, необхідно знайти записи даних зі значенням ключа пошуку, рівним 20. Роль *E1* у цьому випадку виконуватиме третій запис першого блока індексу – це елемент з найбільшим номером, такий, що значення його ключа не перевищує задану межу 20. Черговий елемент, найближчий до початку індексу, містить значення ключа (10), строго менше від 20. Отож елементом *E2* виявиться другий запис першого блока індексу. Покажчики в записах *E2* і *E1* індексу

обмежують шуканий діапазон файла даних другим і третім блоками – саме тут повинні розміщатися записи даних зі значенням ключа пошуку, рівним 20.

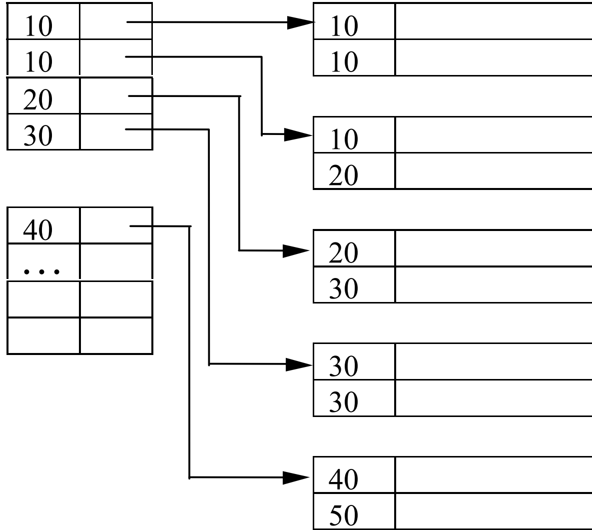


Рис. 44. Розріджений індекс, елементи якого містять найменші значення ключа в кожному блоці даних

Розглянемо ще один приклад. Якщо взяти *К=10*, тоді елемент *Е1* – це другий запис першого блока індексу, а елемента *Е2* насправді не існує, оскільки ключа з меншим значенням у складі індексу немає. Отож для відшукання записів даних з ключовим значенням 10 необхідно використати покажчики у першому і другому елементах індексу, які адресують два перших блоки даних.

Дещо іншу схему організації індексу проілюстровано на рис. 45. Тут елемент індексу, який відповідає блокові даних, містить найменше *нове* значення ключа пошуку, тобто таке, якого не було в попередньому блоці. Якщо у блоці даних немає нових значень ключа (усі записи володіють однаковим – «старим» – ключем), то у відповідний елемент індексу заноситься саме це значення. Система

зможе знайти записи даних зі значенням K ключа пошуку, якщо звернеться до першого елемента індексу, ключове значення якого

а) дорівнює K

або

б) менше від K, проте наступний ключ більший від K,

а потім переміститься за адресою, на яку посилається покажчик у цьому елементі. Якщо знайдений блок даних містить хоча б один запис з ключем K, то решту записів з тим же ключем (якщо вони існують) вдасться відшукати, переглядаючи вміст наступних блоків.

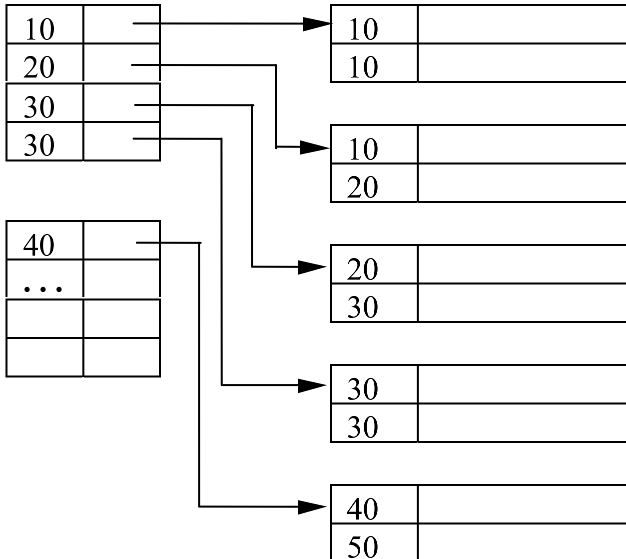


Рис. 45. Розріджений індекс, елементи якого містять найменші нові значення ключа у кожному блоці даних

Приклад 44. Звернемось до структури, наведеної на рис. 45, і припустимо, що K=20. За визначеним вище правилом обирається другий елемент індексу – його покажчик приводить до першого блока даних, який містить запис з ключем 20 (це блок з номером 2).

Переходячи до наступного блока даних, система зможе відшукати ще один запис з ключем 20.

Якщо взяти K=30, то правило змусить нас звернутися до третього елемента індексу, покажчик якого адресує третій блок даних, що містить перший екземпляр запису з ключем 30 (подальші дії очевидні). Зрештою, якщо покласти K=25, то доведеться скористатися пунктом (*б*) правила і зчитати другий елемент індексу, покажчик якого посилається на другий блок даних. Якщо б файл даних містив записи зі значеннями ключа пошуку, рівними 25, то хоча б один з них був би бути присутнім у цьому блоці, оскільки, як нам відомо, «нове» значення ключа в третьому блоці дорівнює 30. Оскільки записів з ключем 25 у файлі даних немає, то процедура пошуку завершується безуспішно.

13.6. Керування індексами під час модифікації даних

До цього часу ми вважали, що файли даних та індексів є послідовностями блоків, щільно заповнених записами відповідних типів. Оскільки з часом інформація змінюється, то можна очікувати, що користувачі системи, ймовірно, звертатимуться до файлів даних з командами вставки, вилучення та оновлення записів. Отож зовсім не обов'язково, що сьогоднішній вміст блоків структури, подібної до послідовного файла, у незмінному вигляді збережеться і завтра. Для реорганізації файлів даних використовують методи і прийоми, зазначені у темі 12.

Нагадаємо деякі з них.

1. Створення *блоків переповнення*, якщо не вистачає місця в існуючих блоках даних, і вилучення блоків переповнення у випадку вивільнення крупних областей у блоках після вилучення групи записів. Розріджений індекс не містить елементів, які вказують на блоки переповнення, – останні трактуються як розширення основних блоків.
2. Іноді існує можливість замість блоків переповнення створювати нові основні блоки даних, розміщуючи їх у відповідних позиціях послідовності блоків файла. У цій ситуації доданим блокам даних відповідатимуть нові елементи розрідженого індексу. Необхідно пам'ятати,

що зі зміною індексу можуть виникнути ті ж проблеми, які стосуються загального компонування файла під час виконання операцій вставляння, вилучення і оновлення записів файла даних. Під час створення нових блоків індексу системі необхідно розташовувати їх настільки «вдало», наскільки це можливо.

3. Якщо під час вставляння запису у блок даних виявиться, що у блоці немає вільних областей достатньої довжини, то можна здійснити переміщення записів у сусідні блоки з метою вивільнення необхідного простору. Якщо ж, навпаки, після вилучення записів сусідні блоки виявляються заповненими нещільно, то інколи їхні дані вдається об'єднати в один блок.

Зміна файла даних часто спричинює до необхідності відповідної модифікації файла індексу. Вибір підходу часто залежить від приналежності індексу до числа щільних або розріджених і від того, які із перелічених вище методів використовують у процесі зміни файла даних. Однак існує один загальний принцип, справедливий у будь-якій ситуації.

- Файл індексу є послідовним. Пари виду *«ключ-покажчик»* можна сприймати як записи, посортовані за значеннями ключа пошуку. Отож усі стратегії, які використовують для підтримки процедур модифікації послідовних файлів даних, є застосовні і до файлів індексів.

На рис. 46 перелічено усі дії, які відбуваються з файлами розріджених і щільних індексів, під час виконання кожної із семи різних операцій над файлами даних – створення/вилучення порожніх блоків переповнення або блоків послідовного файла, а також вставляння, вилучення і переміщення записів. Вважають, що створювати або вилучати можна тільки порожні блоки. Зокрема, якщо вилученню підлягає блок, що містить записи, то спочатку необхідно «позбутися» записів цього блока – вилучити їх чи перенести в інший блок.

Дія	Щільний індекс	Розріджений індекс
Створення порожнього блока переповнення	___	___
Вилучення порожнього блока переповнення	___	___
Створення порожнього послідовного блока	___	Вставка
Вилучення порожнього послідовного блока	___	Вилучення
Вставка запису	Вставка	Оновлення (?)
Вилучення запису	Вилучення	Оновлення (?)
Переміщення запису	Оновлення	Оновлення (?)

Рис. 46. Вплив на файл індексу дій, що стосуються послідовного файла даних

Розглянемо вміст таблиці детальніше.

- Операції створення або вилучення порожнього блока переповнення не впливають на індекси обох типів: елементи щільного індексу вказують на окремі записи; елементи розрідженого індексу адресують тільки основні блоки, а не блоки переповнення.
- Операції створення або вилучення блока послідовного файла не впливають на щільний індекс, оскільки його елементи посилаються на записи, а не на блоки. Якщо ж індекс є розрідженим, то під час створення або вилучення блока даних система зобов'язана, відповідно, вставити у файл індексу або вилучити з нього певний елемент.
- Під час вставляння або вилучення запису даних змінюється і щільний індекс – у нього необхідно вставити відповідну пару виду *«ключ-покажчик»* або видалити таку пару. Подібна ситуація, зазвичай, не впливає на розріджений індекс. Єдиний виняток виникає у ситуації, коли зачіпається перший запис у блоці даних, – у цьому випадку підлягає оновленню і відповідне ключове значення у файлі розрідженого індексу. Отож три останніх елементи стовпця «Розріджений індекс» таблиці відзначені символом «знак запитання» – операція оновлення індексу, можливо, буде необхідною, проте не завжди.

- Аналогічно, операція переміщення запису (як усередині блока, так і між блоками) спричинює до необхідності оновлення відповідного елемента щільного індексу, проте зачіпає розріджений індекс тільки у тому випадку, коли запис, що переміщається, займав або повинен зайняти першу позицію у блоці даних.

> ✓ ***Корисно знати***. *Як передбачити результати еволюції даних*. Оскільки відношенням і екземплярам класів властива стійка тенденція до зростання об'ємів інформації, то часто розумно наперед закласти у блоки даних і індексів додаткові вільні області. Якщо до моменту початку активної експлуатації бази даних заповнено до 75% простору блоків, то можна сподіватися, що деякий час вдасться обійтися без створення блоків переповнення і переміщення записів між блоками. Якщо кількість блоків переповнення зведена до мінімуму, то кожна процедура діставання запису буде потребувати, в середньому, тільки однієї операції дискового введення/виведення. Чим більше блоків переповнення, тим вищі затрати на пошук запису із заданим ключем.

Для ілюстрації особливостей алгоритмів, які використовують у перелічених нами ситуаціях, ми наведемо кілька прикладів, де розглянемо і щільні/розріджені індекси, і техніку переміщення записів та використання блоків переповнення.

Приклад 45. Спочатку розглянемо операцію вилучення запису з блока послідовного файла даних за наявності щільного індексу. Звернемося до структури, зображеної на рис. 40, і припустимо, що вилученню підлягає запис з ключем 30. На рис. 47 проілюстровано фрагмент тієї ж структури після виконання операції вилучення.

Спочатку з послідовного файла даних видаляють запис з ключем 30. Цілком можливе існування зовнішніх покажчиків, які посилаються на запис блока, що залишився, отож пересувати записи на початок блока недоцільно: на місці видаленого запису система здатна «поставити» ознаку-«обеліск» (див. п. 12.2).

Далі з індексу вилучають пару *«ключ-покажчик»* зі значенням ключа 30. Зовнішні покажчики на записи індексу, ймовірно, відсутні і необхідності у використанні «обеліска» немає. Отож є сенс перемістити записи, що залишилися, у блоці індексу, на одну позицію «вверх» для того, щоб консолідувати вільні ділянки усередині блока (якщо їх декілька) в одну крупнішу область.

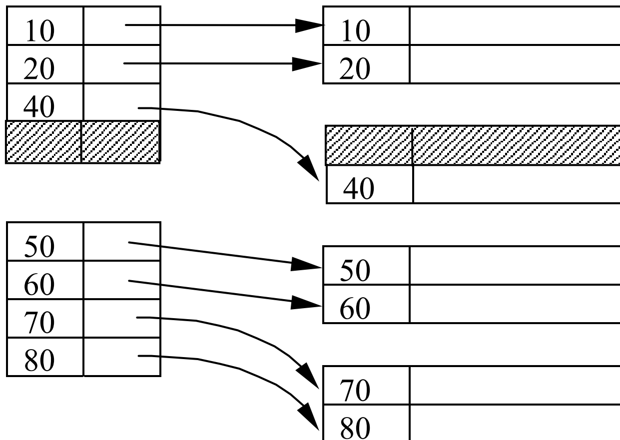


Рис. 47. Результат вилучення запису з ключем 30 зі структури з щільним індексом (див. рис. 40)

Приклад 46. Тепер продемонструємо, як виконуються операції вилучення двох записів із послідовного файла даних, який має розріджений індекс. Візьмемо за основу структуру, зображену на рис. 41, і припустимо, що першим необхідно видалити запис з ключем 30. Окрім того, вважатимемо, що переміщенню записів між блоками ніщо не загрожує – або наперед відомо, що зовнішні покажчики, які могли б посилатися на записи усередині блока, відсутні, або для підтримки операцій переміщення записів між блоками використовується таблиця зміщень, аналогічна до представленої на рис. 35.

Результат вилучення запису даних з ключем 30 схематично зображено на рис. 48. Після вилучення запису з ключем 30

наступний запис блока, який має ключ 40, зміщений «вверх» з метою укрупнення вільної області всередині блока. Оскільки запис з ключем 40 тепер займає першу позицію у другому блоці даних, то необхідно оновити вміст елемента ключа, який посилається на цей блок. На рис. 48 проілюстровано, що значення ключа, якому відповідає покажчик, що адресує другий блок даних, змінено з 30 на 40.

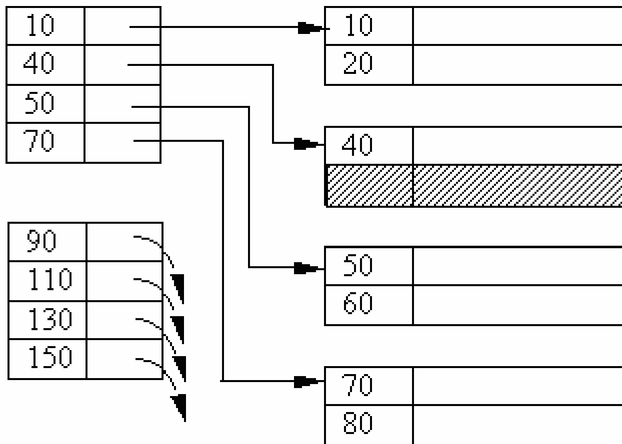


Рис. 48. Результат вилучення запису даних з ключем 30 зі структури з розрідженим індексом (див. рис. 41)

Нехай тепер необхідно видалити і запис даних з ключем 40. Підсумок операції наведено на рис. 49. Другий блок даних стає порожнім. Якщо послідовний файл зберігається у довільних блоках (а не у суміжних блоках циліндра, що було б також цілком можливо), то вивільнений блок можна долучити до списка областей дискового простору, які не використовують.

Щоб завершити операцію, необхідно модифікувати і файл індексу. Оскільки другого блока даних вже немає, то відповідний елемент індексу необхідно вилучити. Записи індексу, як проілюстровано на рис. 49, можна змістити до початку блока, хоча ця дія не обов’язкова.

Приклад 47. Зосередимо увагу на операції вставляння запису даних. Звернемося до рис. 48, який ілюструє ситуацію, коли запис з ключем 30 щойно вилучено з файла даних з розрідженим індексом, однак запис, що містить ключ 40, залишається на місці. Припустимо, що необхідно вставити у файл даних запис з ключем 15. Звертаючись до файла розрідженого індексу, визначимо, що запис з ключем 15 повинен належати першому блокові даних. Але цей блок заповнений – у ньому вже є записи з ключами 10 і 20.

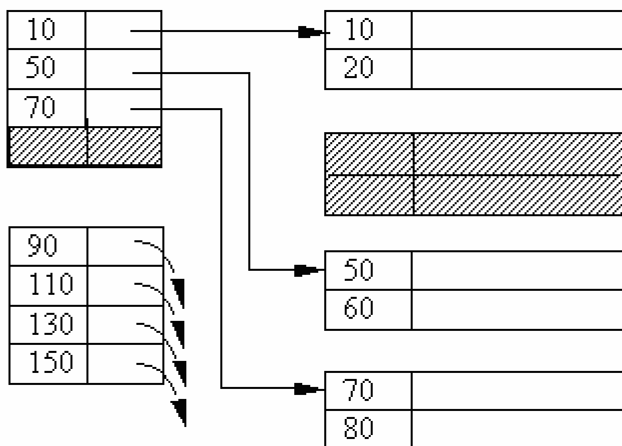


Рис. 49. Результат вилучення запису з ключем 40 (див. рис. 48)

Один із варіантів подальших дій передбачає пошук найближчого блока даних, в якому є вільна область необхідного розміру. У нашому випадку пошук приводить до другого блока. Запис з ключем 20 переноситься з першого блока у другий, звільняючи місце для запису з ключем 15. Результат наведено на рис. 50. Для розміщення запису 20 у другому блоці запис 40 необхідно зсунути на одну позицію «вниз», щоб зберегти встановлений порядок сортування.

На останньому кроці необхідно модифікувати вміст індексу. За певних обставин може виникнути ситуація, коли нам доведеться

оновити вміст поля ключа у відповідному елементі індексу, проте у цьому випадку такого робити не доведеться, оскільки вставлений запис не є першим записом блока. Необхідно, однак, модифікувати ключ у другому елементі індексу, адже значення ключа у першому записі другого блока даних змінилось з 40 на 20.

Приклад 48. Підхід, представлений у прикладі 47, не спричинює до проблем тільки у тому випадку, коли вільну область вдається знайти безпосередньо у сусідньому блоці даних. Якщо б запис з ключем 30 попередньо не видалили, то процес пошуку ділянки, придатної для вставки нового запису, не був би таким коротким – для звільнення місця під новий запис необхідно було б послідовно зміщати всі записи, починаючи з 20-го, до кінця файла і створювати додатковий блок.

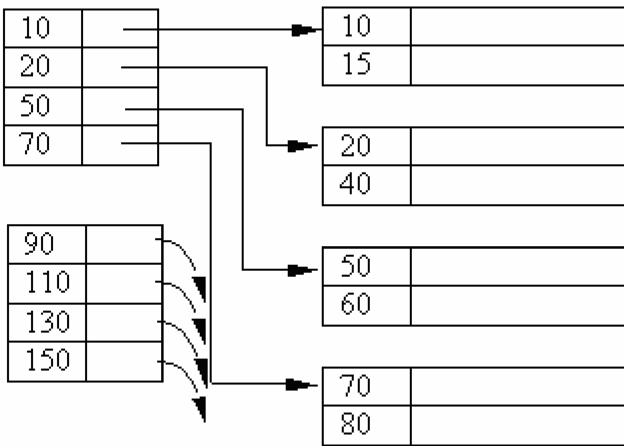


Рис. 50. Результат вставки запису даних з ключем 15 у структуру з розрідженим індексом (див. рис. 48)

Якщо файл містить велику кількість записів, то така стратегія, очевидно, приведе до серйозних витрат, отож доцільніше скористатися альтернативним механізмом створення блоків переповнення. На рис. 51 проілюстровано результат вставки запису

з ключем 15 у блок переповнення, долучений до структури, яку наведено на рис. 48. У першому блоці даних вільної області, достатньої для вставки нового запису, немає. Замість переміщення запису з ключем 20 у другий блок створюється блок переповнення для першого основного блока.

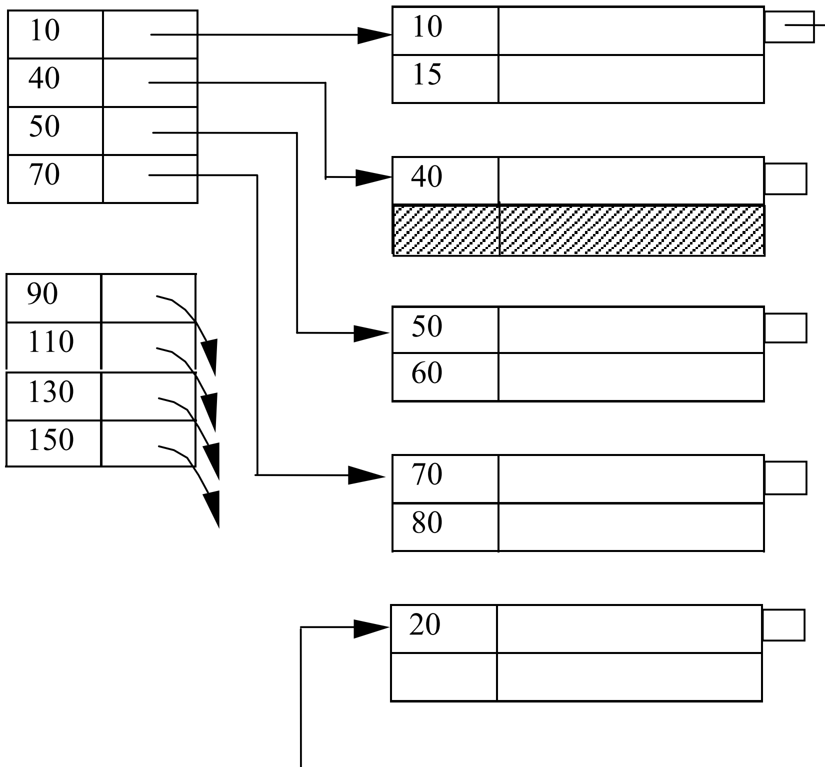


Рис. 51. Операція вставки запису даних, яка передбачає використання блока переповнення

На рис. 51 кожен блок даних позначено справа додатковим прямокутним «виступом», який символічно представляє те місце у заголовку блока, де розташований покажчик на відповідний блок переповнення. Отож до одного блока переповнення легко

приєднати другий, до другого третій і т.д. Ланцюг блоків переповнення для основного блока утворює зв'язаний список.

Отже, запис з ключем 15 вставляється на відповідну позицію блока після запису, що містить ключ 10, а запис з ключем 20, який звільняє необхідний простір у блоці, переміщається у новоутворений блок переповнення. Змінювати індекс не потрібно, оскільки перший запис другого блока даних залишається недоторканим. Зверніть увагу, що окремий елемент індексу для блока переповнення не створюється – він трактується як розширення основного блока даних, а не як самостійний блок послідовного файла даних.

Вправи для опрацювання

Вправа 38. Припустимо, що в блок поміщаються три записи даних або десять пар вигляду *«ключ-покажчик»*. Запишіть у вигляді функції від числа n записів даних кількість блоків, необхідних для зберігання файла даних і

а) щільного індексу;

б) розрідженого індексу.

Вправа 39. Виконайте завдання з вправи 38, якщо кожен блок може містити до 30-ти записів даних або 200 індексних елементів *«ключ-покажчик»*, однак жоден з блоків даних і блоків індексу не повинен заповнюватися понад 80%.

Вправа 40. Виконайте завдання з вправи 38, якщо передбачається створення такої кількості рівнів індексів, за якої верхній рівень індексу налічуватиме лише один блок.

Вправа 41. Зверніться до структури, наведеної на рис. 50, і продовжуйте процес модифікації даних, який передбачає вилучення записів даних з ключами 60, 70 і 80 з наступним вставленням записів з ключами 21, 22 і т.д. аж до запису з ключем 29, вважаючи, що необхідний простір надається так:

а) створенням блоків переповнення для файла даних чи файла індексу;

б) переміщенням записів «вниз» і створенням додаткових блоків наприкінці файла даних чи файла індексу;

в) вставлянням нових блоків даних чи індексних блоків усередину відповідного файла.

Тема 14. Вторинні індекси

Структури, про які йшла мова у темі 13, прийнято називати *первинними індексами* (primary indexes) – вони визначають розташування індексованих записів у файлі даних, який посортовано за ключем пошуку.

Іноді виникає необхідність створення *декількох* індексів для одного і того ж відношення, які дають змогу пришвидшити обробку запитів різних категорій. Наприклад, первинним ключем відношення *MovieStar* (див. рис. 20) є атрибут *name* (*ім'я*), отож можна створити первинний індекс на основі значення цього атрибута, вважаючи, що у чималій кількості запитів задаватиметься ім'я актора. Однак припустимо, що інформацію відношення *MovieStar* часто використовують і для підготовки привітальних послань акторам з нагоди їхнього ювілею. З цією метою використовують запити вигляду

```
SELECT name, address
FROM MovieStar
WHERE birthdate = DATE '1952-01-01';
```

Для підвищення ефективності процесів обробки подібних запитів доцільним буде *вторинний індекс* (*secondary index*) для атрибута *birthdate*. У SQL-системі для створення такого індексу використовують команду

```
CREATE INDEX BDIndex ON MovieStar (birthdate);
```

Вторинний індекс виконує ту ж функцію, що й будь-який інший індекс. Він є структурою, яка полегшує пошук записів за заданими значеннями одного або декількох полів. Однак вторинний індекс відрізняється від первинного тим, що він *не* визначає, а тільки вказує позицію записів у файлі даних. З метою визначення взаємного розташування записів можна використовувати первинний індекс для деякого іншого атрибута. Важливий

наслідок, зумовлений різницею між первинним і вторинним індексами, є такий:

- створювати *розріджені* вторинні індекси немає змісту, оскільки вторинний індекс не впливає на фізичне положення записів у файлі даних і його не можна використовувати як інструмент передбачення позиції довільного запису, ключ якого не згадано у файлі індексу явно; вторинні індекси завжди повинні бути *щільними*.

14.1. Проектування вторинних індексів

Вторинний індекс – це щільний індекс, який, зазвичай, містить дублікати ключових значень. Як і раніше, індекс складається з пар значень виду *«ключ-покажчик»*. «Ключ» у цьому випадку – це ключ пошуку, і вимоги щодо його унікальності не ставлять. Елементи файла індексу посортовані за значеннями ключа – це дає змогу пришвидшити пошук необхідного елемента. Якщо необхідно доповнити структуру індексу другим рівнем, то лише розрідженим – відповідні докази ми приводили у пункті 13.4.

Приклад 49. На рис. 52 наведено типову структуру даних, яка містить вторинний індекс. Відповідно до ухвалених нами домовленостей блоки файла даних містять по два записи. На рисунку зображено тільки значення ключових полів записів і ці значення, як і раніше, кратні 10. Зауважте, що, на відміну від файлів даних, розглянутих у пункті 13.5, файл даних у цьому випадку не є посортованим за ключем пошуку. Проте елементи файла індексу *посортовані* за значеннями ключа. Отож покажчики в записах одного блока вторинного індексу здатні адресувати найрізноманітніші блоки даних, а не один або декілька послідовних блоків, як за умови використання первинного індексу. Наприклад, щоб вийняти усі записи даних з ключем 20, доведеться не тільки звернутися до двох блоків індексу, але й звернутися за відповідними значеннями покажчиків до трьох різних блоків даних.

Отже, застосування вторинних індексів загалом вимагає виконання значно більшої кількості операцій дискового введення/виведення порівняно з тим, коли та ж кількість записів даних є впорядкованою за допомогою первинного індексу. Проте

ми не можемо сподіватись на те, що порядок записів даних виявиться саме таким, який необхідний, адже він може залежати від інших атрибутів.

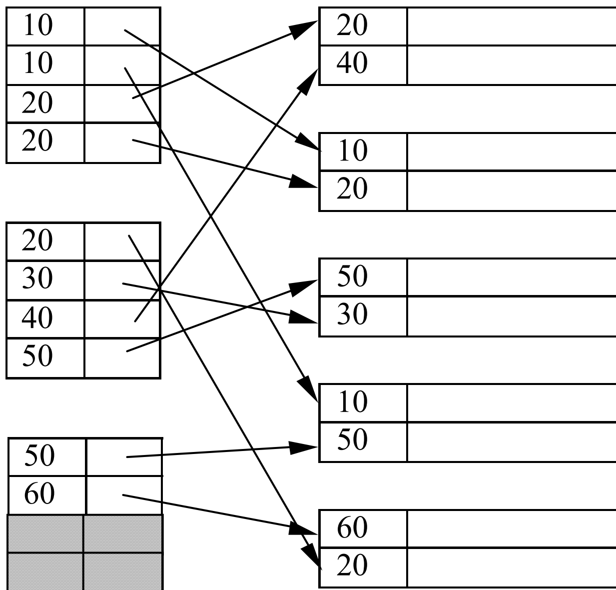


Рис. 52. Приклад структури з вторинним індексом

Цілком можливо доповнити структуру, подібну до наведеної на рис. 52, другим рівнем індексу. Цей – розріджений – індекс повинен містити елементи, які адресують найменші або найменші нові значення ключів кожного блока даних (про це йшлося у пункті 13.5).

14.2. Використання вторинних індексів

Крім забезпечення можливості швидкої обробки різних запитів до відношень (або екземплярів класів), організованих у вигляді послідовних файлів, вторинні індекси здатні виконувати корисні функції і при використанні зі структурами інших типів, такими як

«*купа*» (*heap*), де записи даних розташовують у довільному порядку.

Вторинні індекси часто використовують і для оптимізації доступу до структур іншої категорії, які часто використовуються і називаються *компактно згрупованими файлами (clustered files).* Розглянемо відношення *R* і *S*, з'єднані зв'язком типу «багато до одного», спрямованого від *R* до *S*. У деяких ситуаціях має сенс зберігати кожен кортеж *R* спільно із зв'язаним кортежем *S*, а не впорядковувати *R* за первинним ключем. Проілюструємо сказане за допомогою реального прикладу.

Приклад 50. Звернемося до знайомих відношень «кінематографічної» бази даних:

Movie (title, year, length, inColor, studioName, producerC#)

Studio (name, address, presC#)

Припустимо, що одним із «найпопулярніших» запитів є такий:

```
SELECT title, year
FROM Movie, Studio
WHERE presC# = zzz AND Movie.studioName =
Studio.name;
```

де *zzz* – деякий конкретний сертифікований номер президента кіностудії. Отже, йдеться про відшукання інформації щодо всіх фільмів, знятих на кіностудії, президентом якої є особа з заданим сертифікованим номером.

Якщо такий запит дійсно належить до найтиповіших, тоді замість впорядкування кортежів відношення *Movie* за значеннями первинного ключа *(title, year)* доцільніше створити структуру *компактно згрупованого файла*, яка об'єднує інформацію відношень *Studio* і *Movie* так, як проілюстровано на рис. 53. Кожен кортеж відношення *Studio*, що описує дані про деяку кіностудію, супроводжує група кортежів відношення *Movie*, які представляють відомості щодо усіх кінофільмів, випущених цією студією.

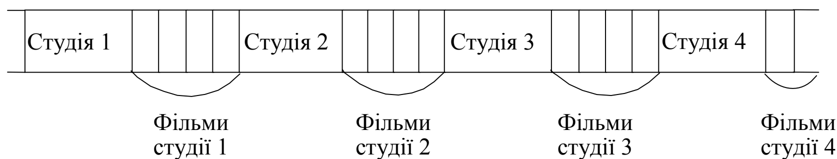


Рис. 53. Приклад компактно згрупованого файла

Якщо створити індекс для відношення *Studio* за ключем пошуку *presC#*, то це даватиме змогу швидко відшукати кортеж з інформацією про кіностудію для довільно заданого значення *zzz*. Окрім того, всі кортежі-записи *Movie*, вміст компонентів *StudioName* яких збігається зі значенням компонента *name* кортежу *Studio*, розташовані у компактно згрупованому файлі безпосередньо після відповідного запису *Studio*. Отож для отримання даних про кінофільми, випущені певною студією, достатньо буде виконати таку кількість операцій дискового введення/виведення, яка близька до теоретичного мінімуму.

14.3. Додаткові рівні у вторинних індексах

Використання структури, наведеної на рис. 52, інколи пов'язане з зайвими (і не малими) затратами дискового простору. Якщо значення ключа пошуку присутнє у файлі даних *n* разів, то воно стільки ж разів фігурує у файлі індексу. Було б доцільніше зберігати тільки одну копію значення ключа спільно з усіма покажчиками, які адресують записи даних, що володіють цим ключем.

Щоб уникнути необхідності повторення ключових значень, можна розмістити між вторинним індексом і файлом даних додатковий рівень індексу, який складається із груп-*сегментів* (*buckets*) покажчиків. Кожному відповідному значенню *K* ключа пошуку відповідає єдиний елемент вторинного індексу, покажчик якого адресує певну позицію у «файлі сегментів», де розташована група покажчиків, що відповідають ключу *K* (рис. 54).

Приклад 51. Звернемось до структури, яку зображено на рис. 54, і простежимо від елемента індексу з ключем пошуку 50 у напрямі покажчика до відповідної позиції у проміжному «файлі сегментів».

На цій позиції розміщений покажчик на запис даних, який займає останнє місце в блоці файла сегментів. Наступний покажчик з тієї ж групи розташований на початку другого блока файла сегментів, а далі іде покажчик, який пов'язаний з іншим елементом (який містить ключ 60) індексного файла. Отже, файл сегментів містить групу з двох покажчиків, які посилаються на записи даних зі значенням ключа пошуку, що дорівнює 50.

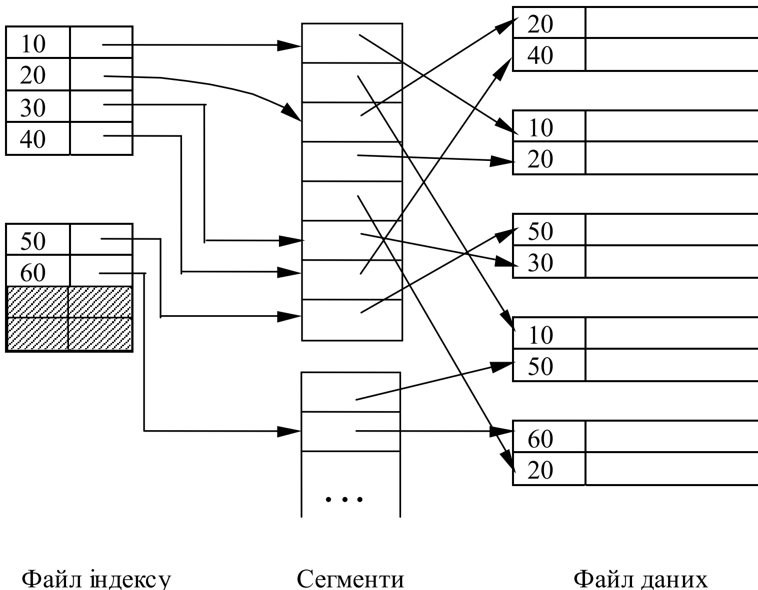


Рис. 54. Додатковий рівень опосередкування у вторинному індексі

Схема, наведена на рис. 54, дає змогу зекономити простір на диску тільки у тому випадку, коли значення ключа пошуку вимагає більше місця, ніж вміст покажчиків, і повторюється не менше, ніж два рази. Додатковий рівень опосередкування між вторинним індексом і файлом даних, однак, обіцяє суттєві переваги і тоді, коли досягти значної економії пам'яті не вдається: часто покажчики у файлі сегментів можуть допомогти системі обробити запит з мінімальною кількістю звернень до файла даних. Зокрема, якщо запит містить декілька умов і кожній з них відповідає певний

вторинний індекс, то шляхом операції пошуку перетину множини покажчиків в оперативній пам'яті можна виявити покажчики, які задовольняють водночас усі умови, і обмежити набір записів, які зчитуються, тільки тими, які адресуються покажчиками результуючої множини. Отож системі вдасться уникнути витрат, пов'язаних із завантаженням записів, що задовольняють деяким, однак не всім, умовам [2].

Приклад 52. Розглянемо відношення:

Movie (title, year, length, inColor, studioName, producerC#)

із «кінематографічної» бази даних. Припустимо, що створені вторинні індекси з проміжними сегментами для значень атрибутів *studioName* і *year* і необхідно отримати відповідь на запит

```
SELECT title, year
FROM Movie
WHERE studioName = 'Disney' AND year = 1995;
```

який має відшукати інформацію про всі кінофільми, випущені студією «Disney» у 1995 році.

На рис. 55 наведено схему дій, необхідних для опрацювання запиту із залученням вторинних індексів. За допомогою індексу для атрибута *studioName* можна швидко знайти покажчики на всі записи, що відповідають кінофільмам студії «Disney» (зауважте, що йдеться тільки про покажчики – жодного запису ще не завантажено в ОП). У свою чергу, індекс для атрибута *year* сприятиме виявленню покажчиків на записи з описом усіх кінофільмів, які вийшли на екрани у 1995 році. Тепер достатньо виконати операцію перетину двох множин покажчиків, щоб виявити саме ті із них, які відповідають записам про кінофільми студії «Disney», зняті у 1995 році, а потім завантажити в пам'ять усі блоки даних (і тільки такі

[2] Трюк з перетином множини покажчиків можна використовувати і в тому випадку, коли покажчики розташовані у файлі індексу, а не у проміжних групах-сегментах. Однак наявність файла сегментів часто знижує витрати на операції дискового введення-виведення, оскільки покажчики менш об'ємні, ніж пари «ключ-покажчик» індексу.

блоки), що містять записи, які адресуються покажчиками підсумкової множини.

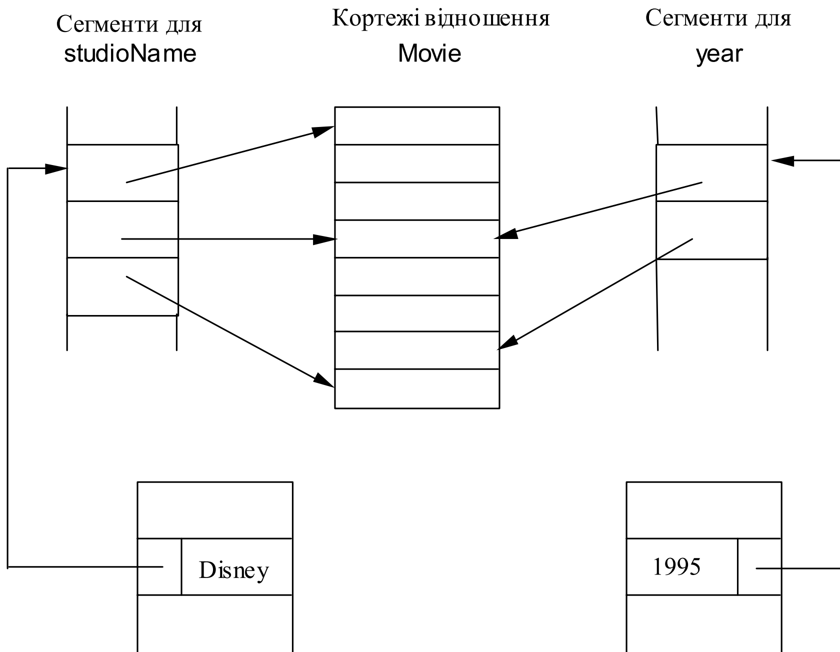


Рис. 55. Перетин множин покажчиків в ОП

14.4. Пошук документів і звернені індекси

Протягом тривалого часу користувачі інформації були стурбовані проблемами ефективного пошуку документів за заданою множиною ключових слів. З приходом технологій Word Wide Web, які забезпечують можливість доступу до документів у режимі on-line, ці проблеми загострюються, отож заслуговують на підвищену увагу. Для відшукання відповідних документів користувачі застосовують найрізноманітніші запити. Найпростішу і поширену форму подібних запитів можна представити наступним способом.

- Інформація, яка описує документ, трактується як кортеж деякого відношення *Doc*. Відношення має дуже велику

кількість атрибутів, по одному на кожне слово в документі. Кожен атрибут зачислено до логічного типу. Компонент атрибуту засвідчує, чи присутнє відповідне слово у документі, чи ні. Приклад схеми відношення *Doc* виглядає так: Doc (hasCat, hasDog, …), де компонент, подібний до *hasCat*, має значення TRUE, якщо і тільки якщо документ містить у крайньому випадку один екземпляр відповідного слова (у нашому випадку –*cat*).

- Кожен атрибут відношення *Doc* має вторинний індекс; індекс містить тільки такі елементи, які відповідають значенням TRUE ключа пошуку, тобто вказують на документи, які містять певне слово.
- Усі індекси об'єднуються в одному, *зверненому* (*inverted*), індексі. Звернений індекс використовується спільно з проміжним рівнем груп-сегментів покажчиків.

Приклад 53. Спосіб застосування зверненого індексу проілюстровано на рис. 56. На місці файла із записами використано колекцію документів, кожен з яких зберігається в одному або декількох дискових блоках. Звернений індекс сформовано з множини пар вигляду «*слово-покажчик»*». Слова виконують функцію значень ключа пошуку. Звернений індекс, як і будь-який інший індекс, міститься в послідовних блоках. Однак у застосуваннях, які стосуються пошуку документів, інформація більш статична, ніж вміст звичайних баз даних. Отож ситуації, що передбачають необхідність створення блоків переповнення або зміни індексу, виникають рідше.

Покажчики в елементах індексу адресують позиції «файла сегментів». На рис. 56, наприклад, покажчик запису індексу з ключовим словом *cat* посилається на початок групи покажчиків, які адресують документи, що містять у своєму складі слово *cat*. На рисунку символічно представлено деякі з таких документів. Аналогічно, елемент з ключем *dog* спрямований до групи покажчиків, які посилаються на документи зі словом *dog*.

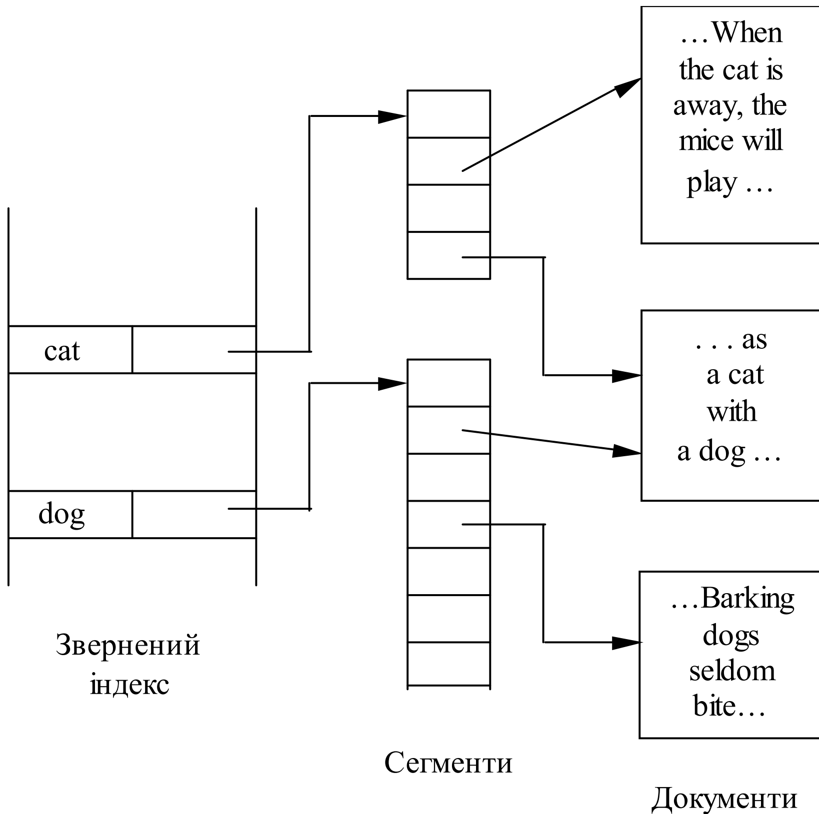


Рис. 56. Пошук документів за допомогою зверненого індексу

✓ ***Корисно знати.*** *Дещо про пошук інформації за ключовими словами.* Існує чимало прийомів, які сприяють підвищенню ефективності процедур пошуку документів за ключовими словами. Хоча детальне висвітлення предмета виходить за рамки тексту лекцій, однак два корисних методи подамо.

1. *Виділення кореневої основи слів*. Перед включенням в індекс кожне слово піддається обробці, яка передбачає вилучення префіксів, суфіксів і закінчень, а також

виконання інших необхідних дій. Наприклад, іменники у множині приводять до однини (цей прийом використано у структурі, зображеній на рис. 56, оскільки процес пошуку за ключовим словом «dog» приводить до отримання документів, які містять не тільки слово «dog», але й відповідний варіант множини – «dogs»).

2. Відкидання *слів-роздільників*. Сполучники, частки, артиклі, прийменники та інші подібні слова, які часто використовуються, і не несуть особливого змістового навантаження і є присутніми практично у будь-якому документі, зі зверненого індексу, зазвичай, вилучаються. Вилучення слів-роздільників не впливає на якість результатів пошуку і дає змогу суттєво зменшити розміри файла індексу і час, необхідний для його перегляду.

Файл сегментів може містити:

1) покажчики на документи загалом;
2) покажчики, які посилаються на екземпляри ключового слова всередині документа; у цьому випадку покажчик може складатися з двох частин – адреси першого блока документа і цілочислового номера слова у документі.

За використання другого варіанта, що передбачає застосування покажчиків, які адресують окремі входження ключового слова в тексті документа, модель файла сегментів може бути розширеною за рахунок додаткової інформації про кожен екземпляр ключового слова всередині документа. Тепер файл сегментів сам по собі стає колекцією записів достатньо складної структури. Перші реалізації подібної моделі давали змогу розрізняти зразки ключового слова, присутні у назві, анотації і тілі документа. З розвитком технологій опису Web-документів засобами HTML, XML та інших мов розмітки стало можливим задавати і різні ознаки формування ключових слів, які необхідно відшукати. Тепер система здатна зовсім незалежно розрізняти слова, які присутні в назвах, заголовках, таблицях, фрагментах «звичайного» тексту або рядках

гіперпосилань, бо вони набрані за допомогою певних шрифтів або відформатовані з використанням тих або інших ознак.

Приклад 54. На рис. 57 зображено файл сегментів, який дає змогу знаходити екземпляри слів у документах формату HTML. Комірка першого стовпця описує тип слова – приналежність екземпляра слова певній складовій частині документа, ознаки форматування слова і т.п. Значення у другому і третьому стовпцях у сукупності утворюють покажчик на екземпляр слова в документі: третя комірка містить посилання на документ, а друга представляє номер слова в документі.

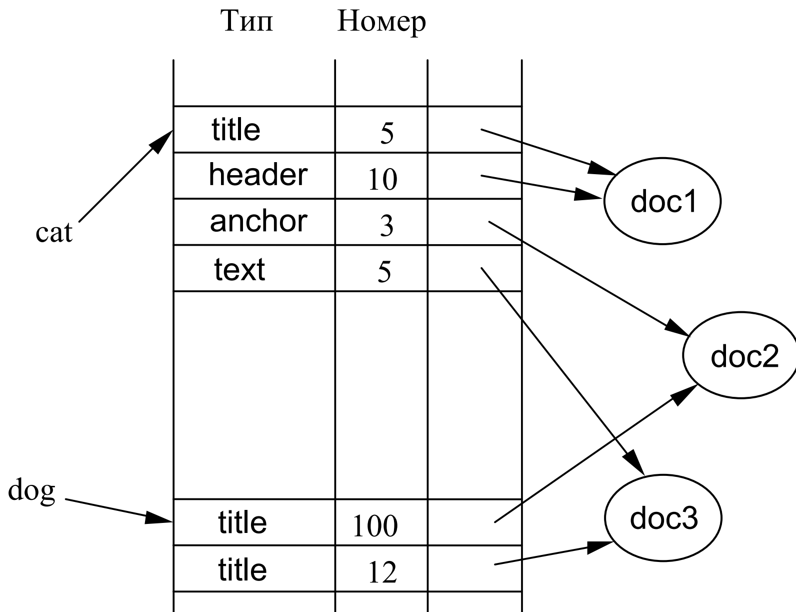


Рис. 57. Звернений індекс з додатковою інформацією про екземпляри ключових слів

Подібна структура даних дає змогу отримувати відповіді на різні запити, що стосуються змісту документів, не потребуючи детального вивчення кожного документа. Припустимо, наприклад,

що необхідно відшукати документи, які присвячено собакам і містять згадки про котів. Якщо не вчитуватися у кожний документ детально, то отримання чіткої відповіді на подібне запитання доволі проблематичне. Загалом, непогану основу для подальшого аналізу не важко забезпечити, якщо спробувати знайти документи, в яких:

а) слово *dog* (*собака*) згадується у назві;
б) слово *cat* (*кіт*) міститься в областях тексту, які слугують гіперпосиланнями (вказують на документи, присвячені котам).

Щоб отримати відповідь на запити, достатньо здійснити операцію перетину множин покажчиків. Покажчик у файлі індексу, який відповідає ключовому слову *cat*, дає змогу визначити групу покажчиків, що адресують екземпляри цього слова в документах. Серед покажчиків групи-сегмента необхідно обрати такі, які належать до типу *anchor* (область тексту гіперпосилання). Потім відшукується група покажчиків, які посилаються на екземпляри слова *dog* у документах, і у ній вибираються покажчики типу *title* (назва). Операція перетину двох множин покажчиків визначить набір посилань на документи, які задовольняють обидві умови: слова *dog* присутні у назвах, а слова *cat* – в областях тексту гіперпосилання.

✓ ***Корисно знати.*** *Файли сегментів: операція вставляння і вилучення елементів.* Структуру файлів сегментів прийнято зображати (як це зроблено, наприклад, на рис. 56) у вигляді масивів елементів. У реальності групи-сегменти складаються із записів з єдиним полем покажчика і зберігаються у блоках, як і будь-які інші колекції записів. Отож у випадку модифікації файлів сегментів, що передбачає вставляння і вилучення покажчиків, допустимо використовувати прийоми, розглянуті вище: попередній розподіл записів по блоках, за якого блоки заповнено не цілковито, що спрощує розширення файла в майбутньому; використання блоків переповнення і переміщення записів усередині блока і між блоками (в останньому ви-

падку необхідно потурбуватися і про відповідну зміну вмісту покажчиків у зверненому індексі, які посилаються на записи файла сегментів, що переміщаються).

Вправи для опрацювання

Вправа 42. Під час здійснення операцій вставки записів у файл даних і вилучення записів відповідного оновлення потребує і файл вторинного індексу. Запропонуйте стратегії підтримки вторинного індексу в актуальному стані за зміни файла даних.

Вправа 43. Розглянемо структуру компактно згрупованого файла, подібну до наведеної на рис. 53, і припустимо, що в один блок поміщаються десять записів даних, які представляють кортежі відношень Movie або Studio. Припустимо також, що кількість фільмів, які відповідають студії, рівномірно розподілено на інтервалі від 1 до *m*. Як виглядає функція від *m*, яка обчислює середнє число операцій дискового введення/виведення, необхідних для отримання інформації про кіностудії і всі фільми, випущені нею? Якою виявиться ця величина, якщо припустити, що розподіл інформації про кінофільми у дискових блоках систематизовано?

Вправа 44. Припустимо, що кожен блок здатен зберігати три записи даних, десять пар виду *«ключ-покажчик»* або п'ятдесят покажчиків і передбачається використання схеми з проміжним рівнем груп-сегментів (див. рис. 54).

1. Якщо кожне значення ключа пошуку міститься в середньому в 10-ти записах, то скільки блоків необхідно для зберігання 3 000 записів даних і відповідної структури вторинного індексу? Скільки блоків необхідно було б виокремити, якби не було файла сегментів?
2. Якщо обмежень на кількість записів даних з певним значенням ключа пошуку немає, то яка мінімальна і максимальна оцінка кількості необхідних блоків?

Вправа 45. Розширений варіант зверненого індексу, подібний до наведеного на рис. 57, дає змогу виконувати запити різних видів. Як можна використовувати цей індекс для відшукання документів, в яких:

а) слова *cat* і *dog* чергуються на п’ятьох сусідніх позиціях текстового фрагмента одного типу, наприклад, *title* (назва), *text* (звичайний текст) або *anchor* (область гіперпосилання);

б) за словом *dog* розташовано довільне слово і слово *cat*;

в) слова *dog* і *cat* є в назві документа.

Тема 15. *B*-дерева

Хоча багаторівневі індекси вважають непоганим інструментом пришвидшення процесів обробки запитів, існує узагальнений, гнучкий та ефективний різновид структур, які найширше застосовують у комерційних СКБД. Йдеться про структури, які називають *B-деревами* (*B-tree*). Найпоширенішим є варіант *B+-дерево* (*B+-tree*).

Особливості *B-дерев* такі:

- автоматична підтримка необхідної кількості рівнів індексації, що відповідає розміру файла даних, який індексується;
- ефективне керування розміром вільних областей усередині блоків (рівень заповнення блоків коливається від 50% до 100%) і відсутність потреби у використанні блоків переповнення.

Нижче ми обговорюватимемо такі характеристики *B*-дерев, які цілком справедливі для *B+*-дерев.

15.1. Структура B-дерева

Як випливає з назви структури *B*-дерева, її блоки організовані у вигляді *деревоподібного графа* (*tree-like graph*). *B*-дерево є *збалансованим* – у тому розумінні, що довжини всіх шляхів від *кореневої* (*root*) вершини до будь-якої з вершин-*листів* (*leaves*) є рівними. Типове *B*-дерево містить три рівні: кореневу вершину, проміжні вершини і листи, – хоча може налічувати і довільну кількість рівнів. Щоб отримати початкове уявлення про *B*-дерева, погляньте на рис. 58 і 59, які описують окремі вершини, і на рис.60, що зображає невелике повне *B*-дерево.

Кожному *B*-деревоподібному індексу відповідає параметр n, який визначає властивості компонування блоків *B*-дерева. Кожен блок має простір, достатній для розміщення n значень ключа

пошуку і $n+1$ покажчиків. Блок дерева насправді подібний до індексних блоків, які розглянуто у темі 13, за винятком того, що поряд з n парами виду *«ключ-покажчик»* він містить додатковий, $n+1$-й, покажчик. Величина n обирається так, щоб забезпечити можливість зберігання у блоці $n+1$ покажчиків і n ключових значень.

Приклад 55. Нехай дисковий блок має розмір 4 096 байтів. Припустимо, що функцію ключів виконують 4-байтові цілочислові значення, а під кожен покажчик відведено по 8 байтів. Якщо вважати, що блоки не містять заголовків, то найбільшим цілим значенням n, яке задовольняє умову $4n+8(n+1) \leq 4\,096$, буде 340.

Існує кілька важливих правил, які регламентують склад вмісту блоків B-дерева.

- Ключові значення, які знаходяться у вершинах-листах, є копіями ключів записів файла даних. Ключі розділені по вершинах-листах зліва направо у порядку зростання значень.
- Коренева вершина містить, щонайменше, два покажчики, які адресують блоки вершини наступного рівня B-дерева[1].
- Останній покажчик вершини-листа посилається на чергову вершину-лист справа, тобто на блок, який містить наступну за порядком порцію ключових значень. Щонайменше $\lfloor (n+1)/2 \rfloor$ з n покажчиків[2] у блоці-листі використовують для посилання на записи даних, а решта вважають вільними та інтерпретують так, наче вони мають значення *null*; i-й покажчик, якщо його використовують, адресує запис з i-м ключем.

---

[1] Можлива ситуація, коли файл даних містить тільки один запис, отож B-дерево складатиметься з єдиної вершини, яка володіє однією парою виду «*ключ-покажчик*» і є водночас і коренем, і листком. Подібний вироджений випадок у подальшому ми ігноруватимемо.

[2] Вирази $\lfloor x \rfloor$ і $\lceil x \rceil$ означають «найбільше ціле, менше від x» (виділення цілої частини x) і «найменше ціле, більше від x» (заокруглення x до найближчого більшого цілого), відповідно.

- Усі n+1 покажчиків у проміжній вершині можна використовувати для посилання на блоки наступного рівня B-дерева, хоча реально повинні використовуватися щонайменше $\lceil(n+1)/2\rceil$ з них (нагадаємо, що, незалежно від величини n, коренева вершина може містити не менше, ніж 2 покажчики, які використовуються за призначенням). Якщо діючими є j покажчиків, то їм відповідають j-1 ключів (наприклад, K_1, K_2, ..., K_{j-1}). Перший покажчик адресує ту частину B-дерева, яка дає змогу відшукати записи з ключовими значеннями, меншими K_1. Другий покажчик посилається на фрагмент дерева, відповідальний за виявлення записів зі значеннями ключа, які більші або дорівнюють K_1 і строго менші від K_2, і т.д. Зрештою, j-й покажчик приводить до гілки дерева, яка відповідає записам з ключовими значеннями, рівними або більшими K_{j-1}. Зауважимо, що записи з ключами, набагато меншими від K_1 і значно більшими від K_{j-1}, з біжучого блока, можливо, і недосяжні – у такому випадку їм відповідатимуть певні сусідні блоки, які належать до того ж рівня дерева.

Приклад 56. У цьому і наступних прикладах, що стосуються B-дерев, ми вважатимемо, що n = 3, тобто кожен блок може містити три ключових значення, які є невеликими (це припущення цілком правдоподібне) цілими числами, і чотири покажчики. На рис. 58 проілюстровано вершину-лист, усі покажчики якої використано за призначенням. Блок вершини містить три ключі – 57, 81 і 95. Перші три покажчики адресують записи даних з зазначеними ключами, а останній (як і в будь-якому листі B-дерева) посилається на сусідній лист справа від біжучого. Якщо біжуча вершина-лист є останньою, то покажчику присвоюється значення *null*.

Ми вже говорили, що допустимо не заповнювати блок-лист до кінця, але в нашому випадку (за умови n = 3) блок повинен містити, як мінімум, дві пари виду *«ключ-покажчик»* (у схемі на рис. 58 ключ 95 і відповідний йому покажчик, зазначений як «До запису з ключем 95» можуть бути відсутніми).

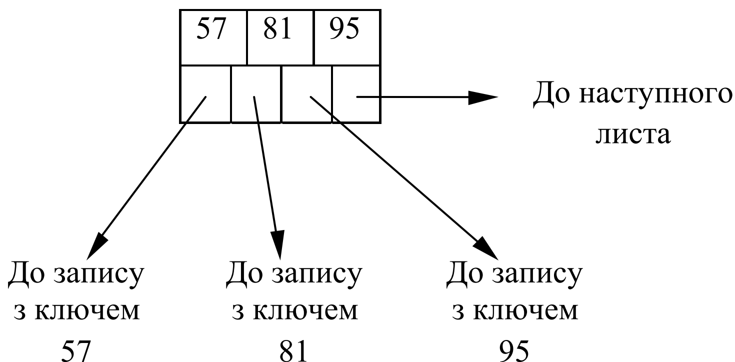


Рис. 58.Схематичне зображення типової вершини-листа *B*-дерева

На рис. 59 зображено звичайну проміжну вершину *B*-дерева. Вона містить три ключі (ми обрали ті ж значення, що і в зразку вершини-листа, проілюстрованому на рис. 58 – 57, 81 і 95)[1] і чотири покажчики. Перший покажчик посилається на гілку дерева, яка дає змогу звертатися до записів з ключами, меншими від 57. Другий покажчик здатний привести до записів з ключовими значеннями з проміжку від 57 (включно) до 81, третій – до записів з ключами з інтервалу від 81 (включно) до 95, а четвертий – до записів, значення ключів яких не менше від 95.

Як і в розглянутому вище прикладі листа *B*-дерева, зовсім не обов'язково, щоб були зайнятими всі «комірки», призначені для зберігання ключів і покажчиків. Однак у випадку $n=3$ проміжна вершина повинна містити щонайменше один ключ і два покажчики. Найекстремальнішим варіантом виявився б такий, коли блок містить ключ 57 і тільки два початкових покажчики: перший з них адресує ключі, менші від 57, а другий – більші або рівні 57.

[1]Не дивлячись на збігання ключових значень, листок на рис. 58 і проміжна вершина на рис. 59 ніяк між собою не пов'язані. Більше того, цілком можливо, що вони ніколи не будуть одночасно представлені в одному і тому ж *B*-дереві.

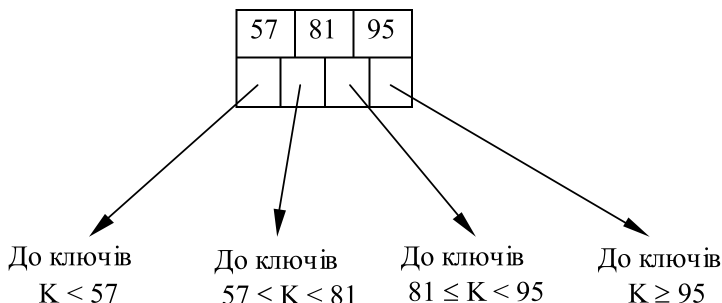


Рис. 59. Приклад проміжної вершини *B*-дерева

Приклад 57. На рис. 60 зображене B-дерево, що має три рівні та містить вершини, зразки яких розглянуто у прикладі 56. Вважають, що файл даних складається з записів, ключові значення яких є простими числами з інтервалу від 2 до 47. Зверніть увагу, що на рівні вершин-листів кожен ключ не повторюється двічі, і ключі впорядковано за зростанням від «лівих» вершин до «правих». Кожен блок-лист містить від двох до трьох пар виду *«ключ-покажчик»*, а також покажчик, який адресує наступний лист послідовності.

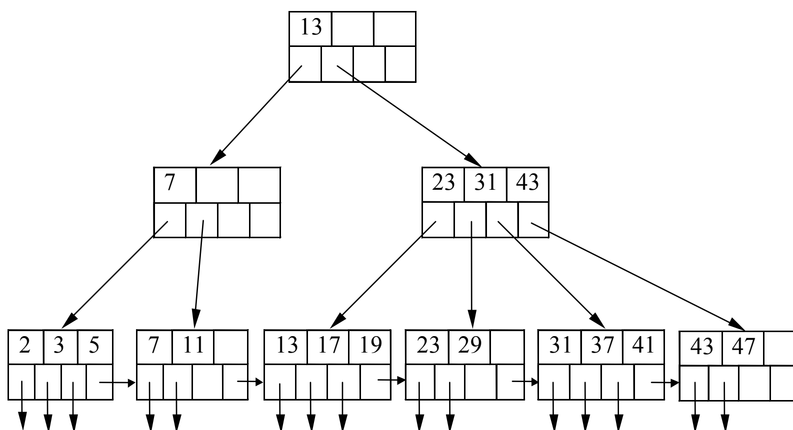


Рис. 60. Приклад *B*-дерева

Коренева вершина містить тільки два покажчики (мінімально можливу кількість), хоча могла б налічувати до чотирьох. Єдиний ключ кореневої вершини ділить множину ключів на дві частини: одна доступна за посередництвом першого покажчика, а друга – другого. Ключі зі значеннями 2-11 (не 2-12, як здається, оскільки за нашою домовленістю ключі є простими числами) можна відшукати в лівому піддереві відносно кореневої вершини, а ключі зі значеннями 13 і вище – у правому.

Перша (ліва) дочірня вершина кореневої вершини, що містить ключ 7, також володіє двома покажчиками, перший з яких посилається на ключі, менші від 7, а другий – на ключі, більші або рівні 7. Зауважимо, що другий покажчик дає змогу звернутися до ключів 7 і 11, а не до всіх ключів, не менших від 7 (наприклад, 13), – решта ключів досяжні із сусідньої проміжної вершини дерева.

Зрештою, друга (права) дочірня вершина містить всі чотири покажчики. Перший адресує підмножину ключів, менших від 23 (а саме – 13, 17 і 19), другий посилається на ключі K, які задовольняють умову $23 \leq K < 31$, третій дає змогу знайти ключі, такі, що $31 \leq K < 43$, а четвертий приводить до підмножини ключів $K \geq 43$.

15.2. Використання В-дерев

Модель *B*-дерева є потужним інструментом конструювання індексів. Послідовність покажчиків на записи, які містяться у вершинах-листах, здатна виконувати ті ж функції, що і будь-яка підмножина покажчиків у «традиційних» індексних файлах, які розглянуто у темах 13 і 14.

Наведемо декілька прикладів.

1. Ключ пошуку в *B*-дереві є первинним ключем файла даних, індекс належить до категорії щільних. Іншими словами, блоки-листи індексу містять по одній парі *«ключ-покажчик»* для кожного запису даних. Файл даних може бути (або може і не бути) посортованим за первинним ключем.
2. Файл даних є посортованим за первинним ключем, і *B*-дерево фігурує як розріджений індекс, листи якого містять по одній парі виду *«ключ-покажчик»* для кожного блока даних.

3. Файл даних є посортованим за значеннями атрибута, який не є первинним ключем, і цей атрибут слугує ключем пошуку в *B*-дереві. Для кожного ключового значення *K*, яке присутнє в записах даних, у деякому блоці-листі існує одна пара виду *«ключ-покажчик»*, і покажчик адресує перший із записів, що володіють значенням *K*.

Варіанти *B*-дерев, що допускають наявність дублікатів значень ключа пошуку[1], також мають широке використання. Приклад подібного *B*-дерева представлено на рис. 61, і він ілюструє розширену модель *B*-дерев, аналогічну до схеми «звичайних» індексів зі значеннями ключа, що повторюються (див. пункт 13.5).

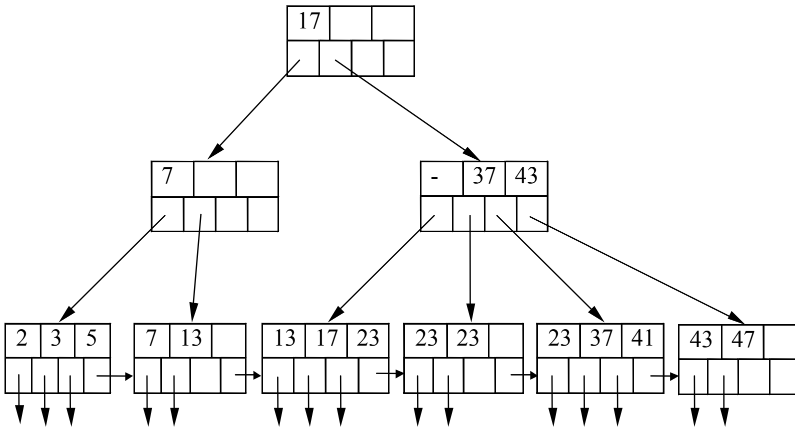


Рис. 61. *B*-дерево з дублікатами значень ключа

Якщо ключ *B*-дерева допускає наявність значень-дублікатів, необхідно трохи переглянути спосіб інтерпретації ключів, які належать проміжним вершинам (ми говорили про це у пункті 15.1). Припустимо, що проміжна вершина містить ключові значення $K_1,K_2,\ldots,K_n$. Тоді K_i буде найменшим новим значенням, присутнім у

[1] Нагадаємо, що термін *ключ пошуку* не завжди засвідчує унікальність ключових значень

тій частині піддерева, яка є доступною за посередництвом *i+1*-го покажчика. Слово «новий» означає, що екземпляри K_i відсутні в частині дерева, розташованій зліва від *i+1*-го піддерева, однак це піддерево містить щонайменше один екземпляр K_i. У деяких випадках такого значення може і не бути, і тоді K_i трактується як *null*. Однак відповідний покажчик є все ще необхідним, оскільки він адресує значну частину дерева, яка виявилась зв'язаною з єдиним ключовим значенням.

Приклад 58. На рис. 61 наведено *B*-дерево, подібне до зображеного на рис. 60, проте воно допускає повторення значень ключа (зокрема, значення 11 замінене на 13, а 19, 29 і 31 – на 23). Як наслідок, ключовим значенням кореневої вершини стає 17 (замість 13). Причина полягає в тому, що хоча 13 і слугує мінімальним ключем правого піддерева, проте це значення не є новим, оскільки вже присутнє в лівому піддереві.

Не залишилась без модифікації і права проміжна вершина. Значення другого ключа змінене з 31 на 37, оскільки саме це число стало найменшим новим ключем у п'ятому зліва листі. Однак найцікавішим є те, що перший ключ правої проміжної вершини набув значення *null*, оскільки четвертий зліва лист не містить нових ключів. Іншими словами, якщо процес пошуку приводить до правої дочірньої вершини кореневої вершини, то продовжувати пошук з четвертого листа не має змісту: якщо потрібно знайти ключ 23 або менший, необхідно так чи інакше звернутися до третього листа. Окрім того, необхідно взяти до уваги такі обставини.

- Якщо необхідно відшукати ключ 13, то з кореневої вершини потрібно звернутися до лівої (а не до правої) дочірньої вершини.
- Якщо передбачено відшукати ключі з проміжку 24-36, то доведеться попрямувати до третього листа, проте за від'ємного результату пошуку звертатися до листів справа, очевидно, не потрібно. Наприклад, якщо у блоках-листах необхідно знайти ключ 24, то його можна було б відшукати у четвертому листі (і в такому випадку ключ *null* правої

проміжної вершини набуде значення 24), або в п'ятому (тоді на 24 треба було б змінити значення 37 проміжної вершини).

15.3. Пошук у В-деревах

Повернемося до попереднього припущення, згідно з яким вершини-листи *B*-дерева не можуть містити значення ключа, які повторюються. Вважатимемо, що *B*-деревоподібний індекс є щільним – кожне ключове значення, присутнє у файлі даних, неодмінно буде представлене у блоках-листах *B*-дерева. Подібні припущення дають змогу значно спростити розгляд особливостей операцій з *B*-деревами і водночас не зачіпають суті цих операцій. У свою чергу, процедури модифікації розріджених *B*-дерев аналогічні до тих, які застосовують щодо «звичайних» індексів для послідовних файлів (див. п. 13.3).

Нехай за допомогою *B*-деревоподібного індексу необхідно знайти запис даних, який володіє значенням *K* ключа пошуку. Процес пошуку виконується рекурсивно – розпочинається з кореневої вершини дерева і завершується після досягнення вершини-листа.

Базис. Якщо досягнута деяка вершина-лист, то переглянути значення ключа, які їй належать. Якщо *i*-те значення дорівнює *K*, то *i*-й покажчик адресує шуканий запис даних.

Індукція. Якщо досягнута деяка проміжна вершина, що містить ключі K_1, K_2,...,K_n., то для визначення чергової дочірньої вершини, до якої треба перейти, необхідно керуватися правилами, викладеними у пункті 15.1. Існує єдина дочірня вершина, з якої вдасться досягнути запису даних з ключем *K*: якщо $K< K_1$, то необхідно перейти до першої дочірньої вершини; якщо $K_1 \le K< K_2$ – до другої і так далі. Продовжити пошук рекурсивно від знайденої вершини.

Приклад 59. Припустимо, що необхідно знайти запис з ключем 40 у структурі *B*-дерева, яку наведено на рис. 60. Процес пошуку розпочнемо з кореневої вершини, яка містить єдиний ключ 13. Оскільки 13≤40, то попрямуємо за вказівкою другого покажчика, який адресує проміжну вершину з ключами 23, 31 і 43. Тут визначаємо, що 31≤40<43, отож скористаємося третім покажчиком,

який приводить до вершини-листа з ключами 31, 37 і 41. Якщо файл даних міститиме запис з ключем 40, то знайдений лист даватиме змогу його відшукати, та оскільки ключ 40 у блоку-листі відсутній, то доведеться зробити висновок, що запису з ключем 40 у файлі даних немає.

Якщо необхідно було б знайти запис з ключем 37, то процес пошуку привів би до того ж листа дерева. Оскільки значення 37 присутнє серед ключів цього листа, то відповідний покажчик (другий) адресує шуканий запис даних.

15.4. Запити у діапазонах значень

B-деревоподібні індекси застосовують не тільки за необхідності відшукання єдиного значення ключа, але й у тих випадках, коли запит передбачає пошук записів, які задовольняють діапазону значень, тобто містить речення *WHERE* з операторами порівняння, відмінними від = («рівне») і <> («нерівне»). Подібний запит до відношення *R* з атрибутом *k*, що виконує роль ключа пошуку, може виглядати, наприклад, так

```
SELECT *
FROM R
WHERE R.k>40;
```

або так

```
SELECT *
FROM R
WHERE R.k >= 10 AND R.k <= 25;
```

Якщо у блоках-листах *B*-дерева необхідно знайти всі ключі, які належать (закритому) діапазону [*a,b*], то доведеться задатися метою відшукати лист з ключем *a*. Незалежно від того, чи існує ключ *a* як такий, чи ні, процес пошуку приведе до листа, в якому цей ключ міг би знаходитися, що даватиме змогу відшукати ключі, не менші від *a* і не більші від *b*. З кожним подібним ключем пов’язаний певний покажчик, що адресує запис даних, ключ якого задовольняє заданому діапазону.

Якщо у біжучому блоці-листі немає ключів, які перевищують *b*, то за допомогою покажчика здійснюється перехід до чергового

листа; тут знову послідовно перевіряється належність кожного ключа встановленому діапазону, і якщо умова задовольняється, то використовується відповідний покажчик, що посилається на шуканий запис даних. Процес повторюється доти, доки не буде знайдено ключ, значення якого перевищує b.

Якщо $b=+\infty$, тобто задано тільки нижню межу діапазону, то переглядаються всі блоки-листи, починаючи від блока, який містить ключ a, і завершуючи останнім блоком у ланцюжку. Якщо ж $a=-\infty$ (задано тільки верхню межу діапазону), то пошук розпочинається з першого листа дерева і продовжується так, як описано вище, тобто до досягнення ключа, значення якого перевищує b.

Приклад 60. Припустимо, що у B-дереві, зображеному на рис. 60, необхідно відшукати ключі, що належать (відкритому) діапазону (10,25). Спроба знайти ключ 10 приводить до другого листа. Перший ключ 7, менший від 10, проте другий, 11, належить заданому діапазону. Покажчик дає змогу відшукати відповідний запис даних і долучити його до підсумкової множини.

Набір ключів другого блока-листа вичерпаний, отож переходимо до третього листа, який містить ключі 13, 17 і 19. Усі вони строго менші від верхньої межі діапазону, рівної 25, що дає підстави звернутися за допомогою покажчиків до відповідних записів і долучити ці записи до підсумкової множини. Зрештою, після переміщення до четвертого листа множина поповниться записом з ключем 23. Черговий ключ листа, однак, володіє значенням 29, яке перевищує 25, отож пошук завершується. Отже, підсумкова множина міститиме записи з ключами 11, 13, 17, 19 і 23.

15.5. Вставляння елементів у B-дерево

Одна з переваг моделі B-дерев порівняно з простішою схемою багаторівневих індексів, розглянутою у пункті 13.4, пов'язана зі спрощенням операцій вставляння нових ключових елементів. Вставляння записів у файл даних, індексований з використанням B-дерева, здійснюється за допомогою будь-якого з методів, згаданих у темі 13. Зосередимо увагу на тому, яким змінам у відповідь на вставляння запису даних піддається B-деревоподібний індекс. Процес за своїм характером є рекурсивним.

- Відшукати у відповідному блоці-листі вільне місце, придатне для розташування нової пари виду *«ключ-покажчик»*, і вставити пару, якщо таке місце існує.
- Якщо вільне місце в листі відсутнє, то розділити лист на два листи і розділити між ними вихідну множину ключів і покажчиків порівно (або, якщо кількість ключів є непарною, то помістити в один блок на одну пару виду *«ключ-покажчик»* менше або більше).
- Розділення вершини дерева на одному рівні спричинює до необхідності вставляння нової пари виду *«ключ-покажчик»* у відповідну вершину наступного вищого рівня. З цією метою використовують таку ж стратегію: якщо місця досить, то вставити пару; у протилежному випадку розділити вершину на дві і продовжити процес вгору по дереву.
- У випадку, коли при спробі вставляння нової пари виду *«ключ-покажчик»* у кореневу вершину виявиться, що вільного місця немає, її розділяють на дві проміжні і створюють нову кореневу вершина. Нагадаємо, що, незалежно від кількості n «комірок» для зберігання ключів у межах вершини, коренева вершина може містити мінімум один ключ і два покажчики.

Випадок, пов'язаний з розділенням вершини і вставлянням нової пари виду *«ключ-покажчик»* у батьківську вершину, заслуговує особливої уваги. Припустимо, що здійснюється спроба вставляння $n+1$-ої пари виду *«ключ-покажчик»* у вершину-лист N, максимальна кількість ключів якої дорівнює n. Створюється нова сусідня вершина-лист M, яка розташована справа від N і має цю ж батьківську вершину, що й N. Перші $\lceil (n+1)/2 \rceil$ пар виду *«ключ-покажчик»* зберігаються у вершині N, решта переміщається в M. Порядок розташування ключів у N і M зберігається. Обидві вершини, N і M, отримують достатню кількість пар виду *«ключ-покажчик»*, не меншу від $\lfloor (n+1)/2 \rfloor$.

Тепер припустимо, що N – проміжна вершина, яка містить n ключів і $n+1$ покажчик, і внаслідок розділення дочірньої вершини здійснено спробу додавання в N нової пари виду *«ключ-покажчик»*. Необхідно виконати такі дії.

1. Створити нову вершину M того ж рівня, що і N, яка розташована безпосередньо справа від N і має цю ж батьківську вершину.
2. Зберегти в N $\lceil (n+2)/2 \rceil$ перших покажчиків і перенести в M решту $\lfloor (n+2)/2 \rfloor$ покажчиків, зберігаючи порядок їхнього розташування.
3. Зберегти в N $\lceil n/2 \rceil$ перших ключів і перенести в M решту $\lfloor n/2 \rfloor$ ключів. Один із ключів K, розташований посередині послідовності, завжди виявляється зайвим і не присвоюється ні N, ні M. Ключ K є найменшим ключем, доступним з першої вершини, дочірньої для M. Хоча K явно не належить ні до N, ні до M, він асоціюється з M (у тому розумінні, що представляє найменший ключ, доступний з M) і використовується у вершині, батьківській щодо N і M, як критерій поділу біжучої гілки дерева на піддерева, які відповідають вершинам N і M.

Приклад 61. Розглянемо процес вставляння ключа 40 у B-дерево, наведене на рис. 60. Для відшукання блока-листка, що підходить для вставки нового ключа, використовують процедуру, розглянуту у пункті 15.3. Як проілюстровано у прикладі 59, модифікації підлягає п’ятий блок-листок. Оскільки $n=3$, а лист повинен містити вже чотири пари виду *«ключ-покажчик»* зі значеннями ключів 31, 37, 40 і 41, то його необхідно розділити. Перша фаза (рис. 62) полягає у створенні нового листа і перенесенні у нього двох останніх ключів, 40 і 41, з відповідними покажчиками. Хоча на перший погляд може здатися, що новий варіант дерева містить чотири рівні, їх насправді, як і раніше, три – нижній рівень складається із семи вершин-листів, з’єднаних у ланцюг зліва направо.

Тепер проміжну вершину з ключами 23, 31 і 43 необхідно доповнити покажчиком, який адресує новий лист, що містить ключі 40 і 41. Окрім того необхідно, щоб покажчику відповідав ключ 40 – мінімальний ключ, досяжний за допомогою нового листа. На жаль, вершина з ключами 23, 31 і 43, батьківська щодо розділеного листа, заповнена – у ній немає місця для додаткової

пари виду *«ключ-покажчик»*. Отож вона також потребує розділення.

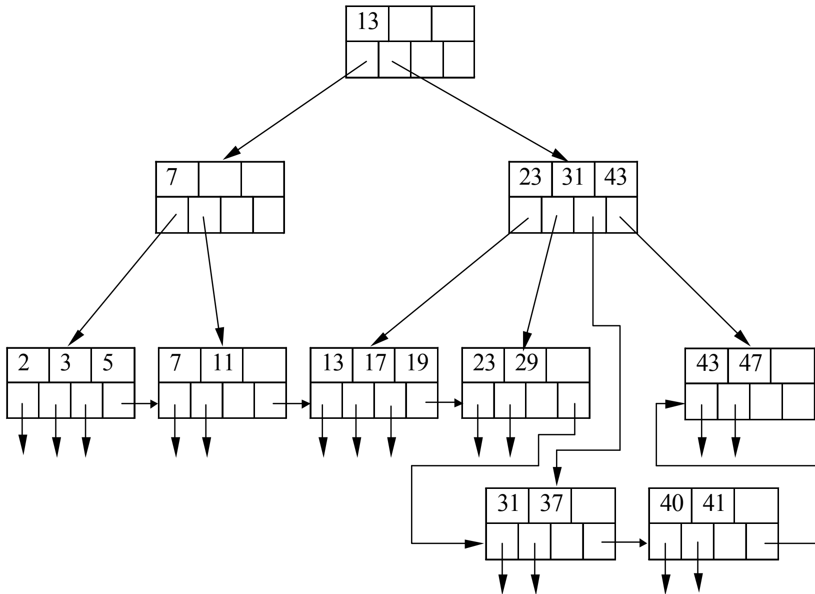


Рис. 62. Результат виконання першої фази процесу вставляння ключа 40 у *B*-дерево, проілюстроване на рис. 60

Розглянемо покажчики, які адресують останніх п’ять листів, і перші ключі – 23, 31, 40 і 43 – чотирьох останніх листів, що містять ці покажчики. Перші три покажчики і перші два ключі залишаються в розділеній проміжній вершині, а останні два покажчики і останній ключ переносять у нову проміжну вершину. Ключ 40, що залишився, є найменшим ключем, доступним за допомогою нової вершини.

Рис. 63 ілюструє другу, заключну, фазу операції вставляння ключа 40. Коренева вершина володіє вже трьома дочірніми вершинами. Дві останні є результатом розділення другої попередньої проміжної вершини. Зверніть увагу, що ключ 40, який є мінімальним ключем, доступним з нової проміжної вершини,

заноситься в кореневу вершину і розділяє «праву» підмножину ключів між другою і третьою проміжними вершинами.

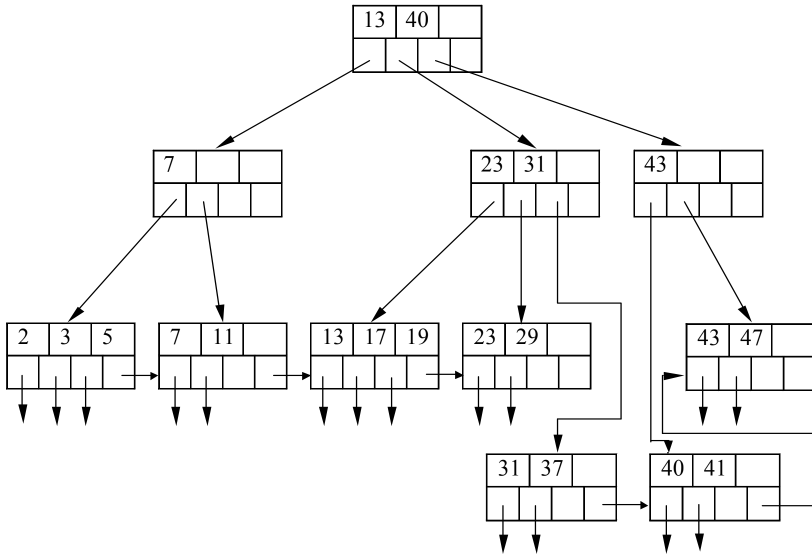


Рис. 63. Остаточний результат процесу вставляння ключа 40 у *B*-дерево, проілюстроване на рис. 60

15.6. Вилучення елементів з В-дерева

За необхідності вилучення запису із заданим ключем *K* передусім необхідно визначити положення цього ключа в одній з вершин-листів *B*-дерева. Ця частина процесу вилучення стосується пошуку, її описано в пункті 15.3. Після відшукання запису даних його видаляють. Потім видаленню підлягає відповідна пара виду *«ключ-покажчик» B*-дерева.

Якщо вершина *N B*-дерева, яка зачіпається процедурою вилучення, все ще містить такий набір пар виду *«ключ-покажчик»*,

кількість яких не є меншою від припустимого мінімуму, то жодних додаткових дій виконувати не потрібно[1].

Однак можлива ситуація, коли вершина *N* зберігає найменшу з можливих кількість пар виду *«ключ-покажчик»*, отож вилучення однієї з них спричинить до порушення обмеження. У подібних випадках необхідно виконати одну із перелічених нижче дій (можливо, з рекурсивним оновленням вершин вверх по дереву).

1. Якщо хоча б одна з двох найближчих до *N* вершин того ж рівня, що мають спільну з *N* батьківську вершину, володіє вищою за мінімальну кількістю ключів і покажчиків, то одну з пар виду *«ключ-покажчик»* такої вершини можна перемістити в *N* зі збереженням порядку розташування ключів. Цілком імовірно, що в такому випадку необхідно змінити значення ключів і у вершині, батьківській щодо *N*. Наприклад, якщо *N* отримує пару виду *«ключ-покажчик»* від сусідньої вершини справа (назвемо її *M*), то ключ цієї пари повинен бути найменшим між усіма ключами *M*. У вершині, батьківській для *N* і *M*, існує ключ, який є мінімальним, доступним з *M*. Значення цього ключа необхідно збільшити. Якщо ж пара виду *«ключ-покажчик»* переміщається в *N* із сусідньої вершини *M* зліва, то ключ цієї пари повинен бути найбільшим між ключами *M*, і значення мінімального ключа у батьківській вершині підлягає зменшенню.
2. Ситуація виявляється серйознішою, якщо жодна з найближчих сусідніх вершин, що належать до тієї ж батьківської вершини, не може надати вершині *N* пару виду *«ключ-покажчик»*, якої не вистачає. Проте у подібному випадку одна з двох суміжних вершин (тобто *N* і одна з дотичних до неї вершин) володіє мінімальною кількістю ключів, а друга містить на

[1]Якщо запис даних, що видаляється, володіє ключем, значення якого є мінімальним для вершини-листа, то є можливість збільшення значення відповідного ключа в одній із батьківських вершин. Хоча робити це не обов’язково – працездатність процедур пошуку зовсім не постраждає

одиницю меншу кількість ключів. Отож обидві вершини разом мають не більше пар виду *«ключ-покажчик»*, ніж дозволено для однієї вершини (ось чому мінімально допустимим обрано половинний рівень заповнення вершин *B*-дерева). Можна здійснити злиття вмісту таких вершин, видаливши одну з них, після чого доведеться виправити значення ключів батьківської вершини і видалити в ній одну пару виду *«ключ-покажчик»*. Якщо рівень заповнення батьківської вершини залишається задовільним, то операція вилучення завершується. У протилежному випадку процес рекурсивним чином повторюється для вершин все вищих рівнів дерева аж до досягнення кореневої вершини.

Приклад 62. Знову звернемося до *B*-дерева, зображеного на рис. 60, і припустимо, що видаленню підлягає запис даних з ключовим значенням 7. Відповідний ключ належить другому листу дерева. Ключ необхідно видалити, а потім видалити відповідний покажчик і запис даних, на який цей покажчик посилається.

На жаль, після вилучення другий лист міститиме тільки одну пару виду *«ключ-покажчик»*, а необхідний мінімум – дві. Проте ситуацію рятує зайвий ключ у лівій суміжній (першій) вершині-листі. Отож найбільший ключ цієї вершини (5) разом з відповідним покажчиком можна перенести у другий лист. Остаточний варіант *B*-дерева представлено на рис. 64. Оскільки найменшим ключем у другому листі тепер є 5, то значення ключа у батьківській вершині змінено з 7 на 5.

Нехай необхідно видалити ще й ключ 11. Операція знову має аналогічну дію на другий лист дерева – кількість пар виду *«ключ-покажчик»* зменшується нижче припустимого мінімуму. Цього разу позичити дані, яких не вистачає, з першого листа не вдасться – він містить гранично малу кількість ключів. Ситуація ускладнюється тим, що листа з правого боку, який належав би до

тієї ж батьківської вершини і до того ж володів би зайвими покажчиками, немає[1].

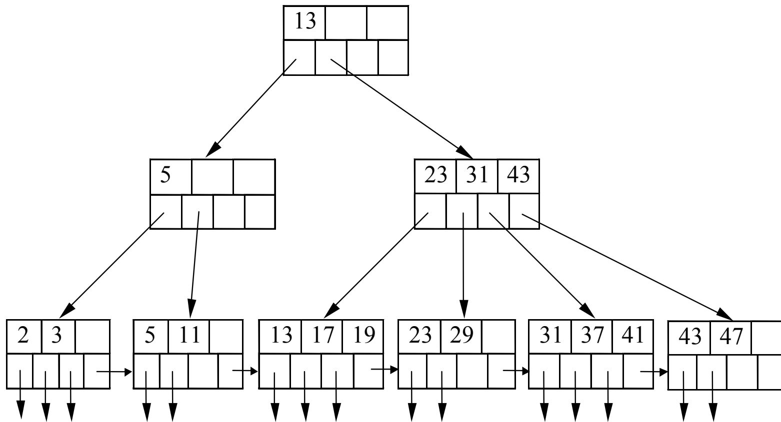


Рис. 64. Результат вилучення ключа 7 з *B*-дерева, проілюстрованого на рис. 60

Три пари виду *«ключ-покажчик»*, які залишилися у перших двох листах, можуть поміститися в одному, отож пару з ключем 5 переносять у перший лист, а другий лист видаляють. Тепер підлягає уточненню і вміст батьківської вершини: два попередніх покажчики заміняють одним (який адресує перший лист), а ключ 5 викидають як такий, що не відповідає дійсності. Цю фазу процесу вилучення проілюстровано на рис. 65.

На жаль, вилучення листа негативно вплинуло на батьківську вершину – ліву дочірню вершину щодо кореневої. Як бачимо з рис. 65, вершина не містить жодного ключа і володіє єдиним покажчиком. Отож спробуємо запозичити необхідну пару виду *«ключ-покажчик»* з сусідньої вершини того ж рівня зі спільною

[1] Зауважимо, що правий (третій) лист, що володіє ключами 13, 17 і 19, має зовсім іншу батьківську вершину. Строго кажучи, можна було б запозичити пару виду «ключ-покажчик» і з цього листа, однак тоді довелося б застосовувати складніший алгоритм виправлення значень ключів у вершинах, «розкиданих» по всьому дереву. Цей варіант ми залишаємо читачу як вправу

батьківською вершиною. У нашому випадку така можливість є: права проміжна вершина може надати лівій свій мінімальний ключ з відповідним покажчиком.

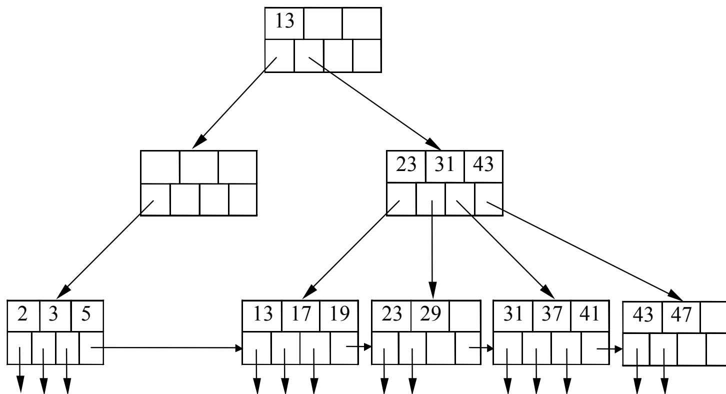


Рис. 65. Результат виконання першої фази процесу вилучення ключа 11 з *B*-дерева, проілюстрованого на рис. 64

Ситуацію, яка склалася, відображено на рис. 66. Покажчик на лист з ключами 13, 17 і 19 перенесено з правої проміжної вершини в ліву.

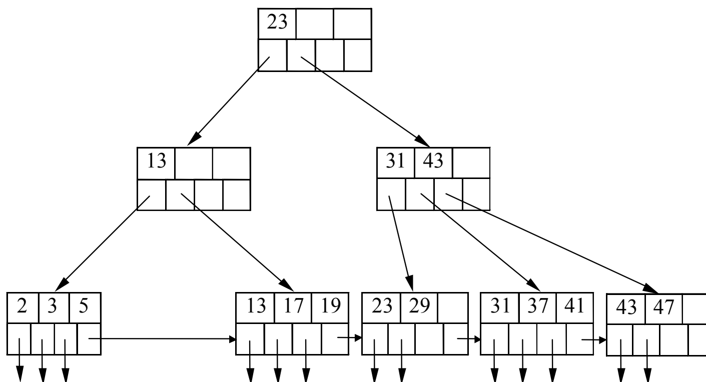


Рис. 66. Остаточний результат виконання процесу вилучення ключа 11 з *B*-дерева, проілюстрованого на рис. 64

Змінились і деякі ключі проміжних і кореневої вершин. Ключ 13, розташований у кореневій вершині, представляв найменше ключове значення, доступне через щойно переміщений покажчик. Тепер він необхідний для першої проміжної вершини. З іншого боку, ключ 23, який розділяв підмножини ключів другої і третьої вершин-листів, у цей момент є мінімальним ключем, доступним у правому піддереві. Отож його перенесено у кореневу вершину.

15.7. Оцінки ефективності B-деревоподібних індексів

Кожна з процедур пошуку, вставки і вилучення записів даних, проіндексованих за допомогою *B*-дерева, вимагає виконання незначної кількості операцій дискового введення/виведення. По-перше, якщо кількість пар виду *«ключ-покажчик»*, які можна розташувати в одному блоці, достатньо велика (наприклад, 10 або більше), то потреба в розділенні або злитті вершин-блоків виникає порівняно рідко. По-друге, навіть у тому випадку, якщо подібні операції є все-таки необхідними, то їхній вплив майже завжди є локальним – зачіпаються тільки два листи і відповідна батьківська вершина. Отож затратами на реорганізацію дерева насправді можна знехтувати.

Однак для виконання будь-якої операції пошуку записів, які володіють заданим ключовим значенням, необхідно відшукати відповідні покажчики на записи, тобто подолати шлях від кореневої вершини до деякого листа. Оскільки блоки *B*-дерева тільки зчитуються, число операцій дискового введення/виведення складається з кількості рівнів дерева та однієї (під час пошуку) або двох (під час вставляння і видалення) операцій, необхідних для маніпуляцій із записом даних як таким. Виникає законне запитання: скільки рівнів містить B-дерево у тому чи іншому випадку? Якщо брати до уваги «типові» розміри ключів, покажчиків і блоків, то трьох рівнів дерева виявляється достатньо для всіх баз даних, окрім особливо крупних. Отже, 3 – це та цифра, якою, зазвичай, характеризують кількість рівнів *B*-дерев. Пояснимо це на прикладі.

Приклад 63. Нагадаємо результати аналізу, здійсненого у прикладі 55. Встановлено, що один блок «звичайного» розміру (4 096 байтів) може містити 340 пар виду *«ключ-покажчик»*.

Припустимо, що значення рівня заповнення блока визначається серединою проміжку між мінімальною і максимальною межами (340/2+340/4), тобто блок містить 255 покажчиків. Отже, коренева вершина адресує 255 проміжних вершин, які разом посилаються на $255^2 = 65\,025$ листів, а кожен лист вказує на 255 записів, що дає змогу звертатися до 255^3, або 16,6 мільйонів записів. Іншими словами, якщо кількість записів не перевищує 16,6 млн, то для індексації такого файла достатньо 3-рівневого *B*-дерева.

Однак на практиці кількість операцій дискового введення/виведення, необхідних для пересування деревом, може бути навіть меншою від трьох. Кореневий блок *B*-дерева – прекрасний кандидат для постійного розміщення в буфері ОП. Таке рішення дає змогу скоротити кількість звернень до диска в процесі перегляду дерева до двох. За деяких обставин є сенс буферизувати в пам'яті навіть блоки вершин другого рівня *B*-дерева, що даватиме змогу зменшити кількість операцій дискового введення/виведення ще на одну одиницю.

✓ ***Корисно знати.*** *Чи необхідно видаляти вершини B-дерева?* Існують такі реалізації моделі *B*-деревоподібних індексів, у яких операції вилучення елементів не передбачено зовсім. Якщо лист дерева містить всього декілька пар виду «*ключ-покажчик*», то дозволено залишати все без змін. Аргументом є те, що більшість файлів даних з часом розширюються, і якщо в один момент операція вилучення приводить до зменшення кількості ключів нижче встановленого мінімуму, то в інший імовірна операція вставки відновлює порушений баланс.

Окрім того, якщо запис даних адресується не тільки з вершини *B*-дерева, але й ззовні, то при видаленні такого запису на його місці, зазвичай, необхідно залишити ознаку-«обеліск» (див. п. 12.2). У цьому випадку про вилучення покажчика, який посилається на подібний запис, не йдеться. У деяких ситуаціях, коли гарантується, що всі можливі звернення до видаленого запису виконуються тільки при

посередництві *B*-деревоподібного індексу, то ознаку-«обеліск» можна перенести з запису даних на місце покажчика у відповідному листі дерева, що даватиме змогу повторно використовувати область запису даних.

Вправи для опрацювання

Вправа 46. Припустимо, що дисковий блок може зберігати 10 записів даних або 99 ключів і 100 покажчиків. Припустимо також, що блок вершини *B*-дерева заповнений у середньому на 70%, тобто містить тільки 69 ключів і 70 покажчиків. *B*-дерева можна використовувати як складову частину різних структур. Вважаючи, що спочатку жодної інформації в оперативну пам'ять не завантажували і ключ пошуку є водночас і первинним ключем записів даних, для кожної з описаних нижче структур необхідно визначити (і) загальну кількість блоків, необхідних для зберігання файлу з 1 000 000 записів, і (іі) середню кількість операцій дискового введення/виведення, необхідних для отримання запису, який містить заданий ключ пошуку.

1. Дані розміщені в послідовному файлі, посортованому за ключем пошуку, і кожен блок складається з 10-ти записів. B-деревоподібний індекс є щільним.
2. Те ж, що і в п.1, однак записи файла даних не впорядковані.
3. Те ж, що і в п.1, однак *B*-дерево є розрідженим.
4. Листи *B*-дерева містять не покажчики на записи даних, а самі записи. Блок здатен помістити 10 записів, однак блок-лист заповнено у середньому на 70%, тобто зберігає 7 записів.
5. Файл даних є послідовним, а *B*-дерево – розрідженим, однак кожному основному блоку даних відповідає блок переповнення. Кожен основний блок, здебільшого, заповнено, а область блока переповнення використано на 50%. Записи в основних блоках і блоках переповнення не впорядковано.

Вправа 47. Виконайте завдання з вправи 46 з припущенням, що діапазон запиту охоплює 1 000 записів.

Вправа 48. Припустимо, що розмір покажчика становить 4 байти, довжина ключа – 12 байтів. Скільки пар виду *«ключ-покажчик»* вдасться «запакувати» у блок об'ємом 16 384 байти?

Вправа 49. Які мінімальні кількості ключів і покажчиків, які можуть зберігатися в (i) проміжних вершинах і (ii) листах B-дерева, якщо:

а) n=10, тобто блок може містити 10 ключів і 11 покажчиків;

б) n=11 (максимальні кількості ключів і покажчиків у блоці дорівнюють 11 і 12, відповідно).

Вправа 50. Виконайте перелічені нижче операції стосовно структури *B*-дерева, наведеної на рис. 60. Опишіть зміни, які необхідно внести в структуру під час здійснення операцій модифікації.

1. Пошук запису з ключем 41.
2. Пошук запису з ключем 40.
3. Пошук усіх записів зі значеннями ключа в діапазоні 20-30.
4. Пошук усіх записів зі значеннями ключа, меншими від 30.
5. Пошук усіх записів зі значеннями ключа, більшими від 30.
6. Вставляння запису з ключем 1.
7. Вставляння записів із значеннями ключа в діапазоні 14-16.
8. Вилучення запису з ключем 23.
9. Вилучення усіх записів з ключами, не меншими від 23.

Тема 16. Геш-таблиці

Існує чимало корисних структур індексів, які реалізують модель *геш-таблиць* (*hash tables*). Вважатимемо, що читач знайомий з варіантами використання геш-таблиць як засобу організації даних в ОП. Кожній подібній структурі відповідає деяка *геш-функція* (*hash function*), яка отримує як параметр значення ключа пошуку (у даному випадку варто називати його *геш-ключем – hash key*) і обчислює число з інтервалу від *0* до *B-1*, де *B* – кількість *сегментів*

(*buckets*). Елементи *масиву сегментів* (*bucket array*) проіндексовані від *0* до *B-1* і містять заголовки зв'язаних списків (їхня кількість дорівнює *B*), по одному на кожен елемент-сегмент масиву. Якщо запис володіє значенням *K* ключа пошуку, то він приєднується до списку сегмента з номером *h(K)*, де *h* – геш-функція.

16.1. Геш-таблиці для даних у вторинних сховищах

Геш-таблиця, яка містить настільки велику кількість записів, що їх доводиться розташовувати, здебільшого, у вторинному сховищі, відрізняється від варіанта геш-таблиці, призначеної для структурування даних в ОП, декількома важливими аспектами. Передусім, масив сегментів повинен складатися з блоків, а не окремих покажчиків на заголовки зв'язаних списків. Змішані геш-функцією записи належать до певних сегментів і займають місце у блоках, що відповідають цим сегментам. Якщо сегмент переповнюється (у тому розумінні, що у його блок водночас не поміщаються всі записи, які належать сегментові), то до основного блока сегмента приєднується ланцюжок блоків переповнення, які дають змогу зберегти додаткові записи.

Вважатимемо, що місце розташування основного блока *i*-го сегмента можна відшукати безпосередньо за номером *i*. Наприклад, припустимо зберігати в ОП масив покажчиків на блоки сегментів, який проіндексовано за номерами сегментів. Альтернативний спосіб пов'язаний з розміщенням основних блоків усіх сегментів у фіксованих послідовних позиціях диска.

✓ ***Корисно знати.*** *Як обирати геш-функцію?* Геш-функція покликана «перемішувати» ключі так, щоб цілочислове значення, яке вона повертає, було близьким до випадкової величини, що залежить від ключа-аргументу. Подібна функція має тенденцію до рівномірного розподілу записів по сегментах таблиці, що (як проілюстровано у пункті 16.4) спричинює до зниження часових затрат на доступ до записів. Окрім того, геш-функція повинна легко обчислюватися, оскільки користуватися нею доводиться дуже часто.

- Звичайний варіант геш-функції, ключі-аргументи якої є цілими числами, передбачає обчислення залишку від ділення *K/B*, де *K* – значення ключа, а *B* – кількість сегментів. Здебільшого, значенням *B* слугує одне з простих чисел, хоча існують докази на користь прирівнювання *B* до степені числа 2 (за деталями звертайтесь до пункту 16.5).
- Якщо ключами слугують рядки символів, то кожен із символів трактується як ціле число; усі числа, які відповідають символам рядка, додають, а потім обчислюють залишок від ділення отриманої суми на кількість сегментів *B*.

Приклад 64. На рис. 67 наведено зразок геш-таблиці. Щоб спростити подальше розмірковування, вважатимемо, що кожен блок може зберігати не більше двох записів і *B=4*, тобто геш-функція *h* повертає значення з інтервалу 0-3. Таблиця містить декілька записів з ключами, позначеними літерами від *a* до *f*. Вважають, що *h(d)*=0, *h(c)*=*h(e)*=1, *h(b)*=2 і *h(a)*=*h(f)*=3. Отож шість записів розподілено у блоках саме так, як проілюстровано на рис. 67.

Зверніть увагу, що на рис. 67 кожен блок позначено з правого боку додатковим прямокутним виступом, що символічно представляє те місце в заголовку блока, де, наприклад, можна розташувати покажчик на відповідний блок переповнення. У пункті 16.5 і далі використовуватимемо подібні «виступи» і для представлення іншої важливої інформації щодо блока.

16.2. Вставляння записів у геш-таблицю

За необхідності вставляння у геш-таблицю запису зі значенням *K* ключа пошуку необхідно обчислити геш-функцію *h(K)*. Якщо у блоці сегмента з номером *h(K)* є вільний простір, достатній для розміщення запису, то запис вставляється у блок цього сегмента. Якщо основний блок заповнено, то запис зберігається в одному з відповідних блоків переповнення. Якщо ж ні основний блок, ні

будь-який із блоків переповнення немає місця, то до ланцюжка блоків переповнення приєднують новий блок.

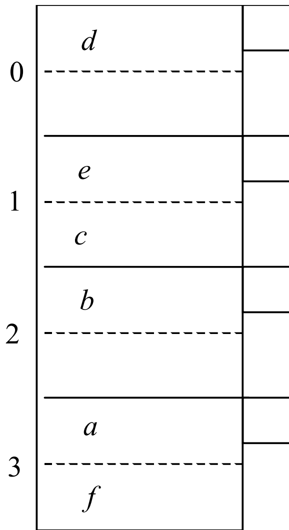


Рис. 67. Приклад геш-таблиці

Приклад 65. Нехай у геш-таблицю, представлену на рис. 67, необхідно вставити запис з ключем g і $h(g)=1$. Запис підлягає розміщенню у блоці сегмента з номером 1. Блок, однак, вже заповнено. Необхідно створити новий блок переповнення і з'єднати його з основним блоком сегмента 1. Запис з ключем g розташовують у щойно доданому блоці, як проілюстровано на рис. 68.

16.3. Вилучення записів з геш-таблиці

Процедура вилучення запису або декількох записів з ключем K виконується за цілком передбачуваною схемою: необхідно перейти до сегмента з номером $h(K)$, відшукати в його блоках записи зі значеннями ключа пошуку K і видалити їх. Якщо є можливість переміщення записів між блоками, то після вилучення записів

можна консолідувати записи, які залишилися, у меншій множині блоків[1].

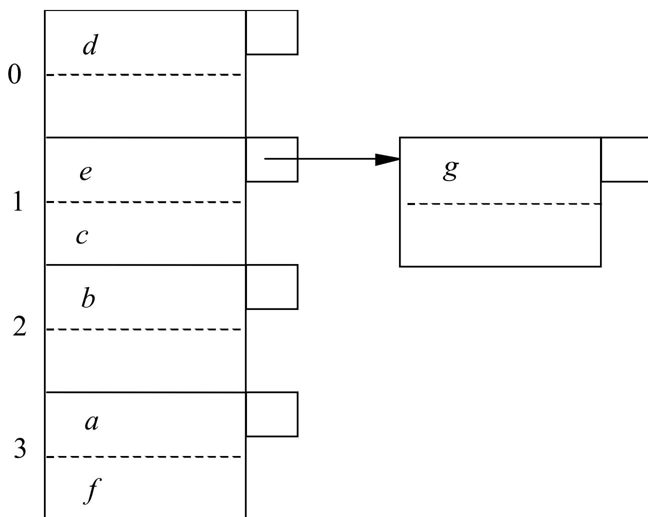


Рис. 68. Результат вставляння запису з ключем g у геш-таблицю, проілюстровану на рис. 67

Приклад 66. На рис. 69 відображено результат вилучення запису з ключем *c* з геш-таблиці, яку наведено на рис. 68. Нагадаємо, що $h(c)=1$, отож необхідно звернутися до сегмента з номером 1 і переглянути всі блоки, які йому належать, у пошуках запису (або декількох записів, якщо ключ пошуку не є водночас первинним ключем) з ключем *c*. Запис розташований в основному блоці сегмента 1. Після його вилучення звільняється область блока, достатня для розміщення запису з ключем *g*, який перебуває у блоці переповнення. Отож при бажанні запис можна перенести, а блок переповнення – вилучити (рис. 69 відповідає саме такому варіанту дій). Із геш-таблиці видалений також запис з ключем *a*,

[1] Недолік такого рішення проявляється у тих випадках, коли операції вставляння і вилучення записів, що чергуються, приводять до необхідності почергового виконання трудомістких процедур створення нових і знищення зайвих блоків

що дає змогу перенести запис з ключем *f* з кінця блока сегмента 3 на його початок.

16.4. Оцінка ефективності гешованих індексів

В ідеальному випадку вмістиме кожного сегмента геш-таблиці заповнює лише один блок, і тоді процедура пошуку запису потребує виконання єдиної операції дискового введення/виведення, а процедура вставки/вилучення записів – двох операцій. Ця оцінка значно перевищує показники ефективності, якими володіють «звичайні» розріджені/щільні або *B*-деревоподібні індекси, хоча останні підтримують можливість «швидкої» обробки запитів у діапазонах значень (див. пункт 15.4), а традиційні варіанти геш-таблиць – ні.

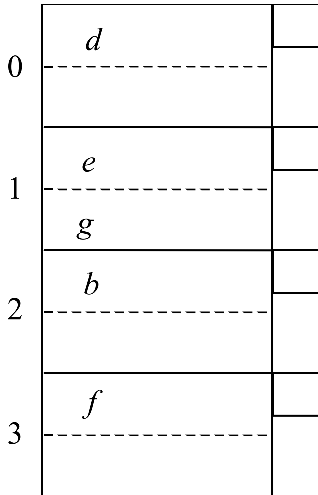


Рис. 69. Результат вилучення записів з ключами *e* та *a* з геш-таблиці, проілюстрованої на рис. 68

Проте внаслідок зростання розмірів файла неминуче виникає ситуація, коли для зберігання вмісту типового сегмента доводиться використовувати достатньо довгий ланцюг блоків. У цьому випадку в процесі пошуку необхідно здійснювати по одній операції

дискового введення/виведення для звертання до кожного з блоків ланцюга. Отож необхідно наближатися до того, щоб співвідношення кількості блоків у розрахунку на один сегмент залишалось настільки гнучким, наскільки це можливо.

До цього часу йшлося про статичні геш-таблиці, в яких значення параметра B (кількість сегментів) залишається незмінним. Однак активне застосування мають і різні варіанти структур динамічних геш-таблиць, де значення параметра B дозволено варіювати співрозмірно частці від ділення загальної кількості записів на найбільше число записів в одному блоці, тобто з метою забезпечення можливості розміщення вмісту сегмента в межах єдиного блока. Нижче ми зосередимо увагу на двох подібних моделях: у пункті 16.5 розглянуто способи розширюваного гешування, а пункт 16.7 присвячено схемам лінійного гешування. Обидві моделі передбачають збільшення значення параметра B зі зростанням кількості записів: у першому випадку значення B за необхідності подвоюється, а в другому – збільшується на одиницю.

16.5. Геш-таблиці, що розширюються

Перший варіант динамічного гешування, який ми розглянемо, стосується використання *геш-таблиць, що розширюються* (*extensible hash tables*). Їхні головні відмінності від моделі статичних геш-таблиць перелічено нижче.

1. Вводиться новий рівень опосередкування: сегмент представляється масивом покажчиків на блоки, а не списком блоків як таких.
2. Масив покажчиків здатен розширюватися. Його довжина завжди дорівнює степені числа 2, отож на наступному кроці збільшення масиву кількість сегментів таблиці подвоюється.
3. Кожному сегментові не обов'язково відповідає окремий блок: якщо сумарна кількість записів у декількох сегментах не перевищує припустимого числа записів у блоці, то сегменти можуть володіти блоком спільно.
4. Геш-функція h обчислює для кожного ключа послідовність із k бітів, де k – задане достатньо велике ціле число (наприклад, 32). Однак для номерів

сегментів використовуватиметься деяка менша кількість бітів (наприклад, i), отриманих з початку послідовності. Отже, довжина масиву сегментів становитиме 2^i для будь-якого $i \le k$.

Приклад 67. На рис. 70 представлено невелику геш-таблицю, що розширюється. Для спрощення вважатимемо $k=4$: геш-функція повертає бітові послідовності довжиною 4. Припустимо, що в даний момент для позначення номерів сегментів використано тільки один біт (як демонструє позначка $i=1$, розташована над масивом сегментів), тобто масив складається всього з двох елементів.

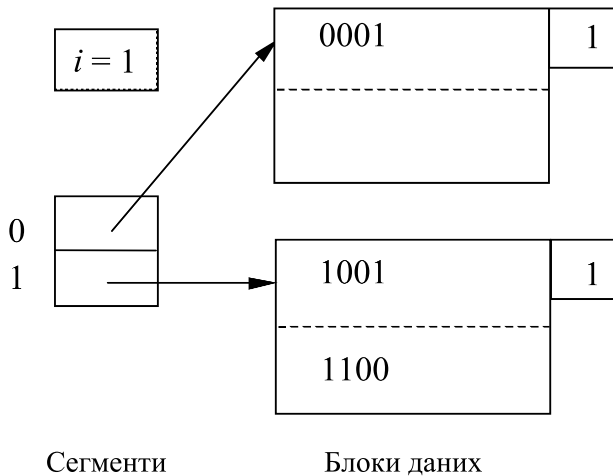


Рис. 70. Приклад геш-таблиці, що розширюється

Елементи масиву сегментів посилаються на два блоки. Перший містить усі наявні в певний момент записи, ключі пошуку яких гешовані в бітові послідовності з нулем у першій позиції, а другий – записи з бітовими геш-кодами, що розпочинаються з одиниці. Для наочності ключі представлено повними послідовностями бітів, які повертаються геш-функцією: верхній блок містить один запис з

ключем пошуку, гешованим у 0001, а нижній – два записи з геш-кодами 1001 і 1100.

Необхідно звернути увагу на одиниці, поставлені в прямокутних «виступах», якими позначено кожен блок на рис. 70. Це число, яке могло б дійсно бути присутнім у заголовку блока, вказує на те, скільки бітів послідовності, що повертається геш-функцією, використовується для визначення приналежності запису біжучому блокові. У ситуації, розглянутій у прикладі 67, для всіх блоків і записів береться до уваги тільки один біт, однак, як буде проілюстровано нижче, зі збільшенням об’єму таблиці це значення для різних блоків може змінюватись. Іншими словами, розмір масиву сегментів визначається максимальною кількістю бітів, які використано в даний момент (для деяких блоків, можливо, необхідна менша кількість бітів).

16.6.Вставляння записів у геш-таблицю, що розширюється

Початкові фази процесів вставляння записів у геш-таблицю, що розширюється, і статичну геш-таблицю за суттю збігаються. Щоб здійснити вставку запису з ключем K, необхідно обчислити значення функції $h(K)$, виокремити з отриманої послідовності перших i бітів і звернутися до елемента масиву сегментів, що володіють індексом, який дорівнює значенню цих бітів. Конкретну величину i визначити легко, оскільки вона зберігається як невід’ємна частина структури геш-таблиці.

Покажчик, який зберігається у відшуканому елементі масиву сегментів, посилається на деякий блок B. Якщо блок містить вільну область достатньої довжини, то запис вставляється і процес успішно завершується. Якщо ж такої області у блоці B немає, то існують дві альтернативи подальших дій, і вибір однієї з них залежить від величини j, що визначає, яку кількість бітів послідовності, поверненої геш-функцією, використано для визначення приналежності запису блока B (нагадаємо, що саме це значення вказане в прямокутних «виступах», якими позначено кожен блок на рис. 70-72).

1. Якщо $j<i$, то масив сегментів не потребує модифікації. Достатньо виконати наступне:
   а) розділити блок B на два блоки;

б) розподілити записи, які належать блокові *B*, у два блоки, керуючись значеннями *j*+1–го біта геш-коду; записи з нулем у зазначеній позиції геш-коду залишати у блоці *B*, а решта перемістити в новий блок;
в) замінити значення у «виступах» кожного з двох отриманих блоків на *j*+1;
г) виправити значення покажчиків в елементах масиву сегментів так, щоб покажчики, які раніше адресували блок *B*, тепер посилалися на *B* або на новий блок – залежно від значення *j*+1-го біта геш-коду.

Зауважимо, що одноразове розділення блока *B*, можливо, не вирішить проблему, оскільки цілком імовірно, що всі записи з блока *B*, цілковито перемістяться в один із нових блоків, на які *B* розділено. У цьому випадку необхідно повторити процес розділення переповненого блока, знову збільшивши *j* на одиницю.

2. Якщо ж $j=i$, то збільшенню на одиницю підлягає значення i. Під час цього довжина масиву сегментів подвоюється і стає рівною 2^{i+1}. Припустимо, що *w* – послідовність із i бітів, яка є індексом одного з елементів попереднього масиву сегментів. У новому масиві елементи з індексами *w*0 і *w*1 (числами, отриманими з *w* додаванням 0 і 1) адресують той самий блок, на який посилався покажчик елемента з індексом *w*. Іншими словами, два нових елементи володіють одним блоком спільно, а блок як такий змінам не піддається. Належність записів цьому блокові все ще можна визначити за допомогою послідовності з тією ж кількістю бітів, що і раніше. Потім блок *B* необхідно розділити, як і у випадку, описаному в п.1 (тепер умова $j<i$ виконується).

Приклад 68. Нехай у геш-таблицю, представлену на рис.70, необхідно вставити запис з ключем, що гешується в бітову

послідовність 1010. Оскільки значення першого біта послідовності дорівнює 1, то запис повинен належати другому (нижньому) блокові. Блок, на жаль, вже заповнений, отож потребує розділення. У нашому випадку $j=i=1$. Отже, спочатку необхідно подвоїти розмір масиву сегментів, як проілюстровано на рис. 71 (тут же є присутньою позначка $i=2$).

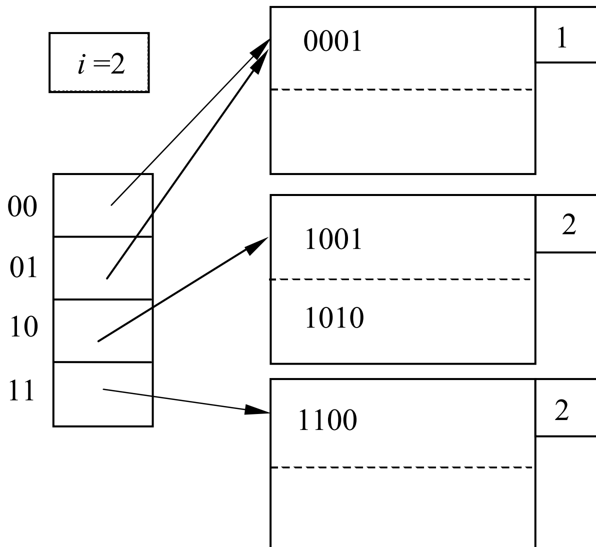


Рис. 71. Результат вставки запису з геш-кодом 1010 у геш-таблицю, проілюстровану на рис. 70

Зауважимо, що два елементи масиву з нулями в першій позиції індексів вказують на блок із записами, геш-коди яких розпочинаються з нуля, і блок у своєму «виступі» все ще містить одиницю, яка засвідчує, що для визначення належності запису до блока достатньо одного першого біта геш-коду. Однак блок із записами, геш-коди яких розпочинаються з одиниці, підлягає розділенню, отож множина записів розбивається на дві підмножини: одні записи мають геш-коди з префіксами 10, а геш-коди інших записів розпочинаються з пари бітів 11. Позначка 2 у «виступі» кожного з двох блоків засвідчує, що для встановлення

факту членства запису у блоці використано два біти геш-коду. На щастя, результат розділення виявився успішним: оскільки кожен із двох нових блоків отримав по одному запису, то необхідності подальшого рекурсивного розщеплення вдалось уникнути.

Тепер припустимо, що необхідно поповнити таблицю записами з ключами, які гешуються в бітові послідовності 0000 і 0111. Обидва записи необхідно розташувати в першому (верхньому) блоці, що спричинює до його переповнення. Оскільки в ролі ознаки належності записів цьому блокові використано тільки один біт при *i*=2, то змінювати організацію масиву сегментів не потрібно.

Достатньо просто розділити блок, залишаючи в його вихідній частині записи з геш-кодами 0000 і 0001 і переміщаючи в новий блок запис з геш-кодом 0111. Покажчик «01»-го елемента масиву сегментів отримує в ролі значення адресу нового блока. І знову нам пощастило: записи не «перейшли» в один із нових блоків усі разом, отож потреби в подальшому рекурсивному розділенні блоків немає.

Розглянемо процедуру вставки запису з геш-кодом 1000. Блок, на який вказує «10»-й елемент масиву сегментів, переповнюється. Оскільки факт членства записів у цьому блоці визначається двома бітами геш-коду, однак *i*=2, то доведеться збільшити значення параметра *i* (*i*=3) і подвоїти місткість масиву сегментів. Результат наведено на рис. 72. Блок для записів з геш-кодами, які розпочинаються з 10, розділено на блоки 100 і 101, а належність записів решти блоків, як і раніше, визначається двома бітами.

16.7. Лінійні геш-таблиці

Модель геш-таблиць, що розширюються (розглянуто у двох попередніх пунктах) має кілька важливих переваг, найсуттєвіша з яких полягає в тому, що під час пошуку записів за заданим значенням ключа доводиться звертатися не більше, ніж до одного блока даних. Зрозуміло, що необхідно також переглядати вміст елементів масиву сегментів, однак він такий малий (може зберігатися в ОП), що необхідності виконання трудомістких операцій дискового введення/виведення у цьому випадку вдається уникнути. Проте доцільно нагадати і про серйозні недоліки, які зменшують цінність моделі геш-таблиць, що розширюються.

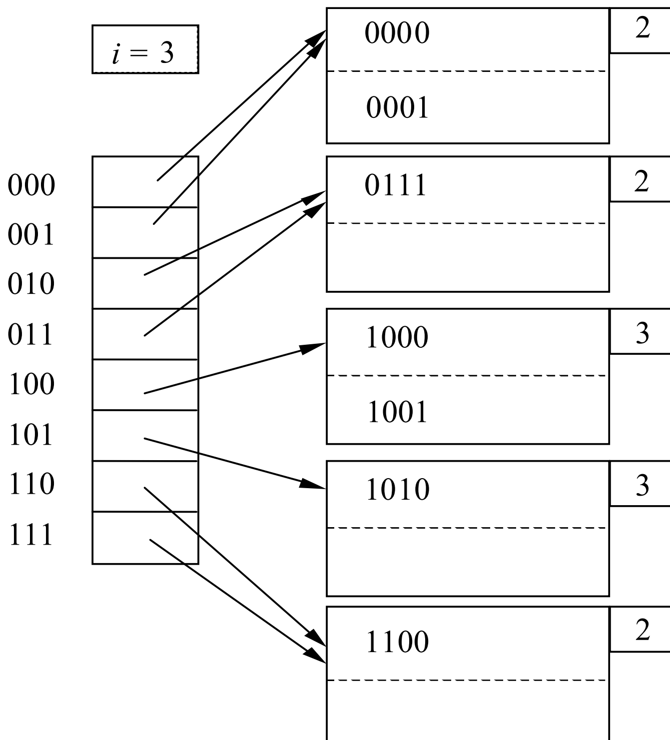


Рис. 72. Результат вставляння записів з геш-кодами 0000, 0111 і 1000 в геш-таблицю, проілюстровану на рис. 71

1. Процедура подвоєння розміру масиву сегментів пов’язана з чималими обчислювальними затратами (якщо значення параметра *i* велике) і спричинює призупинення процесів обробки запитів до файла даних на порівняно тривалі періоди часу.
2. Різке збільшення довжини масиву сегментів здатне спричинити до того, що масив більше не зможе зберігатися в ОП, не витісняючи з неї інші цінні порції інформації, внаслідок чого системі, що демонструвала до цього нормальні показники швидкодії, раптово доведеться здійснювати суттєво більшу кількість операцій дискового

введення/виведення, що негативно вплине на її продуктивність.

3. Якщо місткість блока мала, то необхідність розділення блоків виникає частіше, ніж це вимагалось за інших обставин. Наприклад, якщо кожен блок зберігає по два записи (як у розглянутих вище прикладах), то цілком можлива ситуація, коли три записи володіють геш-кодами, які збігаються у перших 20-ти бітах, хоча загальна кількість записів значно менша від 2^{20}. У подібному випадку доведеться брати i=20 і використовувати масив сегментів з мільйоном елементів навіть тоді, коли кількість блоків суттєво менша від мільйона.

Інша стратегія, що стосується використання лінійних геш-таблиць, дає змогу нарощувати кількість сегментів плавніше. Основні ідеї моделі лінійного гешування такі:

- Кількість n сегментів геш-таблиці завжди обирають так, щоб відношення середньої кількості записів у блоці до максимально допустимої залишалось постійним (наприклад, 80%).
- Оскільки розділення блоків не завжди можливе, то припустимо використовувати блоки переповнення, хоча середня питома кількість блоків переповнення у розрахунку на один сегмент таблиці значно менша від одиниці.
- Кількість бітів, які використовуються для нумерації елементів масиву сегментів, становить $\lceil \log_2 n \rceil$ де n – біжуча кількість сегментів. Біти обирають з молодших розрядів (правої частини) послідовності, яка повертається геш-функцією.
- Припустимо, що для нумерації елементів масиву сегментів використовують i бітів послідовностей, які отримують як результат обчислення геш-функції, і запис з ключем K підлягає розміщенню в сегменті з номером $a_1a_2...a_i$, тобто $a_1a_2...a_i$ – це i молодших бітів послідовності, повернутої функцією $h(K)$. Припустимо, що послідовність $a_1a_2...a_i$ трактується як i-розрядне двійкове

ціле число, яке дорівнює десятковому значенню m. Якщо $m<n$, то це означає, що сегмент з номером m існує і запис у цьому сегменті можна зберегти. Якщо $n \le m < 2^i$, то сегмента m немає, отож запис розташується у сегменті $m-2^{i-1}$, тобто у сегменті, який можна отримати, змінюючи значення a_1 (яке дорівнює 1) на 0.

Приклад 69. На рис. 73 представлено лінійну геш-таблицю, яка задовольняє умову $n=2$. Для визначення приналежності записів сегментам у певний час використовують тільки один біт. Як і в прикладі 67, для спрощення вважатимемо, що геш-функція повертає бітові послідовності довжиною 4 біти, і для представлення геш-кодів записів використовують усі 4 біти.

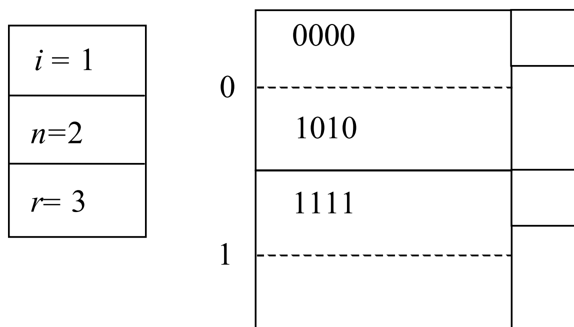


Рис. 73. Приклад лінійної геш-таблиці

Таблиця налічує два сегменти, кожен з яких містить по одному блокові. Сегменти перенумеровані як 0 і 1. Всі записи з молодшим нульовим розрядом геш-коду потрапляють у перший (верхній) сегмент, а записи з одиницею в молодшому розряді – у другий.

Частиною структури є параметри i (кількість бітів послідовності, які використовують у певний момент), n (біжуча кількість сегментів) і r (біжуча кількість записів, які зберігаються в геш-таблиці). Відношення r/n обмежено так, щоб для зберігання одного сегмента необхідно було, у середньому, один дисковий блок. Ми дотримуємося такої політики вибору значення n, згідно з якою файл містив би не більше, ніж $1,7n$ записів, тобто $r \le 1,7n$. Іншими

словами, оскільки блок уміщає тільки два записи, то середній рівень заповнення сегментів не повинен перевищувати 85% від місткості блока.

16.8. Вставка записів у лінійну геш-таблицю

За необхідності вставки в геш-таблицю нового запису для визначення сегмента таблиці, який підходить, використовують алгоритм, розглянутий у пункті 16.7. Спочатку для заданого значення K ключа пошуку запису обчислюють геш-функцію $h(K)$, і як шуканий номер m сегмента використовують i молодших бітів послідовності $h(K)$. Якщо $m<n$, то запис заноситься у сегмент m; якщо ж $m \geq n$, то запис розташовується у сегменті $m-2^{i-1}$. Якщо в обраному сегменті немає вільного місця, то створюється блок переповнення, який приєднується до ланцюга блоків сегмента.

Під час виконання кожної операції вставки значення співвідношення r/n аналізують з урахуванням кількості r записів, яка зросла. Якщо співвідношення стає надто великим, то в таблицю додається новий сегмент. Зауважимо, що сегмент, який додається, не має відношення до того, в який вставляється черговий запис. Якщо бінарним представленням номера доданого сегмента є послідовність $1a_2...a_i$, то розділяється сегмент $0a_2...a_i$. Записи заносяться в один із двох отриманих сегментів залежно від значення бітів a_i. Зазначимо, що всі ці записи повинні мати геш-коди, які завершуються послідовністю $a_2...a_i$, і розглядатиметься тільки значення i-го біта з правого боку.

Ще один важливий аспект стосується ситуації, коли значення n перевищує 2^i. У цьому випадку біжучий вміст i збільшується на одиницю і на початку бітового представлення номера кожного сегмента заноситься додатковий 0. Насправді нічого не змінюється, оскільки бітові послідовності, які інтерпретуються як цілі десяткові числа, залишаються попередніми.

Приклад 70. Продовжимо розгляд структури, представленої на рис. 73, і припустимо, що в неї необхідно вставити запис з ключем, що гешується у бітову послідовність 0101. Оскільки останнім бітом є 1, то запис необхідно розташувати у другому (нижньому) сегменті. Сегмент має достатньо вільного простору, отож блок переповнення не створюється.

Тепер, отже, 4 записи розташовано у 2-х сегментах, тобто задане відношення 1,7 перевищене, і ми зобов'язані збільшити значення n до 3-х. Оскільки $\lceil \log_2 3 \rceil = 2,$ то номерами сегментів замість 0 і 1 стають значення 00 і 01, проте структура даних залишається попередньою. Таблиця доповнюється новим сегментом з номером 10. Потім сегмент з номером 00 (який відрізняється від номера щойно створеного сегмента лише одним бітом) розділяється. У цьому випадку запис з геш-кодом 0000 залишається в сегменті 00 (геш-код завершується бітами 00), а запис з геш-кодом 1010, аналогічно, переноситься в сегмент 10. Результат наведено на рис. 74.

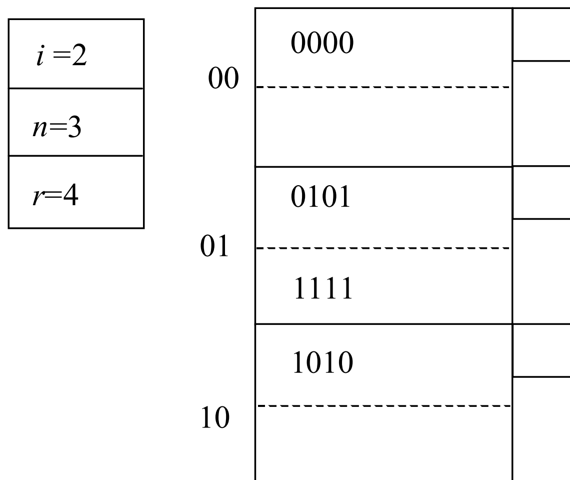


Рис. 74. Результат вставки запису з геш-кодом 0101 і долучення сегмента в лінійну таблицю, проілюстровану на рис. 73

Розглянемо тепер операцію вставки запису з геш-кодом 0001. Два останніх біти коду – це 01, отож запис підлягає розміщенню в існуючий сегмент з номером 01. На жаль, сегмент вже заповнено, і до нього приєднується блок переповнення. Три записи розподіляються у двох блоках сегмента відповідно до числового порядку слідування їхніх геш-кодів (вибір певного порядку не дуже

важливий). Оскільки відношення кількості записів до числа сегментів таблиці зараз дорівнює 5/3, тобто є меншим від 1,7, то новий сегмент не створюватимемо. Результат відображено на рис. 75.

Опишемо послідовність дій, необхідних для вставки запису з ключем, гешованим в 0111. Останні два біти геш-коду, 11, зазначають номер сегмента, в якому необхідно розташувати запис, однак такого сегмента в таблиці немає. Отож запис скеровується у сегмент 01, номер якого відрізняється нулем у першому біті. Запис займає вільне місце у блоці переповнення сегмента 01.

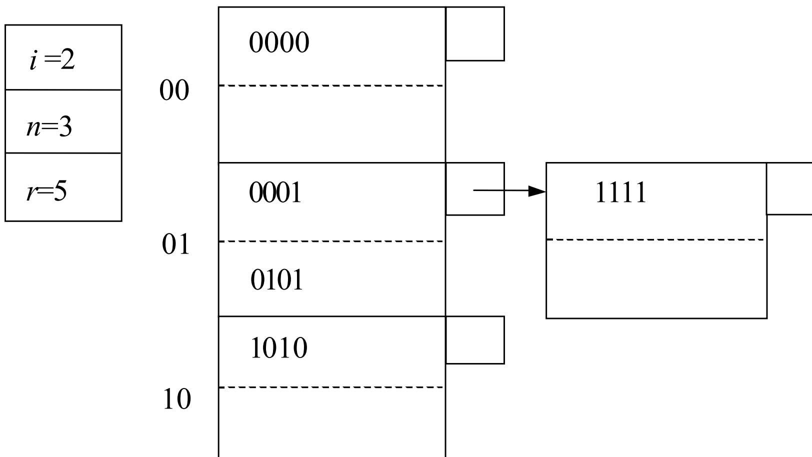


Рис. 75. Результат вставки запису з геш-кодом 0001 і долучення блока переповнення в геш-таблицю, проілюстровану на рис. 74

Однак тепер відношення кількості записів до числа сегментів геш-таблиці вже перевищує поріг 1,7, отож необхідно створити новий сегмент з черговим номером, 11. Так сталось, що це якраз той сегмент, в який потрібно було помістити новий запис. Записи з геш-кодами 0001 і 0101, які належать сегменту 01, залишаються на місці, а два інші (0111 і 1111) переносяться в новий сегмент. Оскільки сегмент 01 тепер містить тільки два записи, то блок переповнення є зайвим і видаляється. Остаточну структуру представлено на рис. 76.

Зауважимо, що чергова можлива операція вставки запису в геш-таблицю на рис. 76 спричинить до перевищення порогу 1,7, отож необхідно збільшити значення n та $\boldsymbol{i}$ до 5 і 3, відповідно.

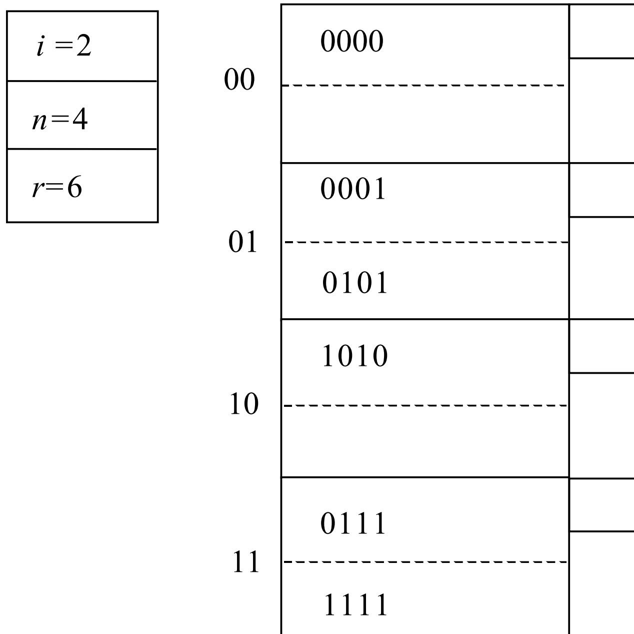


Рис. 76. Результат вставки запису з геш-кодом 0111 і долучення сегмента в геш-таблицю, проілюстровану на рис. 75

Приклад 71. Пошук у лінійних геш-таблицях виконується відповідно до описаної вище процедури, яку використовують при виборі сегмента, в який необхідно помістити запис, що вставляється. Якщо шуканого запису в обраному сегменті немає, то його немає ніде. Щоб проілюструвати сказане, звернемось до структури геш-таблиці, наведеної на рис. 74, де i=2 і n=3. Спочатку намагатимемось знайти запис з ключем, гешованим у бітову послідовність 1010. Оскільки i=2, то звернемо увагу на два останніх біти геш-коду запису – 10. Якщо їх інтерпретувати як ціле десяткове число, то це означатиме $m = 2$. Оскільки $m<n$, то

сегмент 10 існує і за записом необхідно звернутися саме до нього. Зверніть увагу, що факт відшукання запису з геш-кодом 1010 ще не означає, що це шуканий запис. Для впевненості необхідно перевірити повне значення ключа запису.

Тепер розглянемо процес пошуку запису з геш-кодом 1011. Цього разу необхідно звернутися до сегмента з номером 11. Оскільки значення 11, потрактоване як ціле десяткове число, є $m=3$, то маємо $m \geq n$, а це означає, що сегмента 11 у таблиці немає. Замінимо початкову одиницю номера на нуль і перейдемо до сегмента 01. Однак у цьому сегменті немає запису з геш-кодом 1011, і ми можемо бути впевнені, що його немає і в усій таблиці.

Вправи для опрацювання

Вправа 51. Поясніть, як вплинуть на вміст сегментів геш-таблиці, наведеної на рис. 67, перелічені нижче операції вставки і заміни записів із заданими ключами:

а) записи *g-j* вставляються у сегменти 0-3, відповідно;

б) записи *a* і *b* видаляються;

в) записи *k-n* вставляються у сегменти 0-3, відповідно;

г) записи *c* і *d* видаляються.

Вправа 52. Припустимо, що ключі записів гешуються в 4-бітові послідовності (як у всіх прикладах з розширеним і лінійним гешуванням, розглянутих вище). Вважатимемо, що блоки володіють місткістю, достатньою для зберігання трьох (а не двох, як раніше) записів, і розглянемо геш-таблицю з двома порожніми блоками (з номерами 0 і 1). Як виглядатиме структура після вставки записів з наступними геш-кодами:

а) 0000, 0001, ..., 1111; геш-таблиця, що розширюється;

б) 0000, 0001, ..., 1111; геш-таблиця є лінійною з граничним значенням відношення кількості записів до числа сегментів, яке дорівнює 100%;

в) 1111, 1110, ..., 0000; геш-таблиця, що розширюється;

г) 1111, 1110, ..., 0000; геш-таблиця є лінійною з граничним значенням відношення кількості записів до числа сегментів, яке дорівнює 75%.

Вправа 53. Нехай існує файл даних з 1 000 000 записів, які необхідно гешувати в таблицю з 1 000 сегментів. Один блок може вмістити 100 записів. Необхідно підтримувати настільки високий рівень заповнення блоків, наскільки це можливо, враховуючи, що блоки не повинні використовуватися спільно декількома сегментами. Якої кількості блоків (мінімальної і максимальної) достатньо для зберігання зазначеної структури?

ІНДЕКСИ ДЛЯ БАГАТОВИМІРНИХ ДАНИХ

Усі структури, які розглянуто раніше, є одновимірними, тобто мають єдиний ключ пошуку і дають змогу знаходити записи даних, що задовольняють значенням цього ключа. До цього часу ми вважали, що функцію ключа пошуку виконує окремий атрибут або поле. До категорії одновимірних можуть належати і такі індекси, ключ пошуку яких утворюється на основі декількох полів. Якщо необхідний одновимірний індекс з ключем пошуку, сформованим зі списку полів $(F_1, F_2, \dots, F_k)$, то вміст ключа можна представляти у вигляді ланцюга зв'язаних разом значень полів F_1, F_2 і т.д., розділених спеціальними символами.

Приклад 72. Якщо поле F_1 належить до рядкового типу, а поле F_2 – до цілочислового і роздільником слугує символ #, який у цьому випадку заборонено використовувати в середині рядкових значень, то комбінацію величин F_1=*'abcd'* і F_2=*123* можна оформити як рядок *'abcd#123'*.

У попередньому розділі ми зазначили переваги одновимірного простору значень ключа пошуку.

- Ефективність *B*-дерев та індексів для послідовних файлів зумовлена тим, що ключові значення розташовано упорядковано.
- Пошук у геш-таблиці стає можливим тільки в тому випадку, якщо значення ключа пошуку достовірно відоме. Якщо ключ утворено з декількох полів і вміст одного з них не визначено, то геш-функцію застосовувати не вдається – замість цього необхідно переглянути вміст всіх сегментів таблиці.

Існує чимало застосувань, які вимагають, щоб дані трактувались у контексті 2-, 3- або навіть *n*-вимірних просторів. Деякі з таких застосувань підтримуються традиційними СКБД, однак розроблено спеціалізовані системи, орієнтовані на обслуговування застосувань, що взаємодіють з багатовимірними даними. Одна з головних особливостей подібних систем стосується підтримки ними таких структур даних, які передбачають обробку деяких різновидностей запитів, що не є типовими для традиційних застосувань SQL. У

темі 17 представлено деякі зразки запитів, ефективність обслуговування яких суттєво зростає за використання індексів для багатовимірних даних. Теми 18 і 19 присвячено обговоренню характеристик перелічених нижче структур даних.

1. *Сіткові файли* (*grid files*) – багатовимірна версія геш-таблиць.
2. *Роздільні геш-функції* (*partitioned hash functions*) – альтернативний спосіб реалізації механізму геш-таблиць у застосуванні до багатовимірних даних.
3. *Індекси з декількома ключами* (*multiple-key indexes*) – структури, в яких індексні значення одного ключа пов’язані зі значеннями ключа індексів наступного рівня і т.д.
4. *kd-дерева* (*kd-trees*) – узагальнені моделі *B*-дерева у застосуванні до множин точок.
5. *Дерева квадрантів* (*quad trees*) – дерева, кожна дочірня вершина яких представляє один із квадрантів площі, що описується батьківською вершиною.
6. *R-дерева* (*R-trees*) – узагальнення моделі *B*-дерева для представлення множин областей простору.

Тема 17. Застосування моделі багатовимірних даних

У цій темі розглянемо два класи застосувань, які реалізують модель багатовимірних даних. Один стосується географічних інформаційних систем, де елементи даних описують сутності двовимірного (інколи тривимірного) світу. У застосуваннях другого класу поняття виміру трактується дещо абстрактно: кожен атрибут відношення можна інтерпретувати як деякий вимір; тоді кортежі доцільно сприймати як певні точки у просторі, обмеженому атрибутами-вимірами.

Окрім того, ми проаналізуємо способи застосування традиційних індексів, таких як *B*-дерева, для підтримки запитів до багатовимірних даних. У деяких випадках подібний підхід доцільний і ефективний, хоча в інших значно розумніше використовувати певні структури спеціального призначення.

17.1. Географічні інформаційні системи

Типова *географічна інформаційна система* має справу з об'єктами двовимірного (рідше – тривимірного) простору. Часто в базах даних подібних систем представлена картографічна інформація, яка описує положення і взаємну орієнтацію споруд, доріг, мостів, трубопроводів і багатьох інших фізичних об'єктів. Приклад подібної карти наведено на рис. 77.

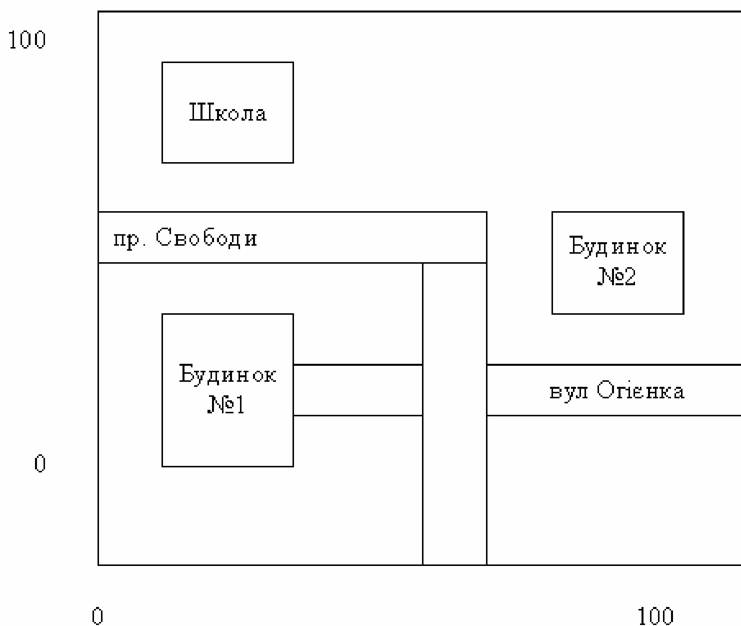


Рис. 77. Декілька об'єктів двовимірного простору

Існує чимало інших областей застосування подібних систем. Наприклад, монтаж інтегральної схеми виглядає як набір двовимірних карт прямокутних областей, виконаних зі спеціальних матеріалів і організованих у формі «шарів». Вікна і піктограми на екрані комп'ютера також можна трактувати як об'єкти двовимірного простору.

Запити, які обслуговуються географічними інформаційними системами, не належать до традиційної сфери використання SQL, хоча за певних умов їх все-таки вдається записати операторами SQL. Нижче перелічено деякі з різновидів подібних запитів.

1. *Запити до даних зі збіганням окремих координат:* задаються значення координат одного або декількох вимірів і повертається інформація про всі точки простору даних, які задовольняють критерію пошуку.
2. *Запити в діапазонах значень*: задаються діапазони значень для одного або декількох вимірів і вимагаються дані про точки, координати яких належать до цих діапазонів (прикладом може бути запит, який передбачає відшукання усіх двовимірних геометричних фігур, що частково або цілковито належать зазначеним областям площини). Подібні запити є узагальненням моделі одновимірних запитів у діапазонах значень, про яку ми розповідали у пункті 15.4.
3. *Пошук найближчих сусідніх об'єктів.* Наприклад, згадаємо запит, який передбачає виявлення точки, найближчої до заданої. Якщо точки представляють міста, то одним із подібних запитів може бути такий: знайти місто з населенням, що перевищує задану межу чисельності, яке найближче до вказаного міста.
4. *Визначення місцезнаходження об'єкта*: задається точка і вимагаються відомості про приналежність точки до тих чи інших фігур, областей чи об'єктів. Характерним прикладом може бути задача визначення положення курсора щодо екранних об'єктів у момент натискання клавіші миші.

17.2. Куби даних

Одне з недавніх досягнень в області технологій баз даних, яке реалізоване в деяких комерційних СКБД, стосується підтримки моделі *кубів даних*, де інформація інтерпретується у контексті багатовимірного простору. Багато корпорацій зайняті збиранням багатовимірних даних, які використовують у застосуваннях *підтримки прийняття рішень*, що дає змогу, наприклад, проаналізувати показники господарської діяльності підприємства на основі

інформації про продажі товарів, що виробляються. Ця інформація налічує:

1) дату і час угоди;
2) найменування пункту продажу;
3) назву товару, його колір, артикул, розмір і т.п.

Зазвичай, дані прийнято інтерпретувати у формі відношення, яке містить по одному атрибуту для списку кожної властивості деякої сутності. Атрибути відношення можна сприймати як окремі виміри багатовимірного простору, або «куба даних», а кожен кортеж – як точку в цьому просторі. Аналітики вводять запити, які групують дані за певним виміром, а потім підсумовують результати, отримані для груп, за допомогою функцій агрегації. Ось такий приклад подібного запиту: «Наскільки великі об’єми продажу оранжевих краваток артикулу 223-322 у кожному з магазинів мережі щомісяця за 1998 рік?»

17.3. SQL-запити до багатовимірних даних

Кожне із перелічених вище застосувань можна сформулювати у вигляді SQL-запиту, зверненого до реляційної бази даних традиційної структури. Розглянемо декілька прикладів.

Приклад 73. Припустимо, що необхідно виконувати серії запитів, які передбачають пошук найближчих сусідніх точок двовимірного простору. Інформацію про точки можна представити відношенням, яке містить пари дійсних чисел:

Points (x,y).

Іншими словами, схема відношення *Points* містить атрибути *x* і *y*, компоненти яких містять координати x та y точок площини. До відношення можуть належати і додаткові атрибути, призначені для зберігання значень інших властивостей точок.

Нехай треба визначити точку, найближчу до точки з координатами (10.0,20.0). Відповідний запит, текст якого наведено на рис. 78, визначає, чи існує для кожної точки *p* деяка точка *q*, розташована ближче до заданої точки, ніж *p*. У порівнянні беруть участь значення віддалей, які обчислюються як суми квадратів різниць координат x і y точки (10.0,20.0) і «біжучої» точки, що розглядається у конкретний момент. Зверніть увагу, що для

зменшення часу обробки запиту справжня віддаль між точками (квадратні корені суми квадратів значень) не обчислюється: порівняння квадратів величин даватиме такий результат, що і зіставлення самих величин.

```
SELECT *
FROM POINTS p
WHERE NOT EXISTS (
      SELECT *
      FROM POINTS q
      WHERE (q.x-10.0)*(q.x-10.0)+(q.y-20.0)*(q.e-20.0)<
                (p.x-10.0)*(p.x-10.0)+(p.y-20.0)*(p.y-20/0)
      );
```

Рис. 78. Приклад запиту, який дає змогу знайти точку, найближчу до заданої

Приклад 74. Об'єктами географічних інформаційних систем вельми часто є прямокутні фігури. Прямокутник на площині може бути описано декількома способами. Один із найуживаніших передбачає задання координат нижнього лівого і верхнього правого кутів. Колекцію прямокутників легко представити як відношення *Rectangles* з атрибутами ідентифікатора фігури і значень координат її кутів (а також будь-якими іншими атрибутами, які відповідають вимогам конкретної ситуації). У цьому прикладі користуватимемось схемою

Rectangles (id, xll, yll, xur, yur),

атрибути якої призначено для зберігання ідентифікатора *(id)* прямокутника, а також абсцис і ординат його нижнього лівого *(xll і yll)* і верхнього правого *(xur і yur)* кутів.

На рис. 79 наведено текст запиту, який дає змогу визначити прямокутники, що охоплюють точку з координатами (10.0, 20.0). Умова речення *WHERE* достатньо очевидна: щоб точка (10.0, 20.0) належала деякому прямокутнику, необхідно, щоб значення абсциси нижнього лівого кута цього прямокутника були не більші, ніж 10.0, а величина ординати – не більша, ніж 20.0. Окрім того, значення

абсциси і ординати верхнього правого кута повинні бути меншими від 10.0 і 20.0, відповідно.

```
SELECT id
FROM Rectangles
WHERE xll<=10.0 AND yll<=20.0 AND
          xur>=10.0 AND yur>=20.0;
```

Рис. 79. Приклад запиту, який дає змогу відшукати прямокутники, що охоплюють задану точку

Приклад 75. Інформацію в системах, які реалізують модель кубів даних, зазвичай, організовано у формі *таблиці фактичних значень*, що представляє базові елементи даних, які зберігаються (наприклад, відомості про кожну операцію продажу), і набору *таблиць розмірностей*, які описують властивості базових значень по кожному виміру. Наприклад, якщо одним із вимірів є параметр, який визначає місце здійснення угоди продажу (магазин), то атрибутами таблиці розмірностей, що відповідає цьому параметрові, повинні бути адреса, номер телефону та ім'я керуючого магазином.

Розмірностями таблиці фактичних значень

```
Sales (day, store, item, color, article)
```

є параметри, зазначені у пункті 17.2. Текст запиту, який передбачає отримання сумарної інформації про об'єми продажу оранжевих краваток у кожному з магазинів і за датами, може виглядати так, як проілюстровано на рис. 80. Тут роль атрибутів, які групують, виконують атрибути-розмірності *day* і *store*, а дані всіх інших вимірів агрегуються за допомогою оператора *COUNT*. Ми зосереджуємо увагу тільки на тих точках куба даних, які задовольняють умові значення *WHERE*, тобто дають відомості про краватки оранжевого кольору.

```
SELECT day, store, COUNT (*) AS totalSales
FROM Sales
WHERE item='tie' AND color='orange'
GROUP BY day, store;
```

Рис. 80. Приклад запиту, який повертає сумарну інформацію про продаж товарів

17.4. Запити в діапазонах значень з використанням традиційних індексів

Тепер дослідимо ступінь підвищення ефективності процесів обробки запитів у діапазонах значень з використанням структур індексів, які розглянуто у попередньому розділі. Припустимо, що йдеться про двовимірний простір властивостей сутностей, які розглядаються. За наявності вторинних індексів, сконструйованих для кожного з вимірів *x* і *y*, найкориснішим для відшукання значень, які належать заданим діапазонам цих вимірів, виявиться *B*-деревоподібний індекс. Спочатку листи *B*-дерева дають змогу отримати множину покажчиків на записи даних, які задовольняють діапазону виміру *x*. Потім *B*-дерево використовується для відшукання покажчиків на записи, які відповідають точкам, координати *y* яких належать до діапазону виміру *y*. Нарешті, за допомогою прийому, описаного в пункті 14.3, обчислюється перетин множин покажчиків. Якщо покажчики розміщені в ОП, то загальна кількість операцій дискового введення/виведення визначається кількістю листів *B*-дерева, що підлягають перегляду, і вимагає декілька додаткових операцій, необхідних для переміщення по дереву (див. пункт 15.7). До цього числа додають ті дискові операції, які необхідні для зчитування записів даних, що задовольняють критерій пошуку (таких записів може виявитися чимало).

Приклад 76. Розглянемо гіпотетичну множину з 1 000 000 точок, розподілених довільно у двовимірному просторі, які володіють координатами *x і y*, що набувають значень у діапазоні 0-1 000. Припустимо також, що в один дисковий блок поміщаються 100 записів з інформацією про точки і листок *B*-дерева містить, в середньому, 200 пар виду *«ключ-покажчик»* (нагадаємо, що в кожен момент часу зайняті не обов’язково всі комірки блоків-вершин дерева). Вважатимемо, що для вимірів *x* і *y* створені окремі *B*-деревоподібні індекси. Уявимо, що потрібно отримати відповідь на запит, який передбачає відшукання точок, що належать квадрату зі стороною 100, який розміщено в центрі вихідного квадрата 1000*1000. Іншими словами, координати точок повинні задовольняти умови $450 \le x \le 550$ і $450 \le y \le 550$. Використовуючи *B*-дерево для

атрибута *x*, неважко знайти покажчики на записи, що відповідають точкам, абсциси яких належать до заданого діапазону. Таких покажчиків виявиться приблизно 100 000, і необхідно, щоб усі вони помістилися в буферах ОП. Аналогічно, за допомогою *B*-дерева для атрибута *y* відшукуються покажчики (їх знову буде близько 100 000) на записи зі значеннями *y* з заданого діапазону. Операція перетину множин покажчиків дає в результаті множину, яка містить приблизно 10 000 покажчиків на записи, що задовольняють обидві умови водночас.

Тепер спробуємо оцінити кількість операцій дискового введення/виведення, які необхідні для отримання відповіді на запит. Передусім, як згадувалося у пункті 15.7, у багатьох випадках вельми продуктивним виявиться рішення, яке передбачає зберігання кореневої вершини *B*-дерева в ОП. Оскільки необхідно відшукати у кожному із двох *B*-дерев відповідний діапазон значень ключа пошуку (маючи на увазі, що покажчики в листах дерев впорядковані за величинами ключів), то для доступу до 100 000 покажчиків у кожному вимірі необхідно звернутися до однієї проміжної вершини і до всіх листів, які містять потрібні покажчики. Ми вважаємо, що один лист містить близько 200 пар виду *«ключ-покажчик»*, отож у кожному з дерев доведеться переглянути 500 блоків-листів. Якщо врахувати процедуру зчитування блока проміжної вершини кожного дерева, то загальна кількість операцій дискового введення/виведення становитиме 1 002.

Зрештою, необхідно завантажити блоки, які містять 10 000 шуканих записів. Якщо записи розподілені випадково, то нам доведеться мати справу з 10 000 різних блоків. Оскільки, як ми вважаємо, 1 000 000 записів файла розподілені у 10 000 блоках, то йдеться про необхідність перегляду в середньому кожного блока файла даних. Отже, традиційні індекси (щонайменше в цьому випадку) навряд чи будуть корисними при отриманні відповіді на запит у діапазонах значень. Якщо б діапазони були коротшими, то операція перетину множини покажчиків давала б змогу звузити область пошуку до невеликої частини блоків файла даних.

17.5. Пошук найближчих сусідніх об'єктів з використанням традиційних індексів

Для відшукання об'єктів, найближчих щодо заданого, як видається на перший погляд, згодиться практично довільна індексна структура: достатньо, наприклад, визначити прийнятні діапазони в кожному вимірі, виконати запит, подібний до розглянутого в попередньому пункті, й обрати точку, яка задовольняє всі діапазони і найближча до заданої. Подібний підхід, на жаль, не враховує дві ймовірні ситуації, за яких:

1) обрані діапазони не міститимуть потрібної точки;
2) найближча точка, яка задовольняє обмеження діапазонів, не буде найближчою у строгому розумінні цього слова.

Проаналізуємо обидві проблеми в контексті запиту з прикладу 73, вважаючи, як і в прикладі 76, що кожен із атрибутів x і y має певний індекс.

Якщо б ми мали достатньо підстав вважати, що на віддалі d від точки (10.0, 20.0) існує хоча б одна точка, то ми змогли б скористатися *B*-деревом для отримання покажчиків на всі записи, які представляють точки з абсцисами x, що належать діапазону від $10-d$ до $10+d$. Аналогічно, *B*-дерево для атрибута y даватиме змогу виявити покажчики на записи з інформацією про точки, ординати яких розташовані всередині проміжку $[20-d, 20+d]$. Якщо внаслідок перетину множин покажчиків було б отримано одну або декілька точок (ключами, які зберігаються разом з покажчиками, у цьому випадку слугує значення координат x та y), то ми змогли б визначити, яка з точок є найближчою до точки (10.0, 20.0), виконуючи завантаження лише одного запису. На жаль, гарантій того, що хоча б одна точка розташована на віддалі d від заданої, не існує, отож весь процес доведеться повторити (можливо, неодноразово) для деякого більшого значення d.

Якщо хоча б одна точка в обраному діапазоні й існує, то за певних обставин найближча точка, яка належить діапазону, виявиться віддаленішою від заданої, ніж деяка точка, що лежить поза діапазоном. Ситуацію проілюстровано на рис. 81. У цьому випадку необхідно розширити діапазон і повторити пошук, щоб переконатися, що ближчих точок немає: якщо найближча точка,

яка належить до обраного діапазону, знаходиться на віддалі d' від заданої точки і $d'>d$, то ми зобов'язані здійснити пошук знову, використовуючи замість d значення d'.

Приклад 77. Звернемось до набору даних та індексів, розглянутих у прикладі 76. Припустимо, що необхідно відшукати точку, найближчу до точки p=(10.0, 20.0). Вважатимемо, що d=1. Оскільки в кожному квадраті одиничної площі в середньому розташована одна точка, то в квадрат зі стороною 2, у центрі якого є точка p, з певною часткою ймовірності «потрапить» чотири точки.

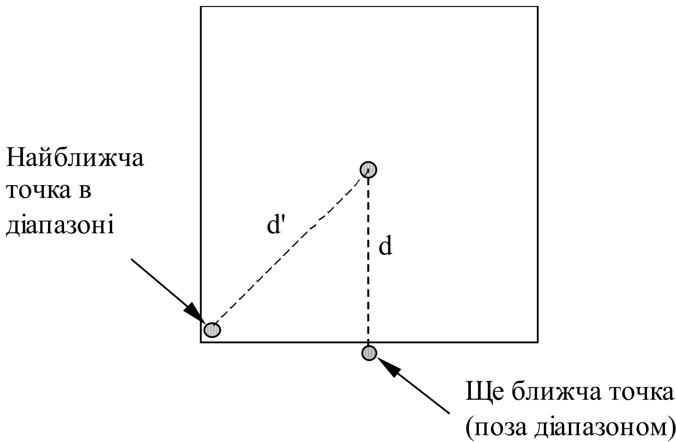


Рис. 81. Точка належить обраному діапазону, однак не є найближчою до заданої

Аналіз вмісту листів B-дерева для ключових атрибутів-координат x зі значеннями в діапазоні $9.0 \leq x \leq 11.0$ дає змогу отримати майже 2 000 точок. Щодо цього доведеться звернутися, щонайменше, до 10-ти блоків (найімовірніше, навіть до 11-ти, оскільки навряд чи початковим ключовим значенням першого блока виявиться $x = 9.0$). Як і в прикладі 76, кореневу вершину B-дерева доцільно зберігати в ОП, що даватиме змогу скоротити кількість операцій дискового введення/виведення до 12-ти: одна операція необхідна для зчитування блока проміжної вершини і 11 –

для звернення до блоків-листів. Ще 12 операцій необхідно для відшукання у другому *B*-дереві таких ключів-координат y, які належать діапазону $9.0 \leq y \leq 11.0$.

Виконуючи перетин двох множин із загальною кількістю покажчиків, яка майже дорівнює 4 000, ми відшукаємо, ймовірно, чотири таких покажчики, які адресують точки, що є найкращими кандидатами на роль найближчих «сусідів» точки *p*. Щоб визначити єдину шукану точку, достатньо взяти зв'язані з покажчиками значення ключів x і y, обчислити віддалі, порівняти їх і завантажити відповідний запис. Отже, у процесі обробки запиту доведеться виконати 25 операцій дискового введення/виведення. Якщо ж квадрат зі стороною 2 (при $d = 1$) не містить жодної точки і точка, найближча до заданої, розташована за межами цього квадрата, то пошук доведеться повторити для більшого значення d.

Напрошується наступний висновок: хоча традиційні структури індексів у деяких випадках придатні для обслуговування запитів, які передбачають пошук об'єкта, найближчого до заданого, вони все-таки вимагають виконання суттєво більшої кількості операцій дискового введення/виведення, ніж у тих випадках, коли за допомогою *B*-дерева необхідно відшукати запис за заданим значенням ключа (на що, ймовірно, буде потрібно не більше двох-трьох операцій). Методи, які розглядатимемо в наступних темах цього розділу, здатні забезпечити значно вищий рівень продуктивності системи, отож знаходять широке використання в СКБД, які орієнтовано на підтримку багатовимірних даних.

17.6. Інші обмеження традиційних структур індексів

Згадані раніше індексні структури демонструють себе не з кращого боку і під час виконання запитів у діапазонах значень, і під час пошуку об'єкта, найближчого до заданого. Підхід, використаний у прикладі 77 для розв'язання задачі відшукання найближчого об'єкта, фактично зводиться до запиту в діапазонах значень і допущення про те, що незначні розміри цих діапазонів виявляться достатніми для відшукання хоча б одного об'єкта. Проте не важко уявити, що відбудеться, якщо діапазони стануть ширшими, – кількість операцій дискового введення/виведення, необхідних для отримання всіх покажчиків на записи з інформацією про об'єкти,

що є потенційними кандидатами на роль «найближчого сусіда», зросте багатократно.

Якість підтримки запитів із багатовимірною агрегацією засобами традиційних індексів, аналогічних до розглянутих у прикладі 75, також не вражає. За наявності індексів для атрибутів *item* і *color* ми, звісно, здатні відшукати покажчики на всі записи, які представляють рівні продажу краваток і предметів одягу оранжевого кольору, і виконати перетин множин цих покажчиків за тою ж схемою, що і в прикладі 77. Однак найменша зміна структури запиту (наприклад, додання в речення *WHERE* умови, яка стосується будь-якого іншого атрибута) потребуватиме додаткового індексу.

Що ще гірше, впорядковуючи файл даних за одним із п’яти атрибутів, ми не можемо водночас підтримувати порядок сортування і за двома атрибутами. Отож для виконання більшості запитів, подібних за формою до розглянутих у прикладі 75, необхідно звернутися якщо не до всіх, то майже до всіх блоків файла даних. Якщо дані розміщуються на носіях вторинних пристроїв зберігання, то витрати стають надто великими.

17.7. Огляд структур індексів для багатовимірних даних

Більшість індексних структур, придатних для підтримки запитів до багатовимірних даних, можна зачислити до однієї з двох категорій:

1) геш-подібні структури;
2) деревоподібні структури.

Перелічимо ті характеристики індексів для одновимірних структур, які розглянуто у попередньому розділі і від яких необхідно відмовитися при переході до багатовимірних даних.

- Геш-подібні схеми – сіткові файли і роздільні геш-функції (детальніше у темі 18) – не забезпечують гарантії того, що для отримання відповіді на запит доведеться звернутися тільки до одного сегмента. Кожна зі структур дає змогу обмежити область пошуку деякою підмножиною сегментів.
- Деревоподібні схеми втрачають щонайменше одну з перелічених властивостей:

а) баланс дерева, за якого всі вершини-листи належать до одного рівня;
б) однозначну відповідність між вершинами дерева і дисковими блоками;
в) можливість швидкої модифікації даних.

Зауважимо, що гілки дерев індексів, сконструйовані для багатовимірних даних, часто володіють різною довжиною: більші гілки нерідко представляють області, що містять більшу кількість об’єктів. Окрім того, досить розповсюдженою є ситуація, коли об’єм інформації, що описує одну вершину дерева, значно менший від місткості дискового блока, що спричинює до необхідності групування вершин у блоках.

Вправи для опрацювання

Вправа 54. Використовуючи схему відношення

Rectangles(id, xll, yll, xur, yur),

описану в прикладі 74, сформулюйте такі SQL-запити.

1. Знайти множину прямокутників, які перетинаються з прямокутником, нижній лівий і верхній правий кути якого розташовані у точках з координатами (10.0,20.0) і (40.0,30.0), відповідно.
2. Знайти пари прямокутників, які перетинаються.
3. Знайти прямокутники, які охоплюють прямокутник з координатами кутів, зазначеними у п.1.
4. Знайти прямокутники, які розташовані всередині прямокутника з координатами кутів, зазначеними у п.1.
5. Знайти кортежі відношення з інформацією, яка не може представляти прямокутники, що реально існують.

Зазначте індекси (якщо такі необхідні), які сприятимуть ефективному виконанню кожного із запитів.

Вправа 55. Виконайте завдання з прикладу 76, вважаючи, що запит передбачає відшукання точок, які належать квадрату зі стороною n, розташованому в центрі вихідного квадрата 1 000x1 000, де 1<n<1 000. Скільки операцій дискового введення/виведення необхідно здійснити для отримання відповіді? Для яких значень n

традиційні структури здатні знизити трудомісткість обслуговування запиту?

Вправа 56. Виконайте завдання з вправи 55, якщо записи файла даних посортовані за значеннями *x*.

Тема 18. Геш-подібні структури для багатовимірних даних

У цій темі ми зосередимо увагу на двох структурах, які узагальнюють класичну схему геш-таблиць, що передбачала використання єдиного ключа. У кожній зі структур номер сегмента, що відповідає точці, є функцією всіх атрибутів або вимірів. Перша структура, (*сітковий файл – grid file)*, зазвичай, не передбачає «перемішування» (*hash*) значень за вимірами, однак розділяє виміри шляхом сортування значень, які до них належать. Схема *роздільних геш-функцій (partitioned hash functions)*, навпаки, передбачає гешування значень вимірів так, що кожен вимір вносить у номер сегмента власну складову.

18.1. Сіткові файли

Одну з найпростіших схем індексації, яка часто перевищує за ефективністю традиційні індекси, що використовуються для обробки запитів до багатовимірних даних, називають *сітковим файлом*. Модель сіткового файла можна представити так, ніби простір точок-об'єктів розбивається на частини уявною *сіткою*. *Лінії сітки*, паралельні до осей кожного виміру, ділять простір на *смуги*. Точки, що належать лінії, яка є нижньою межею певної смуги, належать до цієї смуги. Кількість ліній сітки за різними вимірами може бути різною. Окрім того, ширина смуг також може бути різною – навіть у межах одного виміру.

Приклад 78. Розглянемо запит (ми звертатимемось до нього неодноразово у цьому розділі), який можна коротко сформулювати так: «Які особи здатні придбати коштовності?» Уявимо, що існує база даних з інформацією про клієнтів ювелірних магазинів, яка містить найрізноманітніші відомості про кожного клієнта – його ім'я, адресу, номер телефону і т.п. З метою спрощення обмежимось тільки двома атрибутами, компоненти яких зберігають значення віку клієнта і його річного прибутку, вираженого у тисячах доларів

(К). Припустимо, що база даних містить інформацію про дванадцятьох клієнтів, яку представлено у формі пар «вік – прибуток»:

(25,60)	(45,60)	(50,75)	(50,100)
(50,120)	(70,110)	(85,140)	(30,260)
(25,400)	(45,350)	(50,275)	(60,260)

На рис. 82 зображено 12 точок, розподілених у двовимірному просторі. Кожен вимір поділено на смуги декількома лініями сітки. У нашому випадку ми використовували по дві лінії для кожного виміру, що спричинило до розбиття простору на дев’ять прямокутних областей, хоча жодних особливих доводів на користь вибору однакової кількості ліній сітки для всіх вимірів не існує. Лінії розташовані так, що смуги мають різну ширину. Вертикальні смуги, що відповідають виміру «вік», наприклад, мають ширину 40, 15 і 45.

У нашому прикладі жодна точка не розташована безпосередньо на лінії сітки, хоча в загальному випадку відповідний прямокутник містить такі точки, які належать його нижній або лівій (але не верхній або правій) межам. Наприклад, центральний прямокутник на рис. 82 представляє точки, координати яких задовольняють умови $40 \leq \textit{вік} < 55$ і $90 \leq \textit{прибуток} < 225$.

18.2. Пошук записів у сітковому файлі

Кожну з прямокутних областей, що отримуються як результат перетину смуг простору, можна трактувати як *сегмент геш-таблиці*: точки, які належать області, представлено записами, розташованими у *блоці*, який належить до відповідного сегмента. Якщо для зберігання інформації про всі точки області місткості одного блока не достатньо, то створюють необхідні блоки *переповнення*.

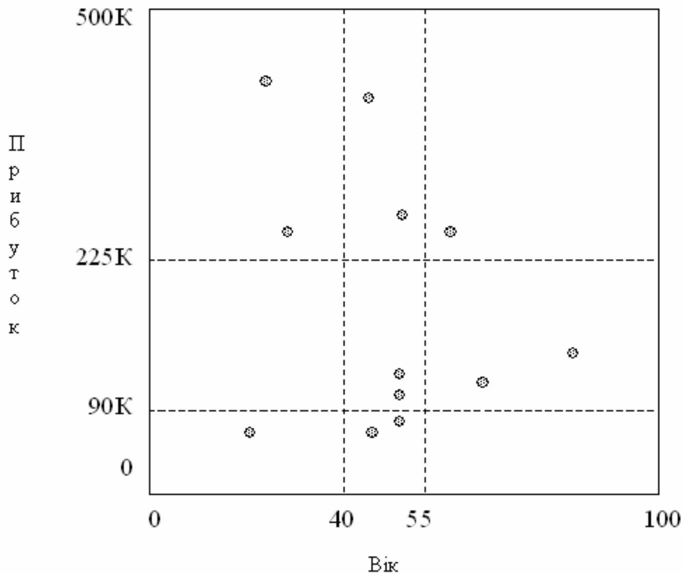


Рис. 82. Фрагмент двовимірного простору даних

На відміну від одновимірного масиву сегментів, що використовується в традиційних геш-таблицях, сітковий файл містить масив сегментів, кількість розмірностей якого збігається з числом вимірів файла даних. Щоб виявити факт належності точки до певного сегмента, необхідно володіти списками позицій ліній сітки за кожним виміром. Процес гешування інформації щодо точки у цьому випадку відрізняється від звичайного використання геш-функції до значень компонентів даних, що представляють точку: значення кожної координати точки порівнюється зі списком позицій ліній сітки для відповідного виміру; номер сегмента визначається сукупністю ознак належності точки смугам простору в усіх вимірах.

Приклад 79. На рис. 83 зображено дані рис. 82, розподілені у сегменти геш-таблиці. Оскільки кількість смуг у кожному з двох вимірів дорівнює 3, то масив сегментів є матрицею 3x3. Два сегменти, перший з яких визначається умовами 0≤*вік*<40 і

90≤*прибуток*<225, а другий – умовами *вік*>55 і 0≤*прибуток*<90, є порожніми. Отож дискові блоки, що їм відповідають, до діаграми не зачислено. Кожен з блоків решти сегментів містить не більше, ніж два записи (очевидно, що подібна ситуація досить штучна), отож необхідності у використанні блоків переповнення не виникає.

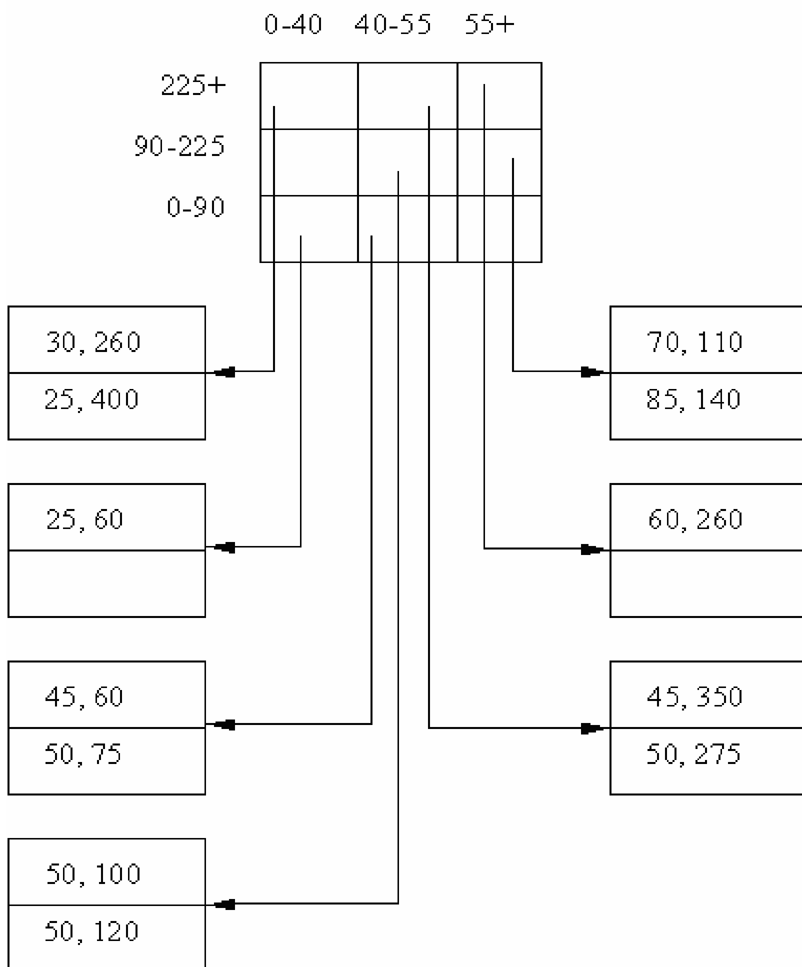


Рис. 83. Сітковий файл, який представляє дані з рис. 82

✓ **Корисно знати.** *Доступ до сегментів сіткового файла.* Відшукання сегмента, якому належить точка із координатами, у сітковому файлі з невеликою кількістю смуг (наприклад, 3x3, як на рис. 83) – це, очевидно, не проблема. Інша справа, якщо вимір містить значно більшу кількість смуг. У подібній ситуації без індексів, ключі пошуку яких слугують значеннями позицій ліній сітки для кожного виміру, звісно не обійтись.

Для заданого значення *v* деякої координати визначають ключове значення *w*, не більше від *v*. Відшукана величина *w* зв'язується індексом з номером смуги, якій належить точка з координатою *v*. Використовуючи аналогічні операції для кожного виміру, ми зможемо визнати елемент матриці, який представляє шуканий сегмент, а потім за допомогою покажчика, що міститься в сегменті, переміститися до відповідного дискового блока.

В екстремальних ситуаціях, коли матриця надто велика і розріджена, зберігання інформації про всі порожні сегменти може виявитися недоцільним або навіть неможливим. У подібних випадках матрицю необхідно перетворити у відношення з компонентами, які містять «координати» непорожніх сегментів і покажчики на дискові блоки, адресовані цими сегментами. Пошук даних у такому відношенні, зазвичай, залишається багатовимірним, однак завдяки меншим розмірам структури він буде ефективнішим.

18.3. Вставка записів у сітковий файл

Для вставки нового запису в сітковий файл спочатку необхідно визначити сегмент, який підходить, використовуючи процедуру, коротко описану в попередньому пункті. Якщо блок, який відповідає сегментові, містить достатньо вільного простору, то

запис вставляється й операція завершується. Проте задача ускладнюється, якщо блок вже заповнено. Щодо її розв'язання використовують дві стратегії.

1. Додати блок переповнення. Подібний підхід непоганий тільки у тих випадках, коли ланцюг блоків переповнення не надто довгий. У протилежному випадку кількість операцій дискового введення/виведення, необхідна для виконання процедури пошуку, вставки і вилучення записів, стає неприйнятно великою.
2. Реорганізувати структуру шляхом додавання ліній сітки або їхнього переміщення. Прийом нагадує технологію динамічного гешування, про який ми розповідали у темі 16, однак проблема ускладнюється тим, що вміст сегментів у межах вимірів взаємопов'язаний. Іншими словами, вставлення нової лінії сітки спричинить до розщеплення усіх областей-сегментів, які вона перетинає. Вибір лінії, «вдалої» для всіх сегментів без винятку, може виявитися неможливим. Наприклад, якщо рівень заповнення деякого сегмента надто високий, цілком імовірна ситуація, коли просто не вдається знайти вимір, що підлягає розщепленню, або точку, поблизу якої доцільно «провести» лінію сітки, не створюючи надто великої кількості порожніх сегментів і не залишаючи інші сегменти надто заповненими.

Приклад 80. Припустимо, що активним покупцем коштовностей є особа віком 52 роки, яка володіє річним прибутком, що дорівнює 200 тисяч доларів. Точка, яка відповідає цьому клієнту, повинна належати центральній прямокутній області простору, зображеного на рис. 82. Після долучення точки сегмент міститиме вже три записи. Для розташування нового запису можна створити блок переповнення. Якщо ж сегмент необхідно розщепити, то доведеться обрати, по-перше, один із двох вимірів і, по-друге, позицію лінії сітки всередині сегмента. Існує три варіанти розташування лінії сітки, які забезпечують розщеплення центрального сегмента так, щоб дві точки залишились з одного боку від лінії, а третя – з іншого.

1. Вертикальна лінія з координатою *вік*=51, яка відокремлює дві точки з координатою *вік*=50 від точки з координатою *вік*=52. Множина точок верхнього і нижнього сегментів, які відповідають тій же умові 40≤*вік*<55, не зачіпаються, оскільки всі точки одного і другого залишаються з лівого боку від нової лінії сітки.
2. Горизонтальна лінія, яка відокремлює точку з координатою *прибуток*=200 від двох інших точок з меншими координатами (100 і 120). Значення координати *прибуток* лінії сітки можна обрати рівним, наприклад, 130, що спричинить до розщеплення множини точок правого сегмента, який задовольняє умови *вік*>55 і 90≤*прибуток*<225.
3. Горизонтальна лінія, яка відокремлює точку з координатою *прибуток*=100 від двох інших точок з більшими координатами (120 і 200). Значенням координати *прибуток* лінії сітки тепер може слугувати обране число, (наприклад, 115), і це знову спричинить необхідність розщеплення множини точок правого сегмента.

Варіант 1, ймовірно, не найкращий, оскільки під час його використання вміст інших сегментів, окрім центрального, не розщеплюється і створюються зайві порожні сегменти. Варіанти 2 і 3, як видається, однаково хороші, проте другий, можливо, кращий, оскільки лінія сітки з координатою *прибуток*=130 проходить ближче до середини діапазону 90-225. Результат зображено на рис. 84.

18.4. Оцінки ефективності структур сіткових файлів

Розглянемо, скільки операцій дискового введення/виведення необхідно для виконання різних запитів до даних, які проіндексовано за допомогою сіткових файлів. Зосередимо увагу на двовимірній версії сіткових файлів, хоча ті ж висновки можна розповсюдити і на структури з довільною кількістю вимірів. Одна з головних проблем, яка супроводжує використання багатовимірних сіткових файлів, полягає в тому, що з доданням чергового виміру кількість сегментів зростає за експоненціальним законом. Якщо крупні ділянки простору залишаються порожніми, то матриця

виявляється розрідженою. Така ж ситуація можлива навіть при використанні двох вимірів.

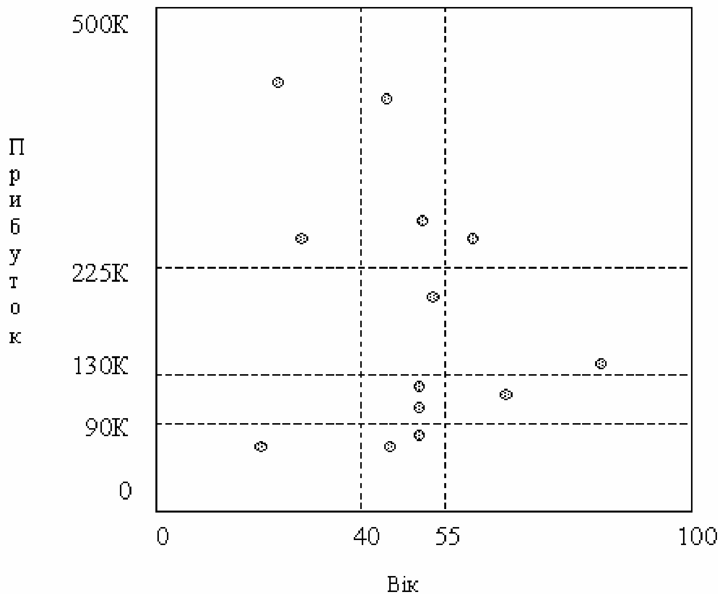


Рис. 84. Вставка точки (52,200) у фрагмент простору даних, який проілюстровано на рис. 82, із розщепленням сегментів

Припустимо, що значення віку і прибутку значною мірою корельовано; тоді, (див. рис. 82) більшість точок буде зосереджено поблизу діагоналі матриці, і, незалежно від позицій ліній сітки, сегменти поза діагоналлю виявляться порожніми. Якщо ж розподіл точок близький до рівномірного і файл даних не надто великий, то положення ліній сітки можна обрати таким, щоб забезпечити виконання наступних умов:

1. Кількість сегментів настільки мала, що індексний файл вдається розмістити в ОП. Це дає змогу уникнути необхідності виконання операцій дискового введення/ виведення під час доступу до сегментів індексу і додавання смуг у випадку вставки нових ліній сітки.

2. Індекси для значень позицій ліній сітки щодо кожного виміру зберігаються в ОП або необхідність використання індексів зникає унаслідок застосування відповідних процедур бінарного пошуку таких значень.
3. Типовий сегмент містить не більше, ніж декілька блоків переповнення, що дає змогу уникнути зайвих затрат під час звертання до сегмента.

Проаналізуємо, як поводять себе сіткові файли, що задовольняють вказані умови, під час обробки запитів декількох основних категорій.

Пошук заданої точки. Координати точки безпосередньо гешуються в номер сегмента, що містить покажчик на блок; для доступу до останнього необхідна одна дискова операція введення/виведення. Під час вставки/вилучення запису необхідною є додаткова дискова операція. Процедура вставки, що вимагає створення блока переповнення, збільшує кількість дискових операцій ще на одиницю.

Запити до даних із рівністю окремих координат. Прикладами подібних запитів можуть бути такі, як «знайти інформацію про всіх 50-річних клієнтів» або «повернути відомості про покупців з прибутком у 200 тисяч доларів». У подібних випадках доводиться звертатись до всіх сегментів, які належать певній смузі простору. Якщо смуга містить багато непорожніх сегментів, то кількість операцій дискового введення/виведення суттєво зростає.

Запити в діапазонах значень. Запит у діапазоні значень визначає прямокутну область сітки, і відповіддю на нього буде множина точок, що належать сегментам, які вкривають цю область (за винятком деяких точок у сегментах, що прилягають до межі області пошуку). Наприклад, за необхідності відшукання відомостей про покупців віком 35-45 років з прибутком у межах 50-100 тисяч доларів доведеться звернутися до чотирьох сегментів у нижній лівій частині рис. 82. У цьому випадку жоден із сегментів не належить області пошуку цілковито, отож значну частину точок з остаточної відповіді, ймовірно, доведеться викинути. Якщо ж область пошуку охоплює значну кількість сегментів, чимало з яких виявляються всередині області, то всі їхні точки одразу ж

задовольняють критерій пошуку. Оскільки обробка запитів у діапазонах значень часто залежить від перегляду багатьох сегментів, то кількість операцій дискового введення/виведення може виявитися вельми значною. Як результат виконання подібних запитів, зазвичай, повертаються крупні порції даних, отож затрати на зчитування блоків файла вихідних даних співрозмірні з витратами на запис блоків (якщо такий передбачається) з підсумковою інформацією.

Пошук найближчих сусідніх об'єктів. Пошук точок, найближчих щодо деякої заданої точки P, розпочинається з визначення сегмента, якому належить P. Якщо сегмент містить хоча б ще одну точку Q, то її можна вважати потенційним кандидатом на роль шуканої точки. Трапляються випадки, коли точки у суміжних сегментах розташовані ближче до P, ніж точка Q з того ж сегмента. Прикладом може бути рис. 82. Отож необхідно переконатися, чи не менша віддаль від точки P до межі сегмента, до якого вона належить, ніж дистанція між P і Q. Якщо ситуація саме така, то необхідно проаналізувати і місцезнаходження точок відповідного сусіднього сегмента. Окрім того, якщо прямокутники-сегменти надто розтягнуті в одному вимірі і дуже вузькі в другому, то ймовірно, що доведеться звернутися не тільки до найближчих, але і до віддаленіших «сусідів» сегмента з точкою P.

Приклад 81. Нехай у структурі, зображеній на рис. 82, необхідно знайти точку, найближчу до точки P=(45,200). Найближча точка, яка належить до того ж сегмента, володіє координатами (50,120) і є від точки P на віддалі 80,2. Жодна з точок трьох нижніх сегментів не може бути ближчою, оскільки верхньою межею цих сегментів є лінія сітки з координатою *прибуток*=90, отож 200-90>80,2, і сегменти можна не розглядати. Проте п'ять інших сегментів звісно потребують уваги, і наші зусилля виявляються не марними: тут ми відшукаємо дві точки з координатами (30,260) і (60,260), розташовані на однаковій віддалі від точки P, що дорівнює 61,8. Загалом, процедура пошуку найближчого сусіднього об'єкта, зазвичай, обмежується декількома сегментами, отож вимагає виконання невеликої кількості операцій дискового введення/виведення. Оскільки сегменти, найближчі до

вихідного, можуть виявитися порожніми, то верхню межу трудомісткості пошуку наперед передбачити неможливо.

18.5. Роздільні геш-функції

Геш-функції як аргумент здатні отримувати списки значень атрибутів, хоча, зазвичай, гешуванню піддаються значення одного атрибута. Наприклад, якщо a і b – цілочисловий і рядковий атрибути, відповідно, то для обчислення номера сегмента геш-таблиці, використаної як індекс для пар значень (a,b), можна додати величину a з ASCII-кодами усіх символів рядка b, поділити отриману суму на кількість сегментів і взяти залишок від ділення. Однак подібну геш-функцію можна використати тільки в таких запитах, в яких водночас беруть участь значення атрибутів a і b. Краще ж проектувати геш-функцію так, щоб вона повертала певну кількість бітів (наприклад, k), розподілених між n атрибутами, де k_i бітів геш-коду обчислюють на підставі i-го атрибута і $\sum_{i=1}^{n} k_i = k$. Якщо казати точніше, то геш-функція h насправді є списком (h_1. h_2. …, h_n) геш-функцій, де функція h_i застосовується до значення i-го атрибута і повертає послідовність з k_i бітів. Номер сегмента індексу, з якого адресується кортеж зі значеннями (v_1, v_2, …, v_n) n атрибутів, обчислюється шляхом зчеплення часткових бітових послідовностей: $h_1(v_1)h_2(v_2)\ldots h_n(v_n)$.

Приклад 82. Припустимо, що є кортеж, компонент a якого містить значення A, а компонент b – значення B (інші компоненти в гешуванні участі не беруть). Нехай сегменти геш-таблиці володіють 10-бітовими номерами, тобто таблиця може містити до 1024 сегментів, і чотири біти відводяться для атрибута a, а решта шість – атрибута b. Функція h_a, пов’язана з атрибутом a, гешує значення A і повертає чотири біти (наприклад, 0101), а функція h_b, поставлена у відповідність атрибутові b, на основі значення B обчислює послідовність із шести бітів (111000). Отож кортежу (A,B) відповідатиме сегмент геш-таблиці, що володіє номером 0101111000, який налічує дві часткові бітові послідовності, отримані для значень A і B.

Таке роздільне гешування має ту перевагу, що дає змогу використовувати відомі значення будь-яких одного або декількох компонентів, які вносять свій вклад в остаточний геш-код. Наприклад, дано значення *A* атрибута *a*, яке за допомогою виклику функції $h_a(A)$ гешується в бітову послідовність 0101. Посилання на будь-які кортежі, що містять у компонентах *a* значення *A*, необхідно шукати тільки в межах 64-х сегментів, номери яких описуються бітовими послідовностями формату 0101..., де трикрапка позначає шість довільних бітів. Аналогічно, якщо відоме значення *B* атрибута *b*, то це дає змогу обмежити область пошуку 16-ма сегментами, номери яких завершуються бітовою послідовністю $h_b(B)$=111000.

Приклад 83. Припустимо, що дані про покупців коштовностей, розглянуті у прикладі 78, необхідно представити у формі роздільної геш-таблиці з 8-ма сегментами, відводячи під номер сегмента по три біти (один відповідає атрибуту віку, а два – атрибуту прибутку). Як і раніше, вважатимемо, що блок може містити не більше, ніж два записи.

Припустимо, що геш-функція для атрибута «вік» обчислює результат ділення значення віку за модулем 2. Іншими словами, вміст кортежу «вік-прибуток» для парного значення віку гешується в бітову послідовність вигляду 0*xy* з деякими бітами *x* і *y*, а для непарного – у послідовність 1*xy*. Геш-функція для атрибута «прибуток» аналогічно визначає результат ділення прибутку, що виражається в тисячах, за модулем 4. Наприклад, для величини прибутку, подібній 57, яка ділиться на 4 із остачею 1, пара значень «вік-прибуток» гешується у послідовність вигляду *z*01 з деяким бітом *z*.

На рис. 85 проілюстровано результат роздільного гешування даних з прикладу 78 відповідно до вказаного вище алгоритму. Зверніть увагу на таку обставину: оскільки більшість обраних значень віку і прибутку ділиться на 10, геш-функція не може забезпечити достатньо рівномірний розподіл точок у сегменти. Два із восьми сегментів містять по чотири записи і потребують блоків переповнення, а три сегменти залишаються порожніми.

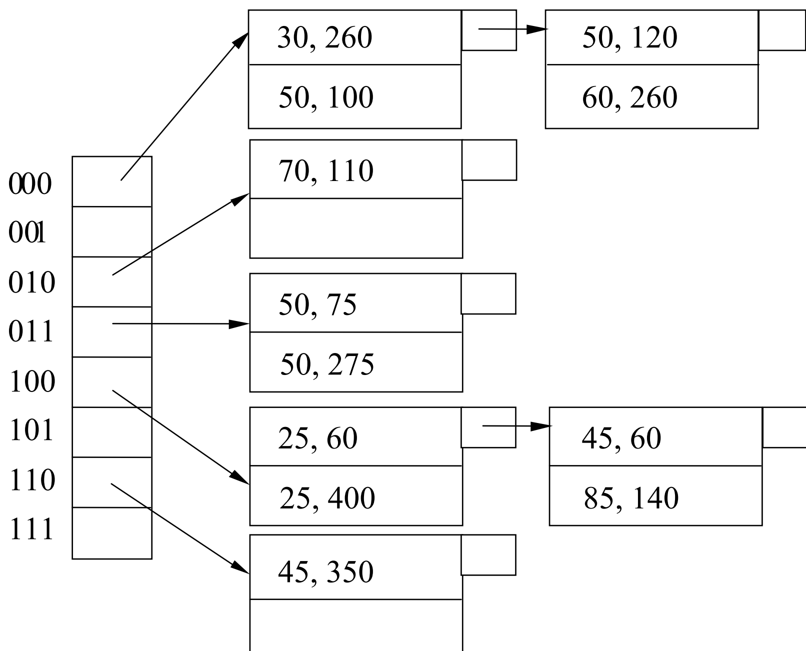


Рис. 85. Приклад роздільної геш-таблиці

18.6. Порівняння структур сіткових файлів і роздільних геш-таблиць

Показники продуктивності структур індексів, розглянутих у цій темі, значно відрізняються. Нижче перелічені основні оцінки ефективності структур сіткових файлів і таблиць, які отримують роздільним гешуванням.

- З метою пошуку найближчих сусідніх об'єктів і обробки запитів у діапазонах значень роздільні геш-таблиці менше корисні, ніж сіткові файли, оскільки значення фізичних віддалей між точками не знаходять відображення в близькості номерів сегментів, яким належать ці точки. Звісно, можна було б спроектувати геш-функцію для деякого атрибута *a* так, щоб для мінімального значення *a* генерувався бітовий рядок, що

представляє найменше число, тобто містить всі нульові біти, наступному за величиною значенню *a* відповідало більше число вигляду 00...01 і т.д. Проте у цьому випадку ми знову мимоволі повернулись до структури сіткового файла.

- Вдало обрані результати геш-функції здатні забезпечити розподіл точок простору у сегменти індексу, близький до рівномірного. Сіткові файли, однак, «схильні» залишати чималу кількість сегментів порожніми (передусім, для значної кількості розмірностей). Причина полягає в тому, що за наявності багатьох атрибутів щонайменше деякі з них значною мірою корельовано, отож значні області простору виявляються незаповненими. Наприклад, у пункті 18.4 ми згадували про те, що чинник кореляції значень віку і прибутку зумовлює розміщення більшості точок фрагмента простору, проілюстрованого на рис. 82, вздовж діагоналі матриці. Як наслідок, за використання роздільної геш-таблиці вдається обійтись меншою кількістю сегментів і блоків переповнення, ніж у випадку використання сіткового файла.

Отож, якщо йдеться переважно про виконання запитів до даних із рівністю окремих координат, де задають значення одних атрибутів й ігнорують іншими, то роздільна геш-таблиця з більшою ймовірністю перевищить за продуктивністю індекс на основі сіткового файла. І навпаки, якщо в системі систематично виконують пошук найближчих сусідніх об’єктів або запити у діапазонах значень, то перевагу треба надавати структурам сіткових файлів.

Вправи для опрацювання

Вправа 57. Розглянемо відношення, яке містить інформацію про персональні комп’ютери та описується схемою:

PC(model, speed, ram, hd)

На рис. 86 наведено специфікації 12 персональних комп’ютерів. Нехай необхідно створити індекс для атрибутів швидкодії процесора (*speed*) і місткості жорсткого диска (*hd*).

1. Оберіть позиції п’яти ліній сітки (йдеться про загальну кількість ліній для двох вимірів) сіткового файла, які забезпечують потрапляння у кожен сегмент не більше, ніж двох точок.
2. Чи можливо виконати завдання з п.1, використовуючи тільки чотири лінії сітки? Наведіть докази за або проти.
3. Запропонуйте варіант роздільного гешування, за якого кожен з чотирьох сегментів міститиме не більше, ніж чотири точки.

model	speed	ram	hd
1001	700	64	10
1002	1500	128	60
1003	866	128	20
1004	866	64	10
1005	1000	128	20
1006	1300	256	40
1007	1400	128	80
1008	700	64	30
1009	1200	128	80
1010	750	64	30
1011	1100	128	60
10013	733	256	60

Рис. 86. Фрагмент екземпляра відношення PC з характеристиками персональних комп’ютерів

Вправа 58. Нехай необхідно розмістити дані, наведені на рис. 86, у тривимірному сітковому файлі, виміри якого відповідають атрибутам швидкодії процесора (speed), об’єму оперативної пам’яті (ram) і місткості жорсткого диска (hd). Запропонуйте варіант розбиття простору даних лініями сітки, який забезпечує рівномірний розподіл точок у сегментах.

Вправа 59. Розробіть версію набору роздільних геш-функцій, які перетворюють значення атрибутів speed, ram і hd відношення, зображеного на рис. 86, в однобітові часткові послідовності і забезпечують рівномірний розподіл точок у сегментах.

Вправа 60. Припустимо, що дані, наведені на рис. 86, оформлено як сітковий файл з двома вимірами, які відповідають атрибутам speed і ram. Лінії сітки для осі speed мають координати 720, 950, 1150 і 1350, а лінії сітки для виміру ram розташовані на позиціях 100 і 200. Припустимо також, що блок сегмента може містити не більше, ніж два записи. Запропонуйте оптимальні варіанти розщеплення сегментів для вставки точок з координатами:

а) speed=1000, ram=192;

б) speed=800, ram128 і speed=833, ram=96.

Тема 19. Деревоподібні структури для багатовимірних даних

Розглянемо кілька індексних структур, які ефективні під час обробки запитів у діапазонах значень і пошуку об'єктів, найближчих до заданого:

1) індекси з декількома ключами;
2) *kd*-дерева;
3) дерева квадрантів;
4) *R*-дерева.

Три перші структури призначено для представлення множин точок. *R*-дерева, зазвичай, використовують з метою опису множин областей простору, хоча застосовують і для відображення множин точок.

19.1. Індекси з декількома ключами

Припустимо, що схема відношення містить атрибути, які представляють значення координат елементів даних багатовимірного простору, і необхідно забезпечити ефективну підтримку запитів у діапазонах значень координат або процедур пошуку об'єктів, найближчих до заданого. Одне з простих рішень пов'язане з використанням індексу індексів або (узагальнено), деревоподібної схеми, в якій вершини кожного рівня є індексами для певного атрибута.

Головну ідею проілюстровано на рис. 87 на прикладі двох атрибутів. «Коренем дерева» слугує індекс для першого атрибута. Подібну роль може відігравати індекс будь-якого з традиційних

типів, таких як *B*-дерево або геш-таблиця. Індекс пов'язує значення ключа пошуку (вміст компонента першого атрибута) з покажчиком на деякий індекс для другого атрибута. Якщо *V* – вміст компонента першого атрибута, то відшукавши елемент зі значенням *V* ключа пошуку в першому індексі і простуючи у «напрямі» покажчика, зв'язаного з цим елементом, ми «потрапимо» до індексу, що представляє підмножину точок, перші координати яких дорівнюють *V*, а другі містять довільні значення.

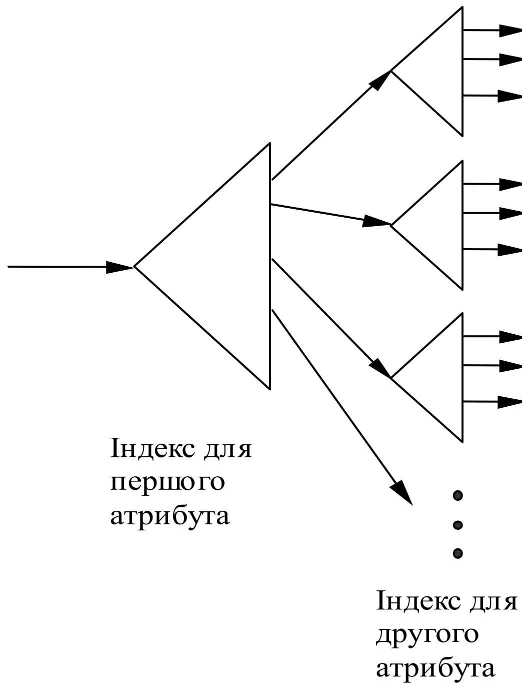


Рис. 87. Приклад багаторівневого індексу з декількома ключами

Приклад 84. На рис. 88 схематично зображено індекс з декількома ключами, який відповідає прикладу, що ми розглядаємо, і який стосується відомостей про колекціонерів коштовностей: «першим» атрибутом у цьому випадку є «*вік*», а «другим» –

«*прибуток*». «Кореневий» індекс, який відповідає атрибуту віку, розташовано у лівій частині рисунка. Тут ми не уточнюємо деталей реалізації цього індексу – сім його пар виду *«ключ-покажчик»*, наприклад, можуть займати позиції у вершинах-листах *B*-дерева. Важливіше те, що індекс містить тільки такі ключові значення віку, яким відповідає одна або декілька існуючих точок простору даних, і дає змогу легко відшукати покажчик, зв'язаний з певним значенням ключа.

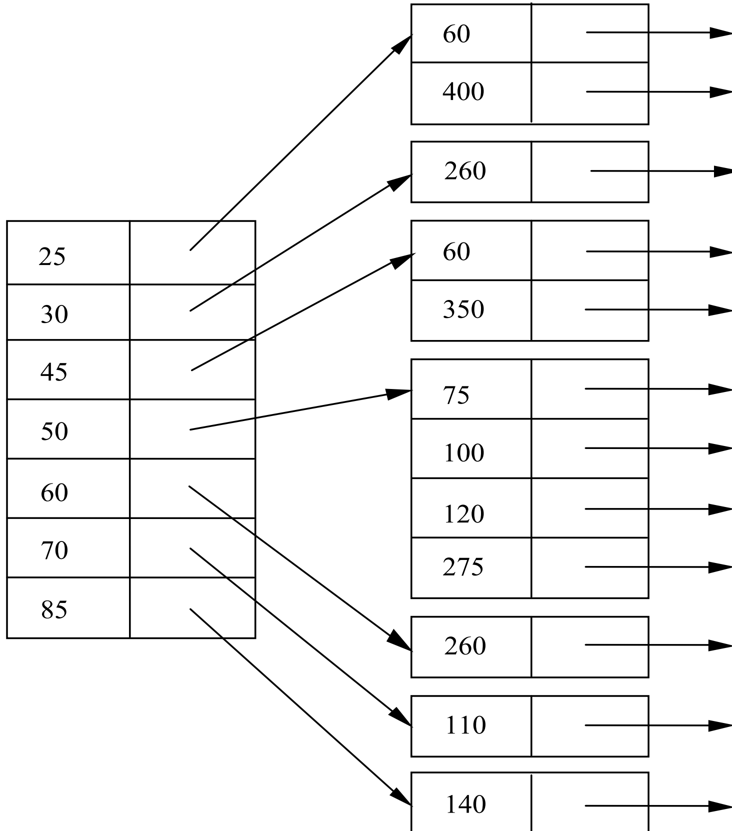


Рис. 88. Індекси з декількома ключами для двовимірного простору даних «вік-прибуток»

У правій частині рис. 88 розташовано сім індексів, які забезпечують доступ безпосередньо до записів даних.

Наприклад, звертаючись до (кореневого) індексу з ключем «*вік*» і стежачи за адресним значенням покажчика, пов'язаного зі значенням координати віку, яка становить 50, ми потрапимо до підпорядкованого індексу з ключем «*прибуток*», чотири елементи якого представляють значення координати прибутку точок, що задовольняють умову *вік*=50. (І знову ми не акцентуємо увагу на особливостях реалізації подібних індексів – достатньо наявності пар виду «ключ-значення»). Покажчики в сегментах індексу, які відповідають кожному з чотирьох ключових значень прибутку (75, 100, 120 і 275), адресують конкретні записи даних (наприклад, покажчик, що стосується значення ключової координати *прибуток*=100, дає змогу знайти запис, що представляє особу віком 50 років, яка володіє прибутком у розмірі 100 тисяч доларів).

У структурах індексів з декількома ключами індекси другого або вищого рівнів підпорядкованості можуть мати дуже малий об'єм. Наприклад, чотири з семи індексів другого рівня, зображених на рис. 88, містять по одному елементу. Такі індекси доцільно реалізувати як прості таблиці, упаковані в один спільний блок.

19.2. Оцінки ефективності структур індексів з декількома ключами

Розглянемо характеристики поведінки індексів з декількома ключами під час використання для різних категорій запитів у контексті багатовимірних даних. Ми зосередимо увагу на структурах з двома атрибутами, хоча всі висновки легко узагальнити на випадок довільної кількості атрибутів.

Запити до даних з рівністю окремих координат. Якщо в запиті задано значення атрибута, який відповідає кореневому індексу, то індексна структура проявляє високу ефективність. Кореневий індекс використовують для відшукання одного із підпорядкованих індексів, який, у свою чергу, спроваджує до шуканої точки. Наприклад, якщо кореневий індекс оформлено як *B*-дерево, то для переходу до визначеного підпорядкованого індексу необхідно виконати дві або три операції дискового

введення/виведення, а для перегляду знайденого часткового індексу і завантаження шуканого запису даних – ще декілька подібних операцій. З іншого боку, якщо значення атрибута, який є ключем пошуку в кореневому індексі, в запиті не згадано, то системі доведеться звернутися до кожного підпорядкованого індексу, що, ймовірно, потребуватиме немалих затрат часу та обчислювальних ресурсів.

Запити у діапазонах значень. Під час виконання запитів у діапазонах значень індекс з декількома ключами демонструє добротні показники продуктивності – звісно, якщо часткові індекси, що належать до нього (наприклад, *B*-дерева або «звичайні» індекси для послідовних файлів), у свою чергу, підтримують запити у діапазонах зміни значень відповідних атрибутів. Діапазони значень ключа пошуку кореневого індексу в сукупності з діапазонами зміни ключів підпорядкованих індексів дають змогу виявити всі шукані точки простору.

Приклад 85. Звернемося до структури індексу з декількома ключами (рис. 88) і припустимо, що необхідно виконати запит у таких діапазонах значень: 35≤*вік*<55 і 100≤*прибуток*<120. Під час перегляду елементів кореневого індексу вдається виявити два ключі (45 і 50), які задовольняють заданому діапазону зміни значень атрибута «*вік*». Пов'язані зі знайденими ключами покажчики адресують два підпорядковані індекси для ключа пошуку «*прибуток*». Індекс для координати *вік*=45 не містить значень прибутку в заданому діапазоні, проте індекс, що відповідає умові *вік*=50, володіє двома відповідними елементами – 100 і 120. Отже, результатом пошуку виявляються дві точки з координатами (50,100) і (50,120).

Пошук найближчих сусідніх об'єктів. Пошук об'єктів, найближчих до заданого, за допомогою індексу з декількома ключами виконується за тією ж стратегією, що й майже всі інші запити, які розглянуто у цьому розділі. Для відшукання найближчої «сусідки» точки (x_0, y_0) обирають віддаль d, утворюють область з центром у (x_0, y_0), яка, як очікується, містить щонайменше одну точку. Потім виконують запит у діапазонах $x_0 - d \leq x \leq x_0 + d$ і $y_0-d \leq y \leq y_0+d$ значень координат. Якщо запит є безрезультатним

або повертає точку, розташовану на віддалі від (x_0,y_0), більшій ніж d, то треба збільшити значення d і повторити процедуру пошуку.

19.3. Використання kd-дерев для побудови індексів

Структуру, яка слугує для представлення інформації в ОП й узагальнює модель бінарних дерев пошуку на випадок багатовимірних даних, називають *kd-деревом (k-dimensional search tree)*. Розглянемо, передусім, головні характеристики схеми *kd*-дерева, а потім розповімо про особливості її реалізації в контексті блокових вторинних сховищ даних. Адже *kd*-дерево – це бінарне дерево, з кореневою і проміжними вершинами якого асоційовані атрибут *a*, що представляє деяку розмірність даних, і певне значення *v* цього атрибута, яке поділяє множину точок даних на дві підмножини: точкам однієї підмножини відповідають значення атрибута *a*, менші від *v*, а точкам другої – значення *a*, рівні або більші від *v*. Атрибути в межах одного рівня дерева однакові, а в сусідніх рівнях – різні: рухаючись від кореневої вершини дерева «вниз», рівнями проміжних вершин, усі атрибути розмірності циклічно заміщають один одного.

У класичному *kd*-дереві, як і в дереві бінарного пошуку, з вершинами пов'язані точки даних. Однак ми дещо змінимо схему таким способом, щоб адаптувати її до особливостей моделі блочного сховища даних:

1) кожній проміжній вершині відповідатиме деякий атрибут, конкретне «роздільне» значення цього атрибута і покажчики-дуги, які адресують ліву і праву дочірні вершини;
2) листи дерева представлятимуть блоки, здатні містити таку кількість записів, яку зумовлено фізичним об'ємом блока.

Приклад 86. На рис. 89 наведено *kd*-дерево, яке представляє дванадцять точок-записів із прикладу, що надає інформацію про колекціонерів коштовностей. Для спрощення місткість блока обмежено двома записами. Подібні блоки разом з їхнім вмістом зображено як прямокутні вершини-листи. Проміжні вершини мають форму овалів, в яких подано назви одного з двох атрибутів-розмірностей («прибуток» або «вік») і відповідне значення цього атрибута. Наприклад, коренева вершина розділяє множину точок

на дві підмножини залежно від значення атрибута «прибуток»: усі точки-записи, які задовольняють умову *прибуток*<150, належать до лівого піддерева, а решта точок (які відповідають умові *прибуток* ≥150) – до правого.

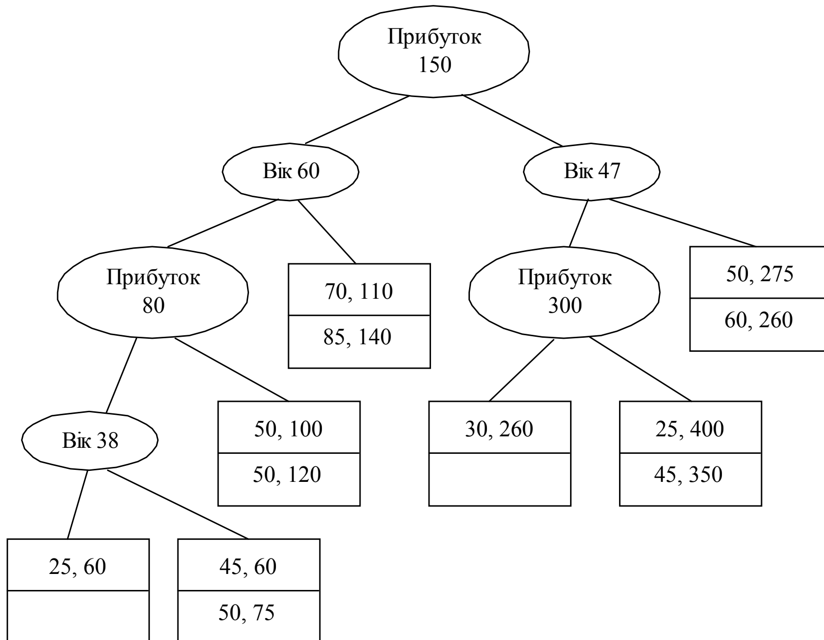


Рис. 89. Приклад kd-дерева

Вершини другого рівня використовують для подальшого розбиття підмножин точок відповідно до значень атрибута *«вік»*. Наприклад, ліва дочірня вершина розділяє множину точок залежно від виконання умови *вік*<60: координати точок, описані лівим піддеревом, задовольняють вимогам *вік*<60 і *прибуток*<150, а точки, які належать до правого піддерева, відповідають умовам *вік*≥60 і *прибуток*<150. На рис. 90 зображено схему розбиття простору точок по блоках-листах, яка відповідає графу, наведеному на рис. 89. Наприклад, горизонтальна пряма з координатою *прибуток*=150 представляє кореневу вершину дерева. Область

простору під прямою розбивається вертикальною лінією *вік*=60 (що відповідає лівій дочірній вершині), а область над прямою – вертикальною лінією *вік*=47 (що представляє праву дочірню вершину).

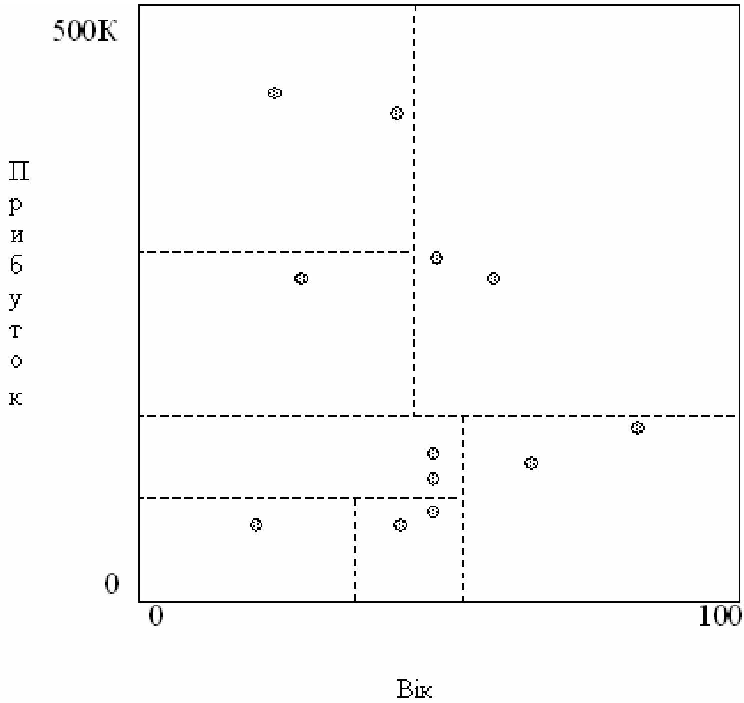


Рис. 90. Розбиття простору точок за посередництвом kd-дерева, зображеного на рис. 89

19.4. Операції з kd-деревами

Процедура відшукання точки (запису) за заданими значеннями координат-компонентів виконується так само, як і в бінарному дереві пошуку: на кожній проміжній вершині ухвалюють рішення щодо того, якою гілкою просуватись далі, аж до досягнення єдиної вершини-листа, яка представляє блок із шуканими даними.

Операція вставки точки (запису) передбачає необхідність звернення до процедури пошуку відповідного блока, яку описано вище. Якщо блок, що відповідає знайденому листу дерева, має достатньо вільного простору, то запис надходить у блок і операція успішно завершується. У протилежному випадку на місці вершини-листа створюється нова проміжна вершина, яка відповідатиме певному атрибутові і «розділюючому» значенню цього атрибута[1], блок листа розділяється на два нових і його вміст розподіляється між блоками відповідно до обраних значень атрибута.

Приклад 87. Припустимо, що за допомогою *kd*-дерева у базу даних з інформацією про покупців коштовностей необхідно занести відомості про 35-річну особу, яка володіє річним прибутком розміром 500 тисяч доларів. Умова, яку задано кореневою вершиною (*прибуток*=150), змушує пройти до правої дочірньої вершини (*прибуток*≥150). Тут значення 35 порівнюють з «розділюючим» значенням віку (47): оскільки 35<47, то доведеться перейти до лівої дочірньої вершини. Вершини третього рівня дерева знову відповідають координаті «прибуток». Оскільки 500>300, то ми досягнули листа, який містить точки (25,400) і (45,300), – сюди ж треба помістити і точку (35,500). Однак блок цілковито заповнений, отож його необхідно розділити. Вершини четвертого рівня дерева повинні відповідати координаті «вік». Ми обиратимемо таке значення координати, яке даватиме змогу розподілити записи у два нові блоки настільки рівномірно, наскільки це можливо. Вдалим, як нам здається, було б значення 35, отож лист замінюється проміжною вершиною з умовою *вік*=35. У лівий лист-блок переноситься точка (25,400), а в правий – точки (35,500) і (45,350). Результат операції вставки проілюстровано на рис. 91.

Запити до даних із рівністю окремих координат. Якщо значення деяких атрибутів задано, то це дає змогу переміщатися деревом від рівня, якому відповідає один із атрибутів з відомим

[1] Можлива така ситуація: усі точки, які належать блокові, мають однакові значення певної координати, отож розділити блок за значеннями цієї координати не вдається. У подібній ситуації можна спробувати обрати іншу координату або створити блок переповнення

значенням. Якщо ж біжуча вершина належить до такого рівня, для якого значення відповідного атрибута не визначене, то доведеться здійснити перевірку умов обох дочірніх вершин. Для прикладу розглянемо, як обробляється запит, який передбачає відшукання усіх точок з координатою *вік*=50 за допомогою *kd*-дерева, представленого на рис. 89.

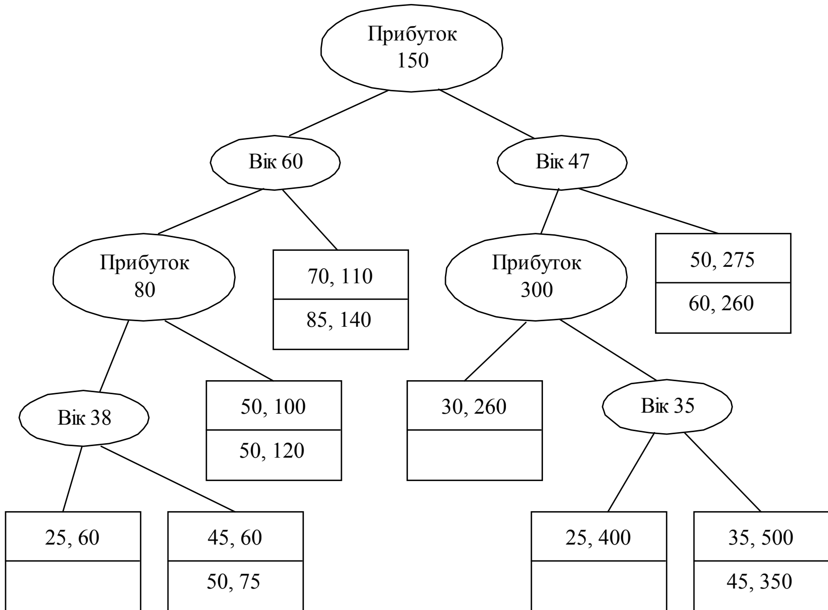


Рис. 91. Схематичне зображення kd-дерева (див. рис. 89) після вставки точки (35,500)

Оскільки кореневій вершині відповідає координата «прибуток», а не «вік», то необхідно проаналізувати умови обох вершин, дочірніх до кореневої: значенню *вік*=50 задовольняють ліва гілка лівої вершини і права гілка правої. Далі необхідно прямувати обома обраними гілками, перевіряючи на кожному рівні усі альтернативи, аж до досягнення всіх вершин-листів, що містять точки, координати яких задовольняють задану умову *вік*=50 (загалом таких точок чотири – (50,75), (50,100), (50,120) і (50,275)).

Якщо *kd*-дерево є збалансоване, то кількість вершин-листів, які перевіряються, буде не більшою $\sqrt{n}$ від загального числа *n* листів.

Запити в діапазонах значень. Іноді заданий діапазон є таким, що дає змогу переміщатися від біжучої вершини *kd*-дерева безпосередньо до однієї дочірньої вершини. Якщо ж «розділююче» значення атрибута біжучої вершини перебуває всередині діапазону, то необхідно розглядати обидві дочірні вершини. Знову звернемось до *kd*-дерева, зображеного на рис. 89, і припустимо, що запит передбачає пошук точок, координати яких належать діапазонам 35≤ *вік* ≤55 і 100≤ *прибуток* ≤200. Діапазон значень прибутку містить «розділююче» значення (150) атрибута «прибуток» кореневої вершини, отож звернемо увагу на обидві вершини, дочірні до кореневої. Розпочнемо з лівої дочірньої вершини. Заданий діапазон значень віку розташований лівіше від «розділюючої» величини (60) координати «вік», що дає змогу переміститися до лівої дочірньої вершини *прибуток*=80. Оскільки заданий діапазон прибутку розташований цілком правіше, то ми переходимо до блока-листа з точками-записами (50,100) і (50,120). Координати точок задовольняють обидва діапазони. Тепер повернемося до правої дочірньої вершини щодо кореневої. «Розділююче» значення *вік*=47 перебуває всередині заданого діапазону значень «вік», отож необхідно проаналізувати обидві дочірні вершини. Перемістившись до вершини *прибуток*=300, ми можемо рухатись тільки вліво: координати точки (30,260) не належать жодному з діапазонів. Правою дочірньою вершиною щодо вершини *вік*=47 є лист з двома записами-точками, проте їхні координати не відповідають діапазонам, оголошеним в умовах запиту.

Пошук найближчих сусідніх об'єктів. Обробка запитів, які передбачають відшукання за допомогою *kd*-дерева точок, найближчих до заданої, виконується за тією ж схемою, про яку йшлося у пункті 19.2: задача зводиться до виконання запиту в певному діапазоні; якщо результат не отримано, то діапазон розширюється.

19.5. Покращення структури kd-дерева для даних у вторинних сховищах

Нехай необхідно створити індекс для файла даних за допомогою *kd*-дерева з *n* вершинами-листами. Середня довжина шляху від кореня до деякого листа *kd*-дерева, як і в будь-якому бінарному дереві, оцінюється величиною $\log_2 n$. Якщо вершині відповідає один блок, то в процесі переміщення деревом необхідно виконувати по одній операції дискового введення/виведення для зчитування даних кожної вершини. Наприклад, при *n*=1000 типова процедура пошуку запису в *kd*-дереві вимагатиме виконання майже 10 дискових операцій, що значно перевищує трудомісткість (2-3 операції) аналогічної процедури пошуку в значно більшому *B*-деревоподібному файлі. Окрім того, оскільки проміжні вершини *kd*-дерева – це порівняно малі фрагменти інформації, то блоки дерева, здебільшого, виявляються не заповненими.

Задачі зменшення середньої довжини шляху в дереві та оптимізації рівнів заповнення блоків є взаємозалежними – домогтися обох цілей водночас не вдається. Однак ми наведемо два підходи, які збільшують продуктивність системи.

Багатократне галуження. Кореневу і проміжну вершини *kd*-дерева можна трактувати так, як і вершини *B*-дерева, що містять підмножини пар даних виду *«ключ-покажчик»*. Якщо вершині поставити у відповідність не одне, а *n* ключових значень, то це дало б змогу розділити значення атрибута *a* на *n*+1 діапазонів і звертатися тільки до одного з *n*+2 піддерев, яке відповідає підмножині точок зі значеннями атрибута *a* з потрібного діапазону.

Зі спробою реорганізації вершин з метою рівномірного розподілу записів і забезпечення балансу (як, зазвичай, відбувається з *B*-деревами) насправді можуть виникнути деякі проблеми. Припустимо, для прикладу, що розділенню підлягає вершина kd-дерева, що відповідає атрибутові «вік», і необхідно здійснити злиття її дочірніх вершин, кожна з яких представляє деяке «розділююче» значення атрибута «прибуток». Ми не можемо створити нову вершину шляхом об'єднання діапазонів атрибута «прибуток» дочірніх вершин, оскільки ці діапазони, зазвичай, перекриваються. Зверніть увагу, наскільки б спростилась задача,

якщо б (як у *B*-дереві) обидві дочірні вершини слугували подальшим уточненням діапазону того ж атрибута «вік», який відповідає батьківській вершині.

Групування проміжних вершин по блоках. Можливий також інший підхід, за якого умова належності вершині тільки двох дочірніх вершин справедлива, проте інформація з описом декількох вершин упаковується в один дисковий блок. Щоб мінімізувати кількість операцій дискового введення/виведення, що виконується в процесі пересування від кореневої вершини дерева до певного листа, доцільно зробити спробу збереження даних цієї вершини і всіх її «спадкоємців» аж до певного рівня в межах одного блока. У цьому випадку, завантажуючи блок, що відповідає вершині, ми можемо бути впевнені, що блок містить дані ще декількох рівнів вершин, дочірніх до біжучої, і для їхнього перегляду додаткові дискові операції не потрібні. Припустимо, для прикладу, що об'єм блока достатній для розміщення інформації про три (овальні) вершини: наприклад, кореневу і дві дочірні вершини дерева, представленого на рис. 89, можна описати за допомогою одного блока, вершину *прибуток*=80 з її лівою дочірньою вершиною розташувати у другий блок, а вершину *прибуток*=300 – у третій (імовірно, другий і третій блоки можна об'єднати, хоча подібна операція спричинить до чималих затрат під час необхідності модифікацій дерева). Отже, для відшукання запису (25,60), наприклад, необхідно звернутися лише до двох блоків із відомостями про кореневу і проміжні вершини, хоча загальна кількість таких вершин, які підлягають перегляду у процесі пошуку, становить чотири.

19.6. Дерева квадрантів

У *дереві квадрантів* (*quad tree*) кожна проміжна вершина відповідає певному квадранту простору даних – прямокутній двовимірній області або *k-вимірному* паралелепіпеду, якщо кількість вимірів дорівнює *k*. Як і під час розгляду решти структур даних, що описуються в цьому розділі, ми зосередимо увагу на випадку двох вимірів. Якщо кількість точок, які належать квадранту, нижча від припустимого максимуму, то квадрант треба сприймати як вершину-лист дерева, що представляється

відповідним блоком. Якщо ж, навпаки, кількість точок надто велика, то квадрант інтерпретується як проміжна (на початку – коренева) вершина з чотирма дочірніми вершинами, що відповідають частковим квадрантам.

Приклад 88. На рис. 92 зображено множину точок з інформацією про покупців коштовностей; точки розподілені в областях-квадрантах, які відповідають вершинам дерева. Щоб спростити обчислення, ми обмежили діапазон зміни значень координати «прибуток» до 0-400 замість звичайного 0-500, який використовувався в попередніх прикладах. Як і раніше, вважатимемо, що блок має місткість, достатню для зберігання двох записів.

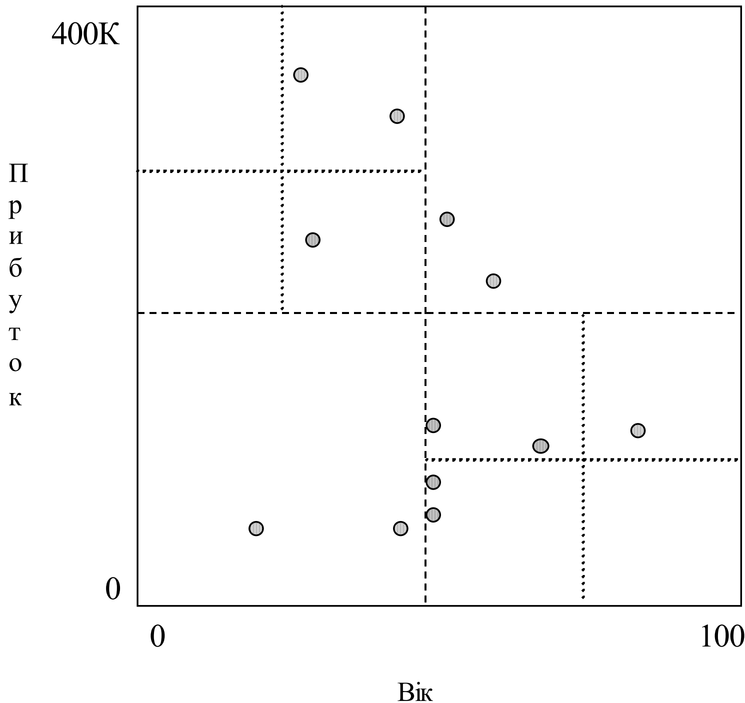


Рис. 92. Фрагмент простору даних, розбитий на квадранти

На рис. 93 зображено дерево квадрантів як таке. Квадранти і відповідні дочірні вершини позначено назвами географічних напрямів: так, наприклад, абревіатура «ПдЗ» означає «південний захід», тобто йдеться про нижній лівий частковий квадрант біжучого квадранту (решта скорочень – «ПдС», «ПнС» і «ПнЗ» – розшифровуються як «південний схід», «північний схід» і «північний захід», відповідно). Порядок слідування дочірніх вершин-квадрантів зберігається таким, який задано для кореневої вершини (у нашому випадку географічні напрями перераховуються проти годинникової стрілки, починаючи з південного заходу). Кожна проміжна вершина (такі вершини позначено овалами) задає координати центру області біжучого квадранта.

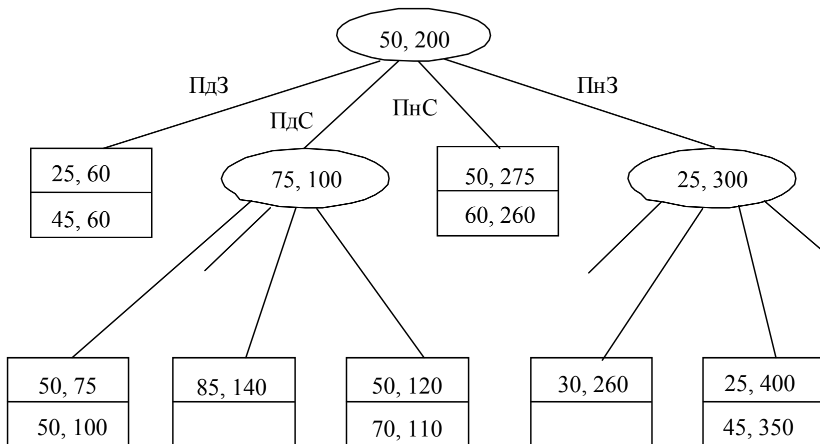


Рис. 93. Дерево квадрантів для фрагмента простору даних, зображеного на рис. 92

Оскільки загальна кількість точок, охоплених фрагментом простору, що розглядається, дорівнює 12, однак кожен блок може вмістити не більше, ніж дві точки, то простір треба розбити на квадранти (останні позначено на рис. 92 штриховими лініями). Два з отриманих квадрантів – південно-західний і північно-східний – містять тільки по дві точки, отож відповідають блокам-листам дерева і не підлягають подальшому розщепленню.

На кожен з двох інших квадрантів припадає більше, ніж дві точки. Отож квадранти піддаються подальшому розділенню. Часткові квадранти позначено на рис. 92 точковими лініями. Кожен з них містить не більше, ніж дві точки, отож подальше розділення не потрібне.

Оскільки коренева і кожна проміжна вершина *k*-вимірного дерева квадрантів володіють 2^k дочірніми вершинами, то існують певні значення *k*, за яких інформація вершини «вписується» у дисковий блок. Якщо, наприклад, блок здатен містити $128=2^7$ покажчиків, тоді *k*=7 – «вдала» кількість розмірностей. У двовимірному випадку, однак, ситуація не набагато краща, ніж при використанні *kd*-дерев: коренева і проміжні вершини містять всього по чотири дочірні вершини. Окрім того, якщо в *kd*-дереві вибір «розділюючого» значення атрибута вершини нічим не обмежено, то вершина в дереві квадрантів завжди представляє центр біжучої області простору, яка підлягає розбиттю на часткові квадранти, якими більше або (що значно гірше) менше рівномірно розподіляються точки області. Якщо кількість розмірностей велика, то варто очікувати виникнення чималої кількості «нульових» покажчиків, що адресують пусті квадранти. Звісно, працюючи з багатовимірним деревом квадрантів, можна подбати про його економічніше представлення за рахунок вилучення нульових покажчиків і вжиття заходів, які даватимуть змогу розрізняти, якому квадранту відповідає той чи інший покажчик.

Не зупинятимемось на деталях, які стосуються виконання стандартних операцій (подібних до тих, які розглянуто в пункті 19.4 стосовно *kd*-дерев) з деревами квадрантів: висновки, отримані для *kd*-дерев, цілком застосовні і до моделі дерев квадрантів.

19.7. R-дерева

R-дерево (*R-tree*, або *region tree – дерево областей*) – це структура даних, яка наслідує чимало властивостей моделі *B*-дерева у застосуванні до багатовимірної інформації. Нагадаємо, що вершина *B*-дерева містить підмножину значень ключа, які ділять числову вісь на сегменти так, як зображено на рис. 94. Кожна точка осі може належати тільки одному сегментові. Відшукати задану точку в *B*-дереві не важко, оскільки точка

розташована на осі, представленій вершиною *B*-дерева, і ми можемо визначити єдиний сегмент (дочірню вершину), який дає змогу звузити область пошуку і, зрештою, або знайти точку, або переконатися, що її не існує.

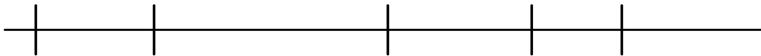


Рис. 94. Вершина *B*-дерева розподіляє значення ключа пошуку по осі і розділяє останню на сегменти

R-дерево, з іншого боку, представляє інформацію у вигляді *2*- або *k*-вимірних *областей даних*. Коренева (або проміжна) вершина *R*-дерева відповідає деякій *внутрішній області*, або просто «області», яка, зазвичай, не є областю даних. Область може мати будь-яку форму, хоча на практиці, зазвичай, використовують області прямокутних чи інших простих форм. Вершина *R*-дерева замість ключових значень містить *підобласті*, які описують вміст дочірніх вершин. На рис. 95 зображено вершину *R*-дерева, асоційовану з великим прямокутником, позначеним суцільною лінією. Внутрішні прямокутники, обмежені відрізками точкових ліній, представляють підобласті, що відповідають чотирьом дочірнім вершинам. Зверніть увагу, що підобласті не вкривають область цілковито – це нормально. Окрім того, області можуть перетинатися, хоча ступінь подібних накладок бажано зменшувати.

19.8. Операції з R-деревами

Типовим прикладом запиту, під час обробки якого *R*-дерево виявляється особливо корисним, може слугувати запит, що передбачає визначення місцезнаходження об’єкта: задається деяка точка *P* і вимагаються відомості про належність точки тим чи іншим областям. Спочатку розглядається коренева вершина дерева, з якою пов’язана крупна область. Потім перевіряється кожна підобласть кореневої вершини і встановлюється, які вершини, дочірні до кореневої й асоційовані з внутрішніми областями, містять точку *P*. Таких областей може бути одна або декілька (або може не бути зовсім). Якщо областей немає, то операція завершується: *P* не належить жодній області даних. Якщо, навпаки,

існує, в крайньому випадку, одна внутрішня область, що охоплює точку P, то необхідно здійснити рекурсивний перегляд вершин, що відповідають кожній із областей, яким належить P. Після досягнення однієї або декількох вершин-листів ми виявимо шукані області даних у формі самих записів або покажчиків на такі записи.

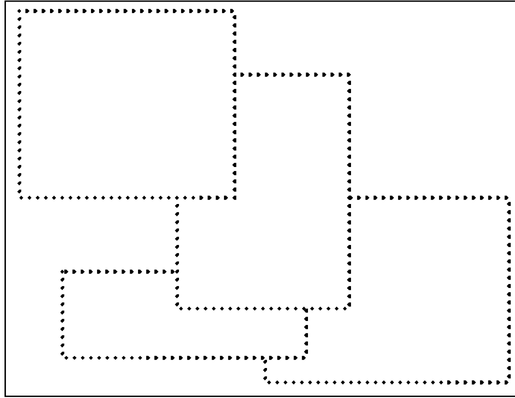


Рис. 95. Область вершини R-дерева, яка охоплює підобласті дочірніх вершин

Під час вставки нової області R_m у R-дерево необхідно звернутися до кореневої вершини і спробувати знайти деяку підобласть, придатну для розміщення в ній області R_m. Якщо подібних підобластей декілька, то обирають одну область, переходять до відповідної їй дочірньої вершини дерева і повторюють процес пошуку в цій підобласті. Якщо підобластей, здатних містити R_m, немає, то одну із підобластей необхідно розширити. Рішення про те, яку з підобластей належить розширити, часто не очевидне. Інтуїція підказує, що вплив на існуючі області необхідно звести до мінімуму. Отож доцільно визначити, яка із підобластей дочірніх вершин вимагає найменшого збільшення, достатнього для розміщення в ній області R_m, і, відповідно, змінити межі визначеної області і внести рекурсивні зміни у структуру дерева.

З часом ми досягнемо вершини-листа, куди і необхідно вставити область R_m. Якщо в області, яка відповідає листу, для R_m немає

місця, то лист необхідно розділити; як саме – це особливе питання. Хотілося б, щоб обидві підобласті виявились такими малими, наскільки це можливо, проте все ж покривали всі області даних вихідного листа. Під час розділення листа область і покажчик на лист, розташовані у вершині, батьківській щодо листа, заміняються парою областей і покажчиків, що відповідають двом новим листам. Якщо блок батьківської вершини володіє достатнім вільним простором, то процедура успішно завершується. У протилежному випадку, як і під час виконання аналогічної операції в B-дереві, рекурсивним чином розділюються вершини рівнів, все ближчих до кореневої вершини.

Приклад 89. Розглянемо операцію вставки нового об’єкта у фрагмент двовимірного простору, зображений на рис. 77. Припустимо, що блок листа R-дерева дає змогу зберігати опис шести областей. Вважатимемо також, що шість областей-об’єктів (див. рис. 77) представлені в одному листі, область якого зображено на рис. 96 прямокутником, обмеженим відрізками суцільної лінії.

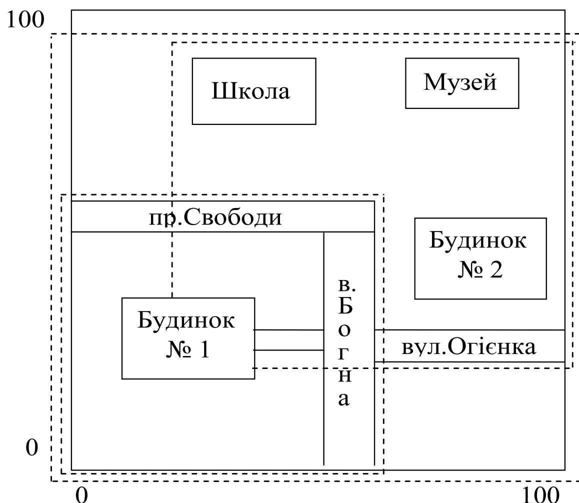


Рис. 96. Розділення множин об’єктів

Припустимо також, що поряд з об’єктами, зображеними на карті, відкрили новий музей. Оскільки сьома область даних не може бути збереженою в тому ж листі дерева, то лист підлягає розділенню: чотири області належать до одного, щойно створеного листа, а три – до другого. Кількість можливих альтернатив розділення велика, проте ми обрали таку (на рис 96 підобласті позначено штриховими прямокутниками), яка мінімізує рівень перекриття областей і забезпечує рівномірний розподіл об’єктів між ними.

На рис. 97 зображено, як два нових листи заносяться в *R*-дерево. Батьківська вершина містить покажчики на обидва листи і пов’язані з ними значення координат нижнього лівого і верхнього правого кутів прямокутних областей, що описуються листами.

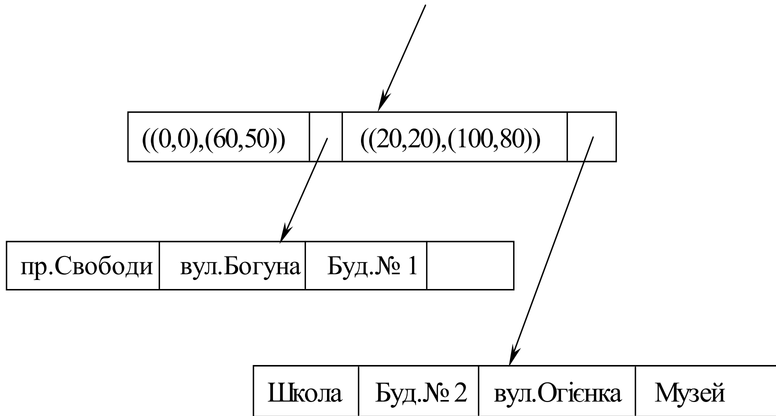


Рис. 97. Приклад R-дерева

Приклад 90. Припустимо, що у праву нижню частину карти на рис. 96 необхідно внести інформацію про будинок, який побудували. Координати нижнього лівого і верхнього правого кутів прямокутника «будинок №3» дорівнюють (70,5) і (80,15), відповідно. Оскільки об’єкт цілковито не поміщається в жодну з областей, що відповідають листам дерева, то необхідно вибрати область, яка підлягає розширенню. Якщо розширити нижню ліву область, що відповідає лівому листу *R*-дерева, наведеного на рис. 97, то площа

області збільшиться на 1 000 квадратних одиниць (горизонтальний розмір зросте на 20 одиниць). Якщо ж розширити праву верхню область (перемістити її нижню границю на 15 одиниць вниз), то це спричинить до збільшення площі області на 1200 квадратних одиниць. Ми оберемо перший варіант. Результат дії зображено на рис. 98. Опис області в кореневій вершині дерева на рис. 97 необхідно змінити з ((0,0),(60,50)) на ((0,0),(80,50)).

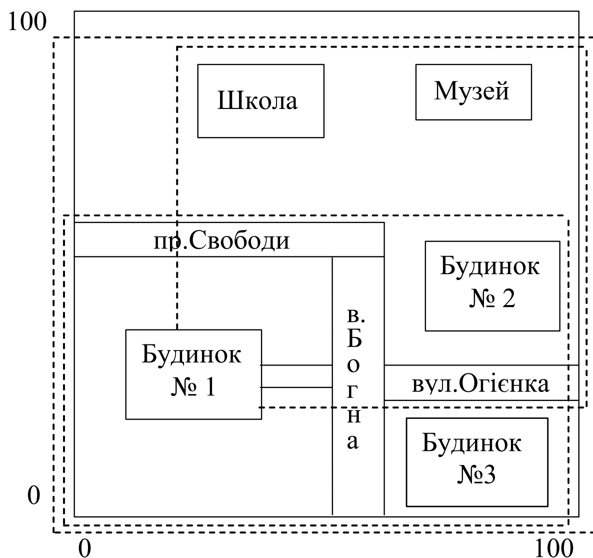


Рис. 98. Розширення області з метою включення нового об’єкта

Вправи для опрацювання

Вправа 61 Зобразіть структуру індексу з декількома ключами для даних, наведених на рис. 86, якщо порядок підпорядкованості індексів такий:

а) speed, ram;
б) ram, hd;
в) speed, ram, hd.

Вправа 62. Представте дані, зображені на рис. 86, у вигляді *kd*-дерева, вважаючи, що блок може зберігати не більше, ніж два записи, і для вершин кожного рівня виберіть «розділююче» значення атрибутів таке, яке забезпечує рівномірний розподіл даних, наскільки можливо, якщо атрибути відповідають рівням дерева від кореня в напрямі листів у вказаному нижче порядку:

а) speed і ram, почергово;
б) speed, ram і hd, почергово;
в) на кожному рівні обирається такий атрибут, який забезпечує найрівномірніший розподіл даних.

Вправа 63. Зобразіть варіанти дерева, представленого на рис. 91, після вставки точки (20, 110), а потім точки (40, 400).

Вправа 64. Представте дані, зображені на рис. 86, як дерево квадрантів з розмірностями speed і ram, вважаючи, що діапазони зміни значень атрибутів speed і ram дорівнюють 100-500 і 0-256, відповідно.

Вправа 65. Виконайте завдання з вправи 64, додавши третій вимір hd з діапазоном значень, який дорівнює 0-32.

Вправа 66. Припустимо, що нова область, яка вставляється в *R*-дерево, зображене на рис. 98, може бути поміщена в підобласть, яка охоплює об’єкт «школа», або в підобласть, яка містить об’єкт «будинок №3». Вкажіть області, які отримують виграш від того, що нова область розміщується в підобласті з об’єктом «школа» (тобто подібний вибір мінімізує збільшення розмірів підобласті).

ДОДАТОК А. РЕЛЯЦІЙНА МОДЕЛЬ ПРЕДСТАВЛЕННЯ ДАНИХ

А.1. Основні поняття реляційної моделі

Реляційна модель передбачає єдиний спосіб представлення даних – у вигляді набору двовимірних таблиць, які називають *відношеннями* (*relations*). На рис. А.1 наведено приклад відношення *Movies*, яке призначене для зберігання інформації про елементи множини сутності *Movies* («кінофільми»).

Кожен рядок відношення Movies відповідає одній сутності «кінофільм», а кожен стовпець – одному із *атрибутів* множини сутностей *Movies*.

title	*year*	*length*	*filmType*
Star Wars	1977	124	color
Mighty Ducks	1991	104	color
Wayne′s World	1992	95	color

Рис. А.1. Відношення Movies

Атрибути. У верхній частині таблиці-відношення задається перелік найменувань атрибутів: *title, year, length, filmType*. Атрибути відношення виконують функцію найменування його стовпців і змістовно описують значення і призначення елементів даних, розташованих у відповідних комірках.

Схеми. Найменування відношення та атрибутів цього відношення в сукупності називають *схемою* (*schema*) відношення. Наприклад, схема відношення *Movies* може виглядати так:

Movies (title, year, length, film Type)

Проект бази даних, виконаний у рамках реляційної моделі, містить одну або декілька схем відношень. Набір схем відношень називають реляційною схемою бази даних, або просто *схемою бази даних (database schema)*.

Кортежі. Рядки відношення, окрім того рядка, який містить назву атрибутів, називають *кортежами* (*tuples*). Кортеж містить по одному *компоненту* для кожного атрибута відношення. Якщо необхідно описати кортеж окремо від відношення, то його задають так:

(Star Wars, 1977, 124, color).

Домени. Одна з вимог реляційної моделі полягає в тому, що кожен компонент кортежу повинен бути атомарним, тобто належати до деякого базового типу, такого як цілочисловий або рядковий. Як значення компонентів кортежу не можна використовувати записи, множини, списки, масиви або інші об'єкти, які допускають природне розбиття на дрібніші елементи.

Окрім того, вважають, що з кожним атрибутом відношення асоціюється певний *домен* (*domain*), тобто деякий базовий тип. Значення компонентів всіх кортежів повинні належати відповідним доменам, які визначаються кожним з атрибутів відношення. Тобто перші компоненти всіх кортежів відношення *Movies* повинні бути рядками, другі і треті – цілими числами, а четверті повинні містити одну з двох допустимих рядкових констант, color або black and white. Домени є частиною схеми відношення.

Форми представлення відношень. Відношення – це множини (але не впорядковані списки) кортежів. Порядок, в якому кортежі перечислюються у рамках відношення, є несуттєвим. Цілком можливо представити кортежі відношення *Movies* довільним із шести різних способів. Це ніяк не вплине на його зміст. Можна також змінювати і порядок атрибутів відношення.

Екземпляри відношення. Відношення, яке містить інформацію про кінофільми, за своєю природою не є статичним. Йому і більшості інших відношень з часом властиво змінюватися. Конкретну множину кортежів відношення називають *екземпляром відношення*. Традиційні системи баз даних, зазвичай, підтримують тільки одну версію будь-якого відношення: набір кортежів, які містяться у відношенні «в даний час». Такий екземпляр відношення називають біжучим екземпляром.

А.2. Ключі відношень

Множина виду $\{A_1, A_2,\ldots,A_n\}$, яка складається з одного або декількох атрибутів, є *ключем* відношення *R*, якщо виконуються наступні умови:

1. Атрибути $A_1, A_2,\ldots,A_n$ функціонально обумовлюють всю решту атрибутів відношення; ситуація, коли два різні кортежі R збігаються в усіх атрибутах $A_1, A_2,\ldots,A_n$ є неможливою.
2. Жодна з допустимих підмножин множини $\{A_1, A_2,\ldots,A_n\}$ атрибутів не є функціональним обумовленням решти атрибутів відношення, тобто *ключ мінімальний*.

Функціональна залежність (FD) для відношення *R* – це твердження наступного виду: «Якщо два кортежі відношення R збігаються в атрибутах $A_1,A_2,\ldots,A_n$, то вони збігатимуться і в іншому атрибуті, B»:

$$A_1, A_2,\ldots,A_n \rightarrow B$$

$A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$

Іноді для відношення може бути створено декілька ключів. У такій ситуації, зазвичай, одному з них відводиться роль *первинного ключа* (*primary key*). У комерційних системах баз даних вибір первинного ключа може впливати на певні властивості реалізації, наприклад, на те, як відношення зберігається на диску.

А.3. Індекси

Індекс (*index*) атрибута *A* деякого відношення *R* – це структура даних, яка дає змогу підвищити ефективність процедури пошуку деякого конкретного значення, що зберігається в компонентах атрибута *A*. Наявність індексу, зазвичай, здатна спричинити до суттєвого зменшення часу опрацювання запитів, у яких значення атрибута *A* порівнюється з константами за допомогою таких виразів, як A=3 або, навіть $A \leq 3$.

Вибір відповідних індексів належить до категорії задач, які роблять виклик проектувальнику бази даних і максимально визначають продуктивність останньої і навіть її подальшу долю. Під час вибору індексів необхідно брати до уваги два важливих чинники:

- Наявність індексу для певного атрибута спричинює до суттєвого підвищення швидкості опрацювання запитів, в яких задаються значення цього атрибута, і в деяких випадках дає змогу також пришвидшити операції з'єднання з участю атрибута.
- З іншого боку, наявність в базі даних будь-якого індексу, створеного для атрибутів відношення, здатна ускладнити і сповільнити процедури вставки, вилучення та оновлення кортежів відношення.

Вибір адекватних індексів – одна з найскладніших стадій процесу проектування, і для її успішного «проходження» бажано отримати достовірну відповідь на питання щодо номенклатури запитів та інших команд, з якими користувачі будуть звертатися до бази даних у майбутньому.

ДОДАТОК Б. АЛГЕБРА ДОДАВАННЯ ЗА МОДУЛЕМ 2

Для розуміння механізмів контролю парності корисно знати алгебричні закони, справедливі для операцій додавання за модулем 2, що застосовуються до бітових векторів. Оператор додавання за модулем 2 позначають символом $\oplus$. Як приклад можна навести співвідношення $1100 \oplus 1010 = 0110$. Нижче перелічено деякі з найуживаніших правил алгебри додавання за модулем 2:

- комутативний закон : $x \oplus y = y \oplus x$;
- асоціативний закон: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$;
- додавання довільного вектора з нульовим: $x \oplus \bar{O} = \bar{O} \oplus x = x$;
- власна інверсія: $x \oplus x = \bar{O}$; один із наслідків: якщо $x \oplus y = z$, тоді із $x \oplus x \oplus y = x \oplus z$ випливає $y = x \oplus z$.

ДОДАТОК В. ТИПИ ДАНИХ ВІДНОШЕНЬ У SQL

Розглянемо основні атомарні типи даних, які підтримуються більшістю СКБД (кожний атрибут будь-якого відношення повинен відповідати тому чи іншому типу даних).

1. *Рядки символів* постійної або змінної довжини. Тип CHAR(*n*) позначає рядок постійної довжини, який містить *n* символів. Тип VARCHAR(*n*) визначає рядок змінної довжини з кількістю символів від 1 до *n*.
2. *Рядки бітів* постійної або змінної довжини, в яких можуть зберігатися тільки ′0′ або ′1′. BIT(*n*) – постійної довжини, BIT VARYING(*n*) змінної довжини.
3. *Логічні значення:* BOOLEAN.
4. *Цілочислові значення*: INT; SHORTINT;
5. *Числові значення з плаваючою комою:* FLOAT (REAL); DOUBLE RRECISION; DECIMAL(*n,d*), *n* – кількість розрядів у цілій частині числа, *d* – у дробовій частині; NUMERIC;
6. *Значення дат і часу.* Значення типів DATE і TIME відповідно є символьними рядками, записаними у спеціальному форматі: ′1948-05-14′, ′15:00:03.1415′

СПИСОК ЛІТЕРАТУРИ

1. *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. - М.: Издательский дом «Вильямс», 2003. – 384 с.
2. *Гарсиа-Молина Г., Ульман Дж., Уидом Дж.* Системы баз данных. Полный курс. – М.: Издательский дом «Вильямс», 2003.– 1088 с.
3. *Дейт К.* Введение в системы баз данных. – М.: Издательский дом "Вильямс", 2001. – 1071 с.
4. *Інформаційні технології. Аналітичні матеріали.*– Львів, 2005. – [Цит.2005, 15 вересня]. – Доступний з :<http://it.ridne.net>.
5. *Кнут Д.Э.* Искусство программирования. Т.3. Сортировка и поиск. – М.: Издательский дом «Вильямс», 2000. – 822 с.
6. *Костів О.* Структури даних. Частина 1: Тексти лекцій. – Львів: Видавничий цент ЛНУ імені Івана Франка, 2000. – 56 с.
7. *Мартин Дж.* Организация баз данных в вычислительных системах. – М.: Мир, 1980. –662 с.
8. *Цегелик Г.Г.* Организация и поиск информации в базах данных. – Львов: Вища школа, 1987. – 176 с.
9. *Цегелик Г.Г.* Системы распределенных баз данных. – Львов: Свит, 1990. – 160 с.

ЗМІСТ

ВСТУП **3**
ПРИНЦИПИ ЗБЕРІГАННЯ ІНФОРМАЦІЇ **5**
Тема 1. Приклад вигаданої системи баз даних **5**
1.1. Особливості реалізації СКБД Straw 5
1.2. Як СКБД Straw виконує запити 7
1.3. Негативні моменти в системі Straw 8
Тема 2. Ієрархія пристроїв пам’яті **9**
2.1. Кеш-пам’ять 11
2.2. Оперативна пам’ять 11
2.3. Віртуальна пам’ять 11
2.4. Вторинні пристрої зберігання 12
2.5. Третинні пристрої зберігання 14
2.6. Енергозалежні та енергонезалежні пристрої пам’яті. 15
Тема 3. Диски **16**
3.1. Внутрішні механізми дискових накопичувачів 17
3.2. Контролер дисків 19
3.3. Загальні параметри дисків 19
3.4. Параметри доступу до даних на диску 22
3.5. Записування блоків 26
3.6. Модифікація вмісту блоків 27
Вправи для опрацювання 27
Тема 4. Використання вторинних пристроїв зберігання .28
4.1. Модель обчислень з функціями введення/виведення 28
4.2. Сортування даних у вторинних сховищах 30
4.3. Сортування злиттям 31
4.4. Сортування двофазним багатокомпонентним злиттям 33
4.5. Багатокомпонентне злиття і надзвичайно великі відношення 37
Вправи для опрацювання 38
Тема 5. Підвищення ефективності дискових операцій 39
5.1. Групування даних по циліндрах диска 40
5.2. Використання декількох дискових пристроїв 42
5.3. Створення дзеркальних копій дисків 44
5.4. Впорядкування дискових операцій і алгоритм ліфта. 44

5.5. Попереднє зчитування і крупномасштабна буферизація даних 48
5.6. Прийоми оптимізації дискових операцій: переваги і недоліки 50
Вправи для опрацювання 52
Тема 6. Відмови дискових пристроїв 53
6.1. Періодичні відмови 54
6.2. Контрольні суми 55
6.3. Стійкі сховища 56
6.4. Подолання наслідків помилок 57
Вправи для опрацювання 58
Тема 7. Відновлення даних за цілковитої відмови диска 58
7.1. Модель відмови дискових пристроїв 59
7.2. Дзеркальні диски як засіб резервування 60
7.3. Блоки парності 61
7.4. Масиви RAID рівня 5 65
7.5. Відновлення даних після відмови декількох дисків 66
Вправи для опрацювання 71
ПРЕДСТАВЛЕННЯ ЕЛЕМЕНТІВ ДАНИХ НА ВТОРИННИХ ПРИСТРОЯХ ЗБЕРІГАННЯ 74
Тема 8. Елементи даних і поля 74
8.1. Представлення елементів реляційних баз даних 75
8.2. Представлення об’єктів 76
8.3. Представлення елементів даних полями 77
Тема 9. Записи 80
9.1. Конструювання записів постійної довжини 81
9.2. Заголовки записів 83
9.3. Групування записів постійної довжини в блоки 84
Вправи для опрацювання 86
Тема 10. Представлення адрес записів і блоків 87
10.1. Системи «клієнт/сервер» 88
10.2. Логічні і структуровані адреси 90
10.3. Підміна покажчиків 92
10.4. Збереження блоків на диску 97
10.5. «Закріплені» записи і блоки 97
Вправи для опрацювання 99
Тема 11. Елементи даних і записи змінної довжини 100

11.1.Записи з полями змінної довжини 102
11.2. Записи з полями, які повторюються 103
11.3. Записи змінного формату 106
11.4. Записи крупного об'єму...108
11.5. Об'єкти BLOB...109
Вправи для опрацювання...110
Тема 12. Модифікація записів 111
12.1. Вставка 111
12.2. Вилучення 113
12.3. Оновлення 116
СТРУКТУРИ ІНДЕКСІВ 117
Тема 13. Індекси для послідовних файлів 118
13.1. Послідовні файли 118
13.2. Щільні індекси...119
13.3. Розріджені індекси 122
13.4. Багаторівневі індекси 123
13.5. Дублікати ключових значень 125
13.6. Керування індексами під час модифікації даних ...130
Вправи для опрацювання...139
Тема 14. Вторинні індекси 140
14.1. Проектування вторинних індексів 141
14.2. Використання вторинних індексів............................142
14.3. Додаткові рівні у вторинних індексах.....................144
14.4. Пошук документів і звернені індекси......................147
Вправи для опрацювання...153
Тема 15. *B*-дерева...154
15.1. Структура B-дерева..154
15.2. Використання B-дерев 159
15.3. Пошук у B-деревах...162
15.4. Запити у діапазонах значень.....................................163
15.5. Вставляння елементів у B-дерево 164
15.6. Вилучення елементів з B-дерева...............................168
15.7. Оцінки ефективності B-деревоподібних індексів ..173
Вправи для опрацювання...175
Тема 16. Геш-таблиці...176
16.1. Геш-таблиці для даних у вторинних сховищах......177
16.2. Вставляння записів у геш-таблицю 178

16.3. Вилучення записів з геш-таблиці............................179
16.4. Оцінка ефективності гешованих індексів 181
16.5. Геш-таблиці, що розширюються 182
16.6.Вставляння записів у геш-таблицю, що розширюється 184
16.7. Лінійні геш-таблиці...187
16.8. Вставка записів у лінійну геш-таблицю.................191
Вправи для опрацювання...195
ІНДЕКСИ ДЛЯ БАГАТОВИМІРНИХ ДАНИХ.........................197
Тема 17. Застосування моделі багатовимірних даних.....198
17.1. Географічні інформаційні системи.........................199
17.2. Куби даних 200
17.3. SQL-запити до багатовимірних даних....................201
17.4. Запити в діапазонах значень з використанням традиційних індексів 204
17.5. Пошук найближчих сусідніх об'єктів з використанням традиційних індексів 206
17.6. Інші обмеження традиційних структур індексів 208
17.7. Огляд структур індексів для багатовимірних даних 209
Вправи для опрацювання...210
Тема 18. Геш-подібні структури для багатовимірних даних 211
18.1. Сіткові файли 211
18.2. Пошук записів у сітковому файлі 212
18.3. Вставка записів у сітковий файл............................215
18.4. Оцінки ефективності структур сіткових файлів.....217
18.5. Роздільні геш-функції 221
18.6. Порівняння структур сіткових файлів і роздільних геш-таблиць..223
Вправи для опрацювання...224
Тема 19. Деревоподібні структури для багатовимірних даних 226
19.1. Індекси з декількома ключами 226
19.2. Оцінки ефективності структур індексів з декількома ключами 229
19.3. Використання kd-дерев для побудови індексів 231

19.4. Операції з kd-деревами 233
19.5. Покращення структури kd-дерева для даних у вторинних сховищах 237
19.6. Дерева квадрантів 238
19.7. R-дерева 241
19.8. Операції з R-деревами 242
Вправи для опрацювання 246
ДОДАТКИ 248
СПИСОК ЛІТЕРАТУРИ 253

Навчальне видання

Копитко Марія Федорівна

Основи інформаційних технологій

Тексти лекцій

Редактор *Лоїк І.М.*
Технічний редактор
Комп'ютерний набір і верстання

Підп. до друку   . .2007. Формат 60х84/16. Папір друк.
Друк на різогр. Гарнітура Times New Roman. Умовн. друк. Арк.. 8,8.
Обл..-вид. арк. 9,8. Тираж 300 прим. Зам.   .

Видавничий центр Львівського національного університету
імені Івана Франка. 79000, м.Львів, вул..Дорошенка, 41