1. Управління ризиками в проектній діяльності / О.М. Верес, А.В. Катренко, І.В. Рішняк, В.М. Чаплига // Інформаційні системи та мережі. Вісник Національного університету “Львівська політехніка”. – 2003. – № 489. – С.38–49. 2. Катренко А.В. Методи управління ризиками в IT-проектах / А.В. Катренко, І.В. Рішняк // Комп’ютерні науки та інформаційні технології (CSIT-2008): III міжнародна наук.-практ. конф., 25–27 вересня 2008 р.: тези доповіді – Львів, 2008. – С.245–247. 3. Катренко А.В. Імітаційне моделювання в управлінні проектними ризиками / А.В. Катренко, І.В. Рішняк // Сучасні засоби та технології розроблення інформаційних систем: Міжнародна науково-практична конференція, ХНЕУ, 20–21 листопада 2008 р.: тези доповіді – Харків, 2008. – С.75–76. 4. Модель управління проектними ризиками / І.В. Рішняк // Комп’ютерні системи проектування. Теорія і практика. Вісник Нац. ун-ту “Львівська політехніка”. – 2004. – № 522. – С.155–160. 5. Рішняк І.В. Структурна модель, основні складові та класифікація проектних ризиків / І.В. Рішняк // Інтелектуальні системи прийняття рішень та прикладні аспекти інформаційних технологій (ISDMIT-2007): наук.-практ. конф., 14–18 травня 2007 р.: тези доповіді – Євпаторія, 2007. – Т. 2. – С.214–217. 6. ДСТУ ISO 9000-2001. 7. Chapman C., Ward S., Project risk management: processes, techniques and insights. Chichester: John Wiley & Sons, 1997. 8. Kahkonen K., Integration of risk and opportunity thinking in projects, Fourth European Project Management Conference, PMI Europe 2001, London UK, 6–7 June 2001.

УДК 004.9

Т.М. Рудакевич, Г.Г. Цегелик
Львівський національний університет імені. Івана Франка,
кафедра математичного моделювання соціально-економічних процесів

РОЗШИРЕННЯ ФУНКЦІОНАЛЬНИХ МОЖЛИВОСТЕЙ СКБД POSTGRESQL ДЛЯ ОПТИМІЗАЦІЇ ПОШУКУ ІНФОРМАЦІЇ У ФАЙЛАХ БАЗ ДАНИХ

© Рудакевич Т.М., Цегелик Г.Г., 2010

Розглянута задача реалізації методу пошуку інформації у файлах БД, який враховує розподіл імовірностей звертання до записів, в СКБД PostgreSQL через розширення її функціональних можливостей. Наведено розв’язок підзадачі – реалізацію аналізатора статистики звертань до записів.

Ключові слова: БД, алгоритми пошуку, розширення функцій СКБД.

The PostreSQL DBMS was extended by statistics analyzer realization. It is a part of search method that is based on allocation of entry’s selection probabilities.

Keywords: Database Management System, search algorithm, probability theory.

Вступ

У світі математики та інформатики завжди існує тенденція до прогресу та руху вперед, відкриття якихось нових закономірностей, оптимізації алгоритмів та впровадження їх у життя.

Оскільки основу сучасних інформаційних технологій становлять бази даних (БД) і системи керування базами даних (СКБД), то удосконалення технології опрацювання інформації з використанням концепції БД передбачає, передусім, вирішення проблеми оптимальної організації та пошуку інформації у файлах баз даних, що, своєю чергою, забезпечує доступ користувачів до інформації БД за мінімально допустимий час.

Враховуючи те, що в багатьох системах опрацювання інформації типовими є випадки нерівномірного розподілу ймовірностей звертання до записів файлів, то актуальною сьогодні є

задача проектування СКБД, у яких, залежно від закону розподілу ймовірностей звертання до записів, для пошуку інформації використовувався б найефективніший метод.

Огляд літературних джерел

Сьогодні існує низка методів пошуку інформації у файлах БД [1–4]. У [5] досліджена ефективність цих методів для різних законів розподілу ймовірностей звертання до записів (рівномірного, «бінарного», Зіпфа і узагальненого, частковим випадком якого є розподіл, що наближено задовольняє правило «80-20») Реалізація методів пошуку у комерційних СКБД, як звичайно, є прихованою, а із описів самих систем не зрозуміло, наскільки вони враховують розподіл ймовірностей звертання до записів. Методи пошуку, розглянуті в [1–4], ніяк не враховують розподілу імовірностей звертання до записів. Тому в [6] запропоновано метод, який істотно враховує розподіл імовірностей, і є значно ефективнішим за відомі методи. У зв’язку з цим постає задача розширення функцій СКБД, які дали б можливість здійснювати пошук інформації у файлах БД, враховуючи розподіл ймовірностей звертання до записів. Цей розподіл можна отримати на основі статистичного матеріалу, що зберігатиметься та нагромаджуватиметься упродовж існування бази даних.

Постановка задачі

Розглянемо задачу реалізації методу пошуку інформації у файлах БД, який враховує розподіл імовірностей звертання до записів, в СКБД PostgreSQL через розширення її функціональних можливостей. Ця задача розпадається на такі три підзадачі:

1. Написання коду, який би накопичував статистичні дані, а саме: кількість звертань до конкретних рядочків таблиць в БД.

2. Модифікація одного з фонових процесів СКБД у такий спосіб, щоб ми могли отримувати висновок про закон розподілу ймовірностей звертання до записів.

3. Написання коду, який би модифікував пошук так, щоб, за бажання користувача, СКБД використовувало отримані висновки про закони розподілу ймовірностей звертання до записів для наших даних під час пошуку, тобто власне реалізація самого ймовірнісного методу пошуку.

Наведемо результат розв’язання першої підзадачі, а саме реалізацію аналізатора статистики звертань до записів для методу пошуку інформації у файлах баз даних, який враховує розподіл імовірностей звертання до записів.

Основні результати дослідження

Опишемо спочатку сам метод пошуку інформації у файлах БД, який враховує розподіл імовірностей звертання до записів, а потім реалізацію аналізатора статистики звертань до записів для методу пошуку інформації у файлах баз даних, який враховує розподіл імовірностей звертань до записів [5].

Опишемо спочатку сам метод, а потім надбудову до СКБД, яка, власне, і дасть можливість його використовувати. Для опису алгоритму методу вводиться поняття умовно середнього запису серед записів файла [5]. Вважатимемо, що умовно середнім серед записів з порядковими номерами від m до n включно, де $1 \le m < n \le N$, є запис з номером r, якщо

$$\min_{m \le k \le n} \left| \sum_{i=m}^{k-1} p_i - \sum_{i=k+1}^{n} p_i \right|$$

досягається для $k = r$. Якщо мінімум досягається для двох індексів k, то за r приймаємо менший із них. Крім того, якщо $k = m$ і $k = n$ суми $\sum_{i=m}^{k-1} p_i$ і $\sum_{i=k+1}^{n} p_i$ є невизначеними, то вважатимемо, що $\sum_{i=m}^{k-1} p_i = 0$, якщо $k = m$; $\sum_{i=k+1}^{n} p_i = 0$, якщо $k = n$.

Розглянемо послідовний упорядкований файл, записи якого характеризуються значеннями деякого ключа. Нехай N – кількість записів файла, p_i – ймовірність звертання до i-го запису файла, K_i – значення ключа, яким характеризується i-й запис файла.

Припустимо, що у файлі потрібно знайти запис із значенням ключа *K*. Алгоритм методу складається із низки кроків. На першому кроці *K* порівнюємо зі значенням ключа запису, який є умовно середнім у файлі. Якщо порівняння успішне (два значення, що порівнюються, збігаються), то на цьому робота алгоритму завершується. Якщо два значення, що порівнюються, не збігаються, то з порівняння видно, в якій частині файла треба продовжувати пошук. Тоді на другому кроці *K* порівнюємо зі значенням ключа запису, який є умовно середнім у вибраній частині файла. У разі успішного порівняння робота алгоритму закінчується. У разі неуспішного пошук продовжується у ще меншій частині файла. І т. д. Через скінченну кількість кроків шуканий запис буде знайдений, якщо він міститься у файлі.

У разі рівномірного розподілу ймовірностей звертання до записів умовно середнім серед записів з номерами від m до n включно буде запис з номером r, де

$$r = \left[(m+n)/2\right].$$

Якщо ймовірності звертання до записів розподілені за “бінарним” законом, то умовно середнім серед записів з номерами $2k-1, 2k, 2k+1, \ldots, N$ ($k=1, 2, \ldots, \left[N/2\right]$) буде запис з номером $r = 2k$ [5].

Припустимо, що ймовірності звертання до записів розподілені за законом Зіпфа. Тоді умовно середнім серед записів з номерами від m до n включно при $n > m+1$ буде запис з номером $r = k$, де k – індекс, для якого досягається

$$\min_{m \le k \le n} \left| \sum_{i=m}^{k-1} \frac{1}{i} - \sum_{i=k+1}^{n} \frac{1}{i} \right|.$$

Якщо $n = m+1$, умовно середнім буде запис з номером m. Одержану формулу для знаходження індексу k можна замінити простішою [5]

$$k = [\sqrt{nm+1}\,].$$

Якщо ймовірності звертання до записів задовольняють узагальнений закон розподілу, то умовно середнім серед записів з номерами від m до n включно при $n > m+1$ буде запис з номером $r = k$, де k – індекс, для якого досягається

$$\min_{m \le k \le n} \left| \sum_{i=m}^{k-1} \frac{1}{i^c} - \sum_{i=k+1}^{n} \frac{1}{i^c} \right|.$$

Якщо $n = m+1$, умовно середнім буде запис з номером m. У [5] для знаходження k отримана формула

$$k = \left[\left(\frac{1}{2}\left(m^{1-c} + n^{1-c} \right) \right)^{\frac{1}{1-c}} \right].$$

Зокрема, при $c = 0$ (тобто у випадку рівномірного розподілу ймовірностей) із одержаної формули дістаємо $k = \left[(m+n)/2\right]$.

На рис. 1 наведено блок-схему алгоритму методу пошуку інформації у файлах баз даних, який враховує розподіл ймовірностей звертання до записів.

Розв’язання поставленої задачі подамо у вигляді двох підзадач:

– написання програмного коду для статистичного аналізатора, який накопичуватиме та опрацьовуватиме статистичні дані (кількість звертань до конкретних полів таблиць БД);

– реалізація алгоритму пошуку, який враховуватиме закон розподілу ймовірностей звертання до записів.

У СКБД PostgreSQL, як і в більшості систем такого типу, кожна таблиця містить певні системні стовпці, які є недоступними для простого користувача. Відповідно, для накопичення статистики було створено стовпець, в кожному елементі якого міститься кількість звернень до конкретного рядка таблиці. У цій реалізації вона отримала назву STATS. Її можна використовувати як опцію до запиту на створення таблиці. Створити таблицю тепер можна, задаючи такий запит:

CREATE TABLE *назва_табл* (*поле_1* ТИП, *поле_2* ТИП, …) WITH STATS;

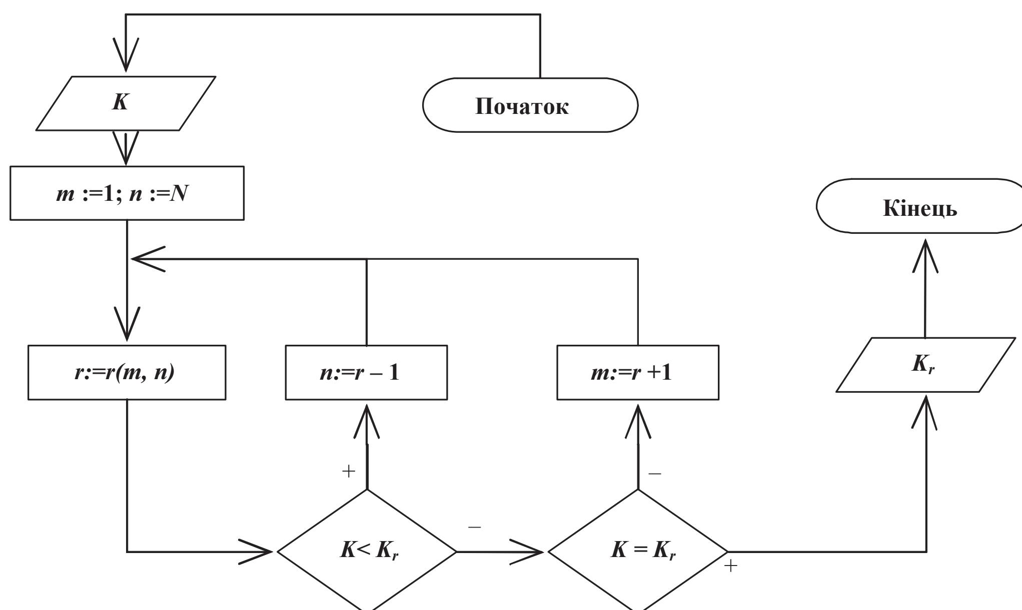

Рис. 1. Блок-схема алгоритму методу

У реалізацію додано новий відповідний конфігураційний параметр, який встановлює цю опцію за замовчуванням (**default_with_stats**). Він може набувати значення “on” та “off”. Є ще конфігураційний параметр, який, власне, визначає, чи накопичувати статистику, чи ні. Він має назву **search_stats_collector.**

Отже, тепер користувач може сам зробити висновок, чи потрібна йому ця функціональність, чи ні, і, відповідно, чи затрачати на неї обчислювальні ресурси.

Нагромадження статистичної інформації здійснюється під час виконання запиту **SELECT** у разі, коли відповідний конфігураційний параметр увімкнений і таблиця, для якої він виконується, має системну колонку **STATS.**

У майбутньому планується реалізувати другу підзадачу, тобто власне “підмінити” пошук, а також додати певну функціональність до самої системної колонки, що давало б можливість відмовлятися від неї або, навпаки, додавати її після створення таблиці, вказувати можливість введення її під час імпортування та копіювання таблиць.

Звичайно, такий підхід до опрацювання даних не завжди може бути виграшним (наприклад, для малих баз даних). Причиною цього є те, що для нагромадження та аналізу статистики потрібно буде затратити більше часу, ніж на використання простого пошуку. Проте для великих БД така функціональність зможе значно вплинути на швидкість їх роботи і забезпечити отримання таких самих результатів, але за коротший проміжок часу. Отже, зроблено крок у напрямі покращання роботи з базами даних. Цей крок є надзвичайно важливим для подальшої оптимізації пошуку інформації у великих файлах баз даних.

Реалізація

Реалізація здійснювалась поетапно згідно з вищеописаними кроками. Реалізовано першу підзадачу.

Для роботи з кодом використовувалось середовище kDevelop та операційна система Linux Ubuntu 8.10. Реалізація виконувалась так.

Написання коду, який би накопичував статистичні дані

У PostgreSQL, як і в більшості СКБД, кожна таблиця містить певні системні колонки, які зовні користувач може побачити лише за допомогою спеціального запиту. Вони призначені для накопичення даних, потрібних для функціонування СКБД. Звичайний користувач їх майже не використовує, оскільки ця інформація корисна лише для користувачів з поглибленим знанням власне самої СКБД. Проте серед них є кілька колонок, які можуть бути корисними для написання власної функціональності мовами, що підтримуються PostgreSQL.

Отже, для накопичення статистики було вирішено створити ще одну таку системну колонку, в кожному елементі якої містилася б кількість звернень до конкретного рядка таблиці. Кількість звернень нам потрібна для того, щоб можна було обчислити ймовірність звертання до кожного конкретного рядка таблиці. Така інформація необхідна для функціонування нового методу пошуку.

Нова системна колонка отримала назву **STATS**, оскільки в ній ми накопичуватимемо статистичні дані. Оскільки така колонка містить додаткову інформацію, вона звичайно вимагає певних затрат ресурсів, а саме затрат пам’яті, тому її можна додавати як опцію до запиту на створення таблиці, який тепер може мати такий вигляд:

```
CREATE TABLE table_name (table_row_1 ROW_TYPE_1, … ) WITH STATS;
```

Порівняно зі звичайним запитом ми отримуємо як результат таблицю із ще однією системною колонкою під назвою **“stats”**. Для наочності візьмемо за приклад таблицю, яка містить описи книжок, та була створена таким запитом:

```
CREATE TABLE `db`.`books` (
  `isbn` INTEGER NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`isbn`)
)
ENGINE = InnoDB;
```

Умовно наша таблиця тепер виглядатиме так, як на рис. 2.

stats	isbn	name	...
1	31896	Treasure Island	...
5	43997	The Jungle Book	...
...	...	...	...
...	...	...	...
2	42361	Othello	...

Рис. 2. Умовний вигляд таблиці з новою системною колонкою

За замовчуванням у PostgreSQL таблиця створюється з шістьма системними колонками, ще одна може додаватись як опція. У клієнтській частині цей запит виглядає так, як зображено на рис. 3.

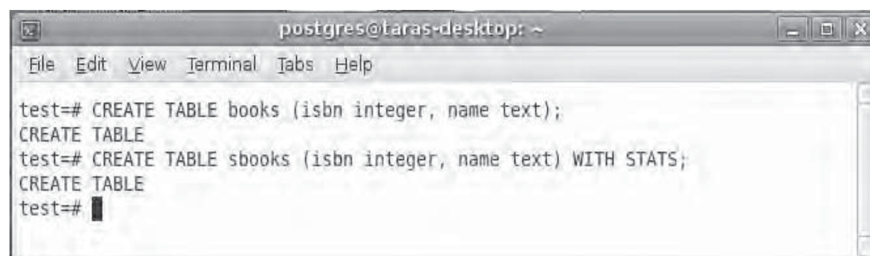

Рис. 3. Запити на створення таблиці без та з колонкою “stats”

Код на рис 3 створює дві таблиці “books” та “sbooks”. Перша відповідно є звичайною, друга містить додаткову системну колонку із іменем “stats”.

Додавання нової системної колонки реалізувалось за кілька кроків. Надалі розглянемо їх детальніше.

Модифікація граматики СКБД

Граматика у PostgreSQL будується за допомогою таких двох програм, як Lex та Yacc. Це програми, які генерують зі сформованих за певними правилами файлів, відповідно лексичний та

синтаксичний аналізатори. Причому ці дві програми працюють разом і результатом генерування фактично є компілятор, написаний мовою С. Дещо детальніше про кожну з них.

Lex – програма, яка автоматизує розроблення лексичних аналізаторів, використовуючи в своїй роботі розширені регулярні значення. Lex-програма складається з трьох частин:

```
Оголошення
%%
Правила трансляції
%%
Допоміжні підпрограми
```

Секція Оголошення містить оголошення змінних, констант та визначення регулярних виразів. Правила для побудови регулярних виразів можна знайти у специфікації самої програми.

Щодо правил трансляції Lex-програми, то вони мають вигляд:

```
p_1 { дія_1 }
p_2 { дія_2 }
................
p_n { дія_n }
```

де кожне p_i – регулярний вираз, а кожна дія_i – фрагмент програми, що описує, яку дію повинен зробити синтаксичний аналізатор, коли зразок p_i порівнюється з лексемою. В Lex дії записуються мовою С.

Третя секція містить допоміжні процедури, необхідні для дій. Ці процедури можуть транслюватися окремо і завантажуватись з лексичним аналізатором.

Лексичний аналізатор, згенерований Lex, взаємодіє з синтаксичним аналізатором так. При виклику його синтаксичним аналізатором лексичний аналізатор посимвольно читає залишок коду, поки не знаходить найдовший префікс, який може бути поставлений у відповідність одному із регулярних виразів p_i. Після цього він виконує дію_i. Як правило, дія_i повертає керування синтаксичному аналізатору. Якщо це не так, тобто у відповідній дії нема повернення, то лексичний аналізатор продовжує пошук лексем доти, доки дія не поверне керування синтаксичному аналізатору. Повторний пошук лексем аж до повного передавання керування дає змогу лексичному аналізатору правильно опрацьовувати пробіли та коментарії. Синтаксичному аналізатору лексичний аналізатор повертає єдине значення – тип лексеми. Для передавання інформації про тип лексеми використовується глобальна змінна yyval. Текстове представлення виділеної лексеми зберігає змінна yytext, а її довжину змінна yylen.

Yacc – програма, яка використовується для побудови синтаксичного аналізатора. Ця програма теж складається з трьох частин:

```
%{
С-текст
%}
%token Список імен лексем
%%
Список правил трансляції
%%
Службові С-програми
```

С-текст (який разом з дужками %{ та }%, що його оточують, може бути відсутнім) переважно містить С-оголошення (включаючи #include та #define), які використовуються в тексті нижче. Цей С-текст може містити і оголошення функції.

Список імен лексем містить імена, які Yacc-препроцесор перетворює на оголошення констант (#define). Як правило, ці імена використовуються як імена класів лексем і слугують для визначення інтерфейсу з лексичним аналізатором.

Кожне правило трансляції має вигляд

```
Ліва частина: альтернатива_1 {семантичні_дії_1}
| альтернатива_2 {семантичні_дії_2}
| …
| альтернатива_n {семантичні_дії_n}
;
```

Кожна семантична дія – це послідовність операторів С. Кожному нетерміналу може бути поставлений у відповідність один синтезований атрибут. На атрибут терміналу лівої частини посилання здійснюється за допомогою значка $$, на атрибути символів правої частини – за допомогою зачків $1, $2, … $n, причому номер відповідає послідовності елементів правої частини, включаючи семантичні дії. Кожна семантична дія може виробляти значення в результаті виконання присвоєння $$=Вираз. Виконання такого оператора в останній семантичній дії визначає значення атрибуту символу лівої частини.

Ось такий спрощений алгоритм побудови компілятора для будь-якої програми, написаної мовою С. Насправді у випадку PostgreSQL використовувались певною мірою аналоги Lex та Yacc, а саме Flex та Bison.

Результат, який ми отримуємо після розбору запиту до PostgreSQL, є дерево, яке містить вузли, що відповідають всім складовим елементам запиту.

Вхідні файли для цих програм містяться у каталозі ../postgresql/src/backend/parser/, що, власне, відповідає їх призначенню, оскільки дія перевірки граматики виконується на самому початку обробки запису.

Вхідний файл синтаксичного аналізатора будується на основі списку правил трансляції, які відповідають опису всіх можливих команд, які можуть бути подані на опрацювання PostgreSQL, а також всіх можливих різновидів цих команд. Для того щоб запит з описаним вище синтаксисом працював коректно, у цей файл було дописано, у відповідні місця, нове ключове слово STATS та можливість його використання у ролі опції до запиту на створення таблиці. Тобто у відповідне правило трансляції для запиту CREATE TABLE було додано ще два термінали :

```
| WITH STATS  {$$ = list_make1(defWithStats(true)); }
|WITHOUT STATS {$$=list_make1(defWithStats(false));
}
```

Результатом натрапляння на таку вказівку для клієнтської частини є створення додаткового вузла, який входить як дочірній вузол до дерева, що відповідає запиту CREATE TABLE. Для створення вузла було написано такий код:

```
/*
 * Create a DefElem setting "stats" to the specified value.
 */

DefElem *
defWithStats(bool value)
{
 DefElem    *f = makeNode(DefElem);
 f->defname = "stats";
 f->arg = (Node *) makeInteger(value);
 return f;
}
```

Трактування очевидне: створюємо новий DefElem, називаємо його "stats" та надаємо його значенню типу integer.

Отже, тепер ми можемо отримувати на виході дерево із ще одним вузлом, який відповідає новій системній колонці. Це дерево передається на подальше опрацювання оптимізатору (../postgresql/src/backend/optimizer/).

Потрібно звернути увагу, що запит на створення таблиці може виглядати і так:

```
CREATE TABLE books (isbn INTEGER, name TEXT) WITHOUT STATS;
```

тобто із ключовими словами "WITHOUT STATS", що вказує СКБД створити таблицю без системної колонки "stats". Для чого це потрібно, буде пояснено нижче.

Отже, це був короткий опис змін, здійснених стосовно зміни граматики PostgreSQL. Як вже зазначалось, далі запит повинен бути оптимізований перед тим, як бути переданим на безпосереднє виконання. Оптимізація ніяк не заторкує вузла, що стосується нової системної колонки, хоча він і містить додаткову інформацію, яка вимагає затрат пам'яті і не є прийнятною для оптимізатора. Однак така поведінка є правильною, оскільки користувач сам і свідомо приєднує цю колонку у вихідну таблицю.

Додавання системної колонки в таблицю

Додавання нової системної колонки вимагає розуміння процесу виконання та обробки запиту на створення таблиці. Особливо важливим моментом є робота з пам'яттю, оскільки саме цей момент роботи СКБД істотно позначається на швидкодії роботи системи загалом. У PostgreSQL робота з пам'яттю реалізована на дуже низькому рівні, що, в принципі, забезпечується вибором мови програмування. Проте вона є достатньо складною для розуміння.

Розглянемо частково процес обробки запиту CREATE TABLE.

Першим кроком після розділення запиту на структурні елементи та побудови дерева є його подальша класифікація. CREATE statement належить до так званого підтипу запитів, які називаються Utility Statements. Ця група об'єднує вирази на опрацювання створення схем та таблиць, створення транзакцій, видалення таблиць та схем, їх зміну, створення копій таблиць, індексів, кластерів, в'юх тощо. Функція на опрацювання цього типу запитів має такий прототип:

```
void
ProcessUtility(Node *parsetree,
               const char *queryString,
               ParamListInfo params,
               bool isTopLevel,
               DestReceiver *dest,
               char *completionTag);
```

У ній ми потрапляємо у великий селектор типу запиту і у разі запиту на створення таблиці ми переходимо до наступного кроку: визначення послідовності команд, які мають виконатись для того, щоб наша таблиця створилась. Ми переходимо у функцію, яка почергово опрацьовує всі дочірні вузли дерева і визначає List команд. Прототип функції має вигляд:

```
List * transformCreateStmt(CreateStmt *stmt, const char
*queryString);
```

Далі керування повертається назад у попередню функцію ProcessUtility й одержана послідовність команд передається на подальше опрацювання функції, яка, власне, створюватиме саму таблицю згідно з отриманими вимогами. Її прототип:

```
Oid DefineRelation(CreateStmt *stmt, char relkind);
```

У ній окремо опрацьовуються всі додаткові опції, які було написано користувачем у запиті, зокрема і наша опція на створення додаткової системної колонки. Функція, яка за це відповідає, має прототип:

```
Datum transformRelOptions(Datum oldOptions, List *defList,
 bool ignoreOids, bool ignoreStats, bool isReset);
```

Цю функцію було відповідно модифіковано для адекватної роботи з новою системною колонкою.

Під час роботи СКБД посилається на кортеж таблиці, з якою працює. Оскільки наш кортеж тепер повинен передбачати можливість появи нової системної колонки, його опис було змінено. У всіх місцях, де він використовувався, були теж виконані відповідні зміни. Тепер структура, яка описує кортеж таблиці, має вигляд:

```
typedef struct tupleDesc
{
 int natts; /* number of attributes in the tuple */
 Form_pg_attribute *attrs; /* attrs[N] is a pointer to the
description of Attribute Number N+1 */
 TupleConstr *constr; /* constraints, or NULL if none */
 Oid  tdtypeid;  /* composite type ID for tuple type */
 int32 tdtypmod; /* typmod for tuple type */
 bool tdhasoid;  /* tuple has oid attribute in its header */
 bool tdhasstats; /* tuple has stats attribute in its header */
 int tdrefcount; /* reference count, or -1 if not counting */
}*TupleDesc;
```

Також було додано опис власне самої системної колонки, її типу, значення за замовчуванням та деяких інших системних атрибутів, пов’язаних, зокрема, з наданням певних прав доступу до неї за замовчуванням. Опис структури, яка їй відповідає, виглядає так:

```
static FormData_pg_attribute a8 = {
 0, {"stats"}, STATSOID, 0, sizeof(Oid),
 StatisticsAccAttributeNumber, 0, -1, -1,
 true, 'p', 'i', true, true, false, true, 0
};
```

Для виділення пам'яті під нову колонку було дописано зміщення на 4 байти (саме стільки займає тип нової колонки) у певних файлах системи.

Функції, які безпосередньо вставляють та отримують значення з колонки, мають вигляд:

```
#define HeapTupleHeaderGetStats(tup) \
( \
 ((tup)->t_infomask & HEAP_HASSTATS) ? \
 *((Oid *) ((char *)(tup) + (tup)->t_hoff – sizeof(Oid)))\
 : \
 InvalidOid \
)

#define HeapTupleHeaderSetStats(tup, oid) \
do { \
 Assert((tup)->t_infomask & HEAP_HASSTATS); \
 *((Oid *) ((char *)(tup) + (tup)->t_hoff – sizeof(Oid))) = (oid); \
} while (0)
```

Це короткий опис змін, виконаних для того, щоб наша колонка справді додавалася до таблиці, щоб для неї виділялася пам'ять і щоб ми потім мали потрібну функціональність для роботи з нею.

Отже, тепер вибірка елементів та стовпців з таблиці може здійснюватись такими запитами (рис. 4).

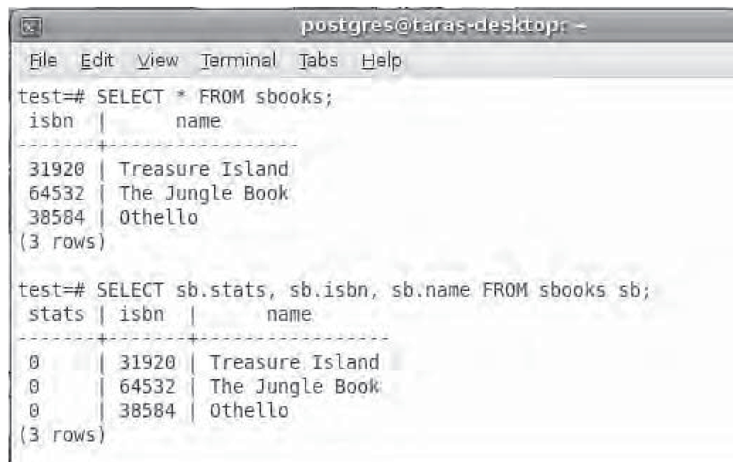

Рис. 4. Приклад вибірки елементів

Якщо потрібна вибірка будь-якої системної колонки, ми повинні вказати псевдонім таблиці і безпосередньо прописати назву самої колонки. За інших способів отримання колонок системні колонки показуватись не будуть.

Такий спосіб реалізації дає користувачу можливість самому зробити висновок, чи потрібна йому ця функціональність, чи ні і, відповідно, чи затрачати на це ресурси.

Додаткові можливості

У кожному СКБД існує можливість у певний спосіб налаштовувати його відповідно до потреб користувача. Здійснюється це за допомогою конфігураційних параметрів. Файл з початковими їх значеннями міститься в робочій директорії СКБД і зчитується ним під час запуску серверного процесу.

Конфігураційні параметри поділяються на кілька типів. Одними з них є власне параметри, які відповідають за різні опції виконання та обробки запитів.

В процесі реалізації нової функціональності було вирішено додати новий конфігураційний параметр, який би давав можливість під'єднувати опцію додавання нової системної колонки за замовчуванням. Тобто при ввімкненні цього параметра нам не потрібно би було кожен раз писати модифікований вираз для створення таблиці для того, щоб ввести колонку для накопичення статистичних даних.

Такий параметр було вирішено назвати `“default_with_stats”`, відповідно до його функціональності.

У код СКБД було додано можливість роботи з ним, а саме вмикання та вимикання його з клієнтської частини.

Для цього було додано його опис, як структури, який мав такий вигляд:

```
{
 {"default_with_stats",PGC_USERSET,
      COMPAT_OPTIONS_PREVIOUS,
      gettext_noop("Create new tables with STATSistics by default."),
      NULL
 },
 &default_with_stats,
 false, NULL, NULL
},
```

Також було додано можливість врахування його під час обробки запиту на створення таблиці. Ця можливість була реалізована у вигляді окремої функції:

```
bool
interpretStatsOption(List *defList)
{
 ListCell   *cell;

/* Scan list to see if STATS was included */
      foreach(cell, defList)
      {
           DefElem *def =
                (DefElem *) lfirst(cell);
      if(pg_strcasecmp(def->defname, "stats")==0)
           return defGetBoolean(def);
      }

 /* STATS option was not specified, so use default. */
 return default_with_stats;
}
```

У ній ми перевіряємо, чи серед отриманого опису колонок є наша колонка під назвою "stats". Якщо ні, тоді повертаємо значення конфігураційного параметра default_with_stats.

Працювати із конфігураційними параметрами ми можемо з клієнтської частини так, як показано на рис. 5.

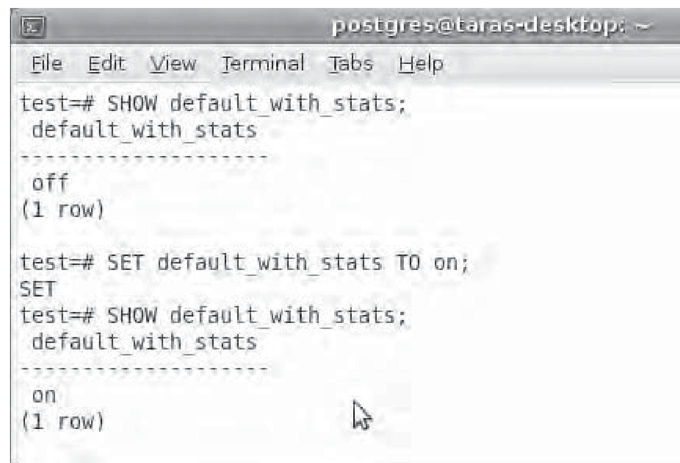

Рис. 5. Приклад роботи з конфігураційними змінними

За допомогою оператора SET … TO … ; ми можемо увімкнути чи вимкнути або просто надати значення будь-якому конфігураційному параметру. За допомогою оператора SHOW … ; ми можемо перевірити це значення.

Робота нової функціональності була продумана так.

Якщо конфігураційний параметр default_with_stats вимкнений, системну колонку ми можемо додати за допомогою запиту:

```
CREATE TABLE books (isbn INTEGER, name TEXT) WITH STATS;
```

У випадку, якщо конфігураційний параметр увімкнений, слова "WITH STATS" стають необов'язковими, оскільки тепер системна колонка створюватиметься завжди. Ця опція дає

можливість спростити написання запиту на створення таблиці у випадку, якщо клієнт хоче використовувати нововведену функціональність завжди.

Проте можуть бути випадки, коли користувач не має бажання створювати таблицю з новою колонкою, проте не хоче вимикати параметр, оскільки знає, що подальші таблиці будуть далі створюватись з колонкою "stats". В цьому випадку можна скористатися виразом, оберненим до описаного вище, який матиме вигляд:

```
CREATE TABLE books (isbn INTEGER, name TEXT) WITHOUT STATS;
```

Така гнучка функціональність забезпечує зручність у користуванні СКБД і надасть користувачу можливість вибору в будь-якому стані функціонування СКБД.

Накопичення статистики

На цьому етапі ми маємо можливість створювати таблицю із системною колонкою, значення елементів якої для кожного рядочка є обнуленими. Тепер все що потрібно – щоб у цій колонці у певний спосіб накопичувались кількість звернень до кожного рядочка. Це означає, що під час виконання запиту SELECT з умовою чи без, до елементів системної колонки "stats" кортежів, які ми отримали в результаті запиту, додається кожен раз одиничка.

Накопичення статистики здійснюється під час виконання запиту SELECT лише у тому випадку, коли відповідний конфігураційний параметр увімкнений і таблиця, для якої він виконується, має системну колонку для зберігання цих статистичних даних, тобто колонку STATS.

Ця функціональність реалізовувалась так.

Частина СКБД, яка відповідає за виконання отриманого на обробку та оптимізованого запиту, називається виконавчою (../postgres/src/backend/executor/). Саме в ній здійснюється вибірка власне самих значень рядків, які ми побачимо на екрані як результат запиту. Функція, яка здійснює цю вибірку, має прототип:

```
static bool
ExecTargetList(List *targetlist,
               ExprContext *econtext,
               Datum *values,
               bool *isnull,
               ExprDoneCond *itemIsDone,
               ExprDoneCond *isDone);
```

У ній передається керування ще двом функціям, одна з яких відповідає за видобування з пам'яті значень звичайних колонок (тобто тих, що створені користувачем):

```
Datum slot_getattr(TupleTableSlot *slot, int attnum, bool *isnull);
```

інша відповідає за видобування з памяті значень системних колонок, якщо такі вказані у запиті:

```
Datum heap_getsysattr(HeapTuple tup, int attnum, TupleDesc
tupleDesc, bool *isnull);
```

До значень системної колонки "stats" ми доступаємось за допомогою двох макросів, один з яких набуває значення відповідно до отриманого дескриптора із кортежа таблиці, інший записує значення, збільшене на одиничку, в комірку за тією самою адресою. Ці макроси мають вигляд:

```
#define HeapTupleHeaderGetStats(tup) \
( \
 ((tup)->t_infomask & HEAP_HASSTATS) ? \
      *((Oid *) ((char *)(tup) + (tup)->t_hoff - sizeof(Oid))) \
 : \
      InvalidOid \
)
```

```
#define HeapTupleHeaderSetStats(tup, oid) \
do { \
 Assert((tup)->t_infomask & HEAP_HASSTATS); \
 *((Oid *) ((char *)(tup) + (tup)->t_hoff - sizeof(Oid))) = (oid); \
} while (0)
```

На майбутнє планується реалізувати певну додаткову функціональність до самої системної колонки, яка би давала можливість відмовлятися від неї або, навпаки, додавати її аж після створення таблиці, вказувати можливість увімкнення її під час наслідування та під час копіювання таблиць.

Отримання висновку про закон розподілу ймовірностей звертання до записів

Для функціонування нового ймовірнісного методу пошуку нам необхідно знати закон розподілу ймовірностей звертання до записів. Якщо таких даних немає, то метод працюватиме набагато довше, оскільки потрібно буде робити додаткові обчислення, які різко зростають із збільшенням рядків у таблиці. Закони розподілу отримують на основі накопичених статистичних даних про звернення до кожного рядка таблиць. Оскільки одержання висновку про закон розподілу вимагає затрат робочого часу процесора та додаткових ресурсів памяті, було вирішено здійснювати його автоматично через деякий, наперед встановлений, період часу, який можна змінювати за допомогою конфігураційних параметрів СКБД. До того ж було вирішено передбачити можливість примусового виклику цієї функціональності за бажанням користувача.

Отже, плани щодо реалізації цієї підзадачі виглядають так.

PostgreSQL являє собою не лише серверний процес, а набір процесів, які разом, синхронізовано функціонують. Деякі з них є фоновими і вступають в дію через певні проміжки часу або на вимогу користувача. Отже, оскільки цей підхід дуже подібний до запропонованого вище, було вирішено реалізовувати його за допомогою дописання відповідної функціональності до одного з фонових процесів PostgreSQL.

Є декілька фонових процесів, які існують поряд з серверним процесом.

`Autovacuum Launcher Process`

Передусім це `autovacuum launcher process`. Цей процес відповідає за автоматичне виконання команд `VACUUM` та `ANALYZE` над таблицями БД. Ці команди звільняють пам'ять, потрачену на зберігання рядків таблиць, які були видалені чи оновлені; накопичують статистику, яка використовується оптимізатором СКБД для побудови оптимального плану виконання запиту та стежать за тим, щоб `transaction ID` не виходив за межі допустимих значень, приводячи систему до краху.

Отже, `autovacuum launcher process` є процесом-демоном, який автоматично приводить в дію вищеописані команди для таблиць, яким це потрібно (тобто він паралельно стежить за станом таблиць) через певний фіксований проміжок часу. Крім того, ці команди може безпосередньо виконати сам користувач у разі потреби.

`Stats Collector Process`

Існує ще `stats collector process`. Цей процес відповідає за накопичення та звітування статистики, пов'язаної з діяльністю сервера. Він вміє рахувати кількість звернень до таблиць та індексів, кількість рядків у таблиці, фіксує останній час виконання команд `VACUUM` та `ANALYZE`.

Оскільки `stats collector process` вимагає затрат процесорного часу, його можна вмикати та вимикати, відповідно до налаштувань СКБД, які можна здійснити зміною конфігураційних параметрів у конфігураційному файлі `postgresql.conf`.

Кожен серверний процес відсилає свої статистичні підрахунки до колектора якраз перед переходом у стан бездіяльності. Існує дуже багато різних в'юх та окремих функцій для роботи з статистичними даними, що забезпечує зручність у їх використанні.

`Wal Writer Process`

Ще одним фоновим процесом є `wal writer process`. `WAL` розшифровується як `Write-Ahead-Log`. Розглянемо суть роботи цього фонового процесу.

Чекпоїнти (`checkpoint`) – це такі точки, моменти в послідовності транзакцій, в які гарантовано вся інформація у файлах з даними є оновлена. Це означає, що всі брудні (тобто змінені) сторінки з даними виводяться на диск і відповідний чекпоїнт-запис робиться у лог-файлі. У разі краху системи процедури відновлення дивляться у останні чекпоїнт-записи, щоб визначити точку в логуванні (яку ще називають `redo`-записом), з якої потрібно почати виконання `REDO` операції. Відомо, що всі зміни, зроблені з даними до цієї точки, перенесені на диск. Відповідно, всі решта записи з логування, що знаходять після цієї точки вже не потрібно і вони можуть бути видалені.

Фоновий процес автоматично робить чекпоїнт, наскільки це потрібно. Переважно чекпоїнт здійснюється автоматично у разі накопичення певної кількості логів сервера або після проходження певного проміжку часу, залежно від того, що настане швидше. Також можливо примусово виконати чекпоїнт за допомогою SQL-команди `CHECKPOINT`.

Чекпоїнти є достатньо дорогими, тому налаштування системи потрібно робити розумно і кожен раз перевіряти його правильність за допомогою вбудованих можливостей системи.

Отже, можна зробити висновок, що найближчими до потрібної функціональності (отримання висновку про закон розподілу ймовірностей звертання до записів) є `autovacuum launcher process` та `wal writer process`. Тому саме один з них буде вибрано для імплементування нової функціональності.

Звичайно, можна розглянути і варіант створення ще одного фонового процесу, який би окремо вміщував у себе все необхідне, але це питання буде розглядатись пізніше, під час безпосередньої роботи над кодом.

Для отримання висновків щодо законів розподілу ми використовуватимемо матеріал з теорії ймовірностей.

Модифікація пошуку

Найважливішим та останнім є крок, на якому потрібно здійснити власне саму модифікацію методу пошуку. В PostgreSQL використовується бінарний метод пошуку.

Алгоритм самого методу розписаний детально вище. Все, що потрібне для його функціонування, – це інформація про закон розподілу ймовірностей звертання до записів.

Його функціональність планується використовувати так. У PostgreSQL вже є введений ще один конфігураційний параметр, який називається `search_stats_collector`. Ймовірнісний метод пошуку спрацьовуватиме тоді, коли цей параметр буде увімкнений і таблиця, для якої виконуватиметься запит, міститиме системну колонку `"stats"`, тобто для неї буде відомо закон розподілу ймовірностей звертання до записів.

Після реалізації самого методу пошуку потрібно буде зафіксувати оцінку швидкодії обох методів пошуку для різних наборів даних і зробити висновок про доцільність його подальшого використання.

Висновки

Зроблено крок у напрямі покращання роботи з базами даних: реалізовано накопичення статистичних даних, які враховуватимуться при пошуку. Цей крок є надзвичайно важливим, оскільки велика частина функціональності СКБД вважається вже максимально оптимізованою і важко знайти ще щось, щоби могло пришвидшити її роботу.

Ось чому будь-який крок в цьому напрямі є надзвичайно важливим та потрібним.

1. Кнут Д. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск / Д. Кнут. – М.: Изд. дом «Вильямс», 2000. – 832 с. 2. Мартин Дж. Организация баз данных в вычислительных системах / Дж. Мартин. – М.: Мир, 1980. – 644 с. 3. Цегелик Г.Г. Организация и поиск информации в базах данных / Г.Г. Цегелик. – Львов: Вища шк., 1987. – 176 с. 4. Цегелик Г.Г. Системы распределенных баз данных / Г.Г. Цегелик. – Львов: Свит, 1990. – 168 с. 5. Мельничин А.В. Ефективність методу пошуку інформації у файлах баз даних, який враховує розподіл імовірностей звертання до записів / А.В. Мельничин, М.І. Філяк, Г.Г. Цегелик // Вісн. Вінниц. політехн. ін-ту. – 2006. – № 6. – С. 187–191. 6. Цегелик Г.Г. Моделювання та оптимізація доступу до інформації файлів баз даних для однопроцесорних і багатопроцесорних систем / Г.Г. Цегелик. – Львів: Видавництво ЛНУ імені Івана Франка, 2010. – 192 с. 7. Lawrence A.R. The POSTGRES Data Model / A.R. Lawrence, M. Stonebraker // Proceedings of the 13th International Conference on Very Large Data Bases, 1987. – 518 p.