Розглянута оцінка скриптів для бази даних. Описані основні помилки та неоптимальності скриптів SQL та їх вага для оцінки і оптимізації скриптів

Ключові слова: база даних, скрипт, помилки, оптимізація, оцінка, швидкість

Рассмотрена оценка скриптов для бази данных. Описаны основные ошибки и неоптимальности скриптов SQL и их веса для оценки и оптимизации скриптов

Ключевые слова: база данных, скрипт, ошибки, оптимизация, оценка, скорость

We consider the evaluation of scripts for data bases. The basic errors and non-optimality of SQL scripts and their weights for the evaluation and optimization script are described

Keywords: database, script, errors, optimization, estimation, speed

УДК 004.415.5

АНАЛІЗ ТА ОЦІНКА СКРИПТІВ SQL

Б.П. Громюк*
Контактний тел.: (0472) 72-62-14
E-mail: gromuk.bogdan@gmail.com

А.А. Рідкокаша
Кандидат технічних наук, доцент*
Контактний тел.: (0472) 73-02-68
E-mail: redan@list.ru
*Кафедра програмного забезпечення автоматизованих систем
Черкаський державний технологічний університет
бул. Шевченко, 460, м. Черкаси, 18006

Вступ

Аналіз скриптів може бути виконаний під час тестування осіб, які навчаються або бажають працювати над програмними проектами, а також у процесі проектування програмних систем (ПС).

При прийомі на роботу, у навчанні застосовується відкрите тестування, зокрема з знань SQL і баз даних (БД). Написані скрипти перевірюється на наявність помилок і оптимальність, визначається їх оцінка.

Під час розробки ПС проводиться її тестування і оптимізація – процес змін скриптів і/або структури БД для зменшення використання обчислювальних ресурсів (збільшення швидкодії ПС і зменшення задіяної ємності запам'ятовуючих пристроїв) при виконанні запитів.

На автоматизацію цих питань спрямовують значну увагу та зусилля, зокрема [1– 4]. Але питання автоматизації повністю не вирішено.

Усі ПС, що виконують оптимізацію та аналіз скриптів (у тому числі і IBM DB2 Performance Expert for Multiplatforms [5]), можуть представити лише альтернативні варіанти формування скрипту або групи скриптів і не можуть мати вплив на саму систему та структуру БД.

Постановка задачі

Потрібно розробити тестову ПС (ТПС), що виконує тестування та оптимізацію.

Кінцеву, чисельну та неупорядковану множину всіх службових слів (СС) мови SQL подамо як C $C = \{c_i : 1 \le i \le n\}$, де c_i – СС, n – ціле число.

Подамо також сукупність СС деякої наявної або потрібної БД як множину T, яка є частиною множини C і має всі її властивості, тобто

$$T \subset C; T = \{t_i : 1 \le i \le m\},$$

де m – ціле та m<n.

Особа, яку тестують, повинна з використанням СС множини T написати задані скрипти, тобто у відповіді сформувати підмножини скриптів БД p(T) і запитів до БД p(S), що потім перевіряються на наявність помилок і визначається близкість скриптів. Наприклад, для скрипта-еталона e та скрипта-відповіді b можна знайти кількісний показник їхньої близькості $k = \gamma(e,b)$, значення якого знаходяться в інтервалі [0, 1]. Властивості та обмеження функції γ при використанні:

– якщо $k = \gamma(e,b) = 1$, то маємо повну відповідність e та b;

– якщо $k = \gamma(e,b) > 0{,}5$, то маємо неповну відповідність e та b;

– якщо $k = \gamma(e,b) \le 0{,}5$, то маємо деяку відповідність e та b;

– якщо $k = \gamma(e,b) = 0$, то маємо повну невідповідність e та b.

При прийомі на роботу беруться до уваги верхні дві оцінки, при навчанні – всі.

При оптимізації з множини T розробником сформовані підмножини скриптів БД p(T) і запитів до БД p(S). Підмножини їх оптимізованих версій позначимо відповідно op(T) та op(S). При оптимізації можливе використання СС множини C, тому потужність множини може бути збільшеною і $|op(T)| \ge |p(T)|, |op(S) \ge |p(S)|$.

Відношення α між p(T) та op(T), яке описує зміни потрібного апаратного ресурсу, маскується різним рівнем наповнення БД, тому його розглядати не будемо.

Зменшення часу на виконання запитів op(S) врахуємо за допомогою функції β, що для двох односпрямованих скриптів запиту надає тренд швидкодії $\beta \equiv p(S) \to op(S)$.

Основний зміст

Основними етапами обробки тестового скрипту-відповіді є перевірка та оптимізація. Ці дії виконує

розроблена ТПС (рис. 1), результати дії β –версії якої наводяться далі.

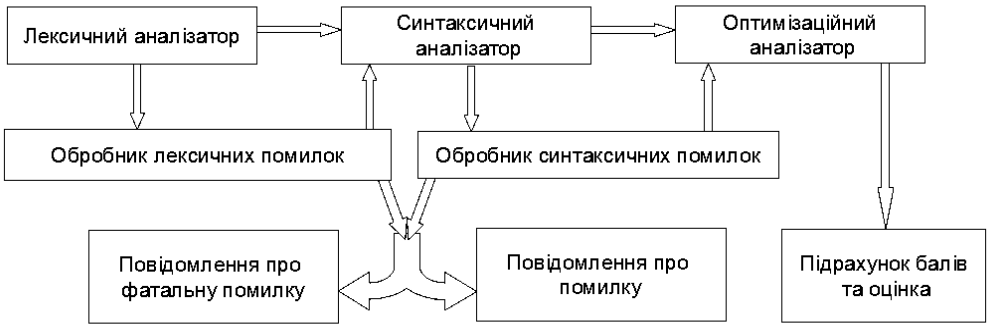

Рис. 1. Блок схема ТПС

При перевірці проводиться лексичний та синтаксичний аналіз – скрипт розбиваєься на його синтаксичні компоненти, перевіряються помилки у написанні. Якщо у скрипті запит, то ТПС вибирає об'єкти з каталогу БД та формує внутрішнє представлення запиту, ця форма внутрішнього представлення буде використана на наступних етапах.

Одними з основних помилок є лексичні і синтаксичні. Крім помилок невірного написання операторів SQL, невірного порядку команд і так далі, є й такі, що не є фатальними, але при певних умовах такий скрипт може крім зменшення швидкості обробки викликати помилку при виконанні.

Найпростіші помилки

Щоб дані відрізнити від СС SQL, імена полів і таблиць слід укладати в зворотні одинарні лапки – "` ". Адже ім'я поля може співпадати з СС, але якщо ми використаємо зворотні лапки, то ТПС зрозуміє все правильно:

SELECT * FROM `table` WHERE `date` = '2006-04-04 ';

Наприклад, у скрипті SELECT * FROM `table` WHERE name = Bill;

ТПС сприйме Bill як ім'я іншого поля, не знайде його, і видасть помилку. Тому дані, що підставляються (Bill), треба брати в лапки.

Однак, і в самих даних теж зустрічаються лапки. Приміром,

SELECT * FROM `table` WHERE name = 'комп'ютер';

Тут ТПС вирішить, що 'комп' – це дані, а ютер – СС, якого нема в множині Т, і теж видасть помилку. Щоб пояснити, що лапки (і деякі інші спецсимволи) відносяться до даних, треба поставити слеш,. В результаті отримаємо вірний запит, який помилок не викличе:

SELECT * FROM `table` WHERE name = 'комп \' ютер';

При **оптимізації** знаходяться помилки у структурі БД, запитів, тобто скрипт написаний вірно, але результат запиту формується повільно, через неоптимальність по кількості операцій запису та зчитування під час його виконання. Основна ідея синтаксичної оптимізації (syntax optimization) – використання еквівалентних алгебраїчних перетворень.

При оптимізації виконуються три етапи – логічна оптимізація, вибір оптимального плану, формування машинних кодів.

На етапі логічної оптимізації виконується перетворення скрипта запиту в оптимальний формат з точки зору синтаксису.

При виборі оптимального плану формується декілька альтернатив для їх виконання. Потім ТПС обирає кращий план запиту згідно статистики або евристичних правил. Найчастіше обирається "найдешевша" альтернатива (менше ресурсів та часу виконання).

На етапі формування машинних кодів вибраний план переформовується у машинні коди для подальшого виконання. Далі формується відповідь і пересилається користувачу.

Приклад

Розглянемо запит, який робить вибірку даних з таблиць Product та Vendor:

SELECT Vendor_Code, Product_Code, Product_Desc FROM Vendor, Product WHERE Vendor.Vendor_Code = Product.Vendor_Code AND Vendor.Vendor_Code = "100";

Найбільш очевидний шлях обробки цього запиту полягає в наступному:

– формуємо декартовий добуток таблиць Product і Vendor;

– обмежуємся в результуючій таблиці рядками, які задовольняють умові пошуку в реченні WHERE;

– виконуємо проекцію результуючої таблиці на список полів, вказаний у реченні SELECT.

Оцінимо вартість процесу обробки цього запиту в операціях введення / виведення. Нехай таблиця Ven-dor містить 50 рядків, а таблиця Product – 1000 рядків. Тоді для формування декартового добутку потрібно 50000 операцій читання і операцій запису (в результуючу таблицю). Для обмеження результуючої таблиці потрібно більше 50000 операцій читання і, якщо 20 рядків задовольняють умовам пошуку, то потрібно 20 операцій запису. Виконання операції проекції викличе ще 20 операцій читання і 20 операцій запису. Таким чином, для обробки цього запиту системі потрібно 100080 операцій читання і запису.

Для даного прикладу запиту можна використовувати таку еквівалентність: (A JOIN B) WHERE restriction ON A OR (A WHERE restriction ON A) JOIN B. Це означає, що обмеження за умовою пошуку може бути виконано якомога раніше для того, щоб обмежити число рядків, які можуть бути оброблені пізніше. Застосовуючи це правило до запиту, наведеного вище, отримуємо наступний процес обробки запиту:

1. Обмеження за умовою пошуку в другій таблиці (Vendor_Code = "100") призведе до 50 операцій читання і 20 операцій запису.

2. Для виконання з'єднання отриманої на 1-му кроці результуючої таблиці з таблицею Product потрібно 20 операцій читання результуючої таблиці, 100 операцій читання з таблиці Product і 20 операцій запису в нову результуючу таблицю.

Обробка запиту в цьому випадку потребуватиме 110 операцій читання і запису для отримання того ж самого результату, що і в першому випадку.

Щоб уникнути перевантаження серверу БД великою кількостю операцій зчитування та запису необхідно використовувати спеціальні реляційні операції. Будемо виконувати Проекцію, Вибір, З'єднання.

Операція проекції (Projection) обмежує число полів таблиці, на які є посилання в команді SELECT name, phone FROM customer.

Операція вибору або обмеження (Selection or restriction) обмежує результуючу множину тільки тими рядками, які задовольняють умові пошуку (SELECT * FROM Customer WHERE Name = 'Jones';)

Операція з'єднання (Join) створює результуючу множину за допомогою конкатенації рядків декількох таблиць. Ці зчеплені рядки повинні задовольняти деякій умові з'єднання.

З'єднання створює результуючу множину з двох або більше таблиць таким же чином, що і декартовий добуток. Воно здійснює конкатенацію рядків для кожного рядка однієї таблиці з кожним рядком іншої таблиці. Відмінність між декартовим добутком і з'єднанням полягає в тому, що в цю операцію залучаються тільки ті рядки, які задовольняють умові з'єднання.

Це є логічним еквівалентом декартового добутку, для якого була виконана операція селекції за умовою.

Проте операція з'єднання реалізується більш ефективно більшістю СУБД, так як оцінювання рядків виконується до первинного формування декартового добутку як проміжного результату.

SELECT P.Name, O.Order_num, C.NameFROM Customer C, Order O, Product PWHERE (C. Cust_no = O. Cust_no) AND (P. Cust_no = O. Cust_no);

Дві найважливіших характеристики операції з'єднання – то, що вона є комутативною і асоціативною:

A JOIN B = B JOIN A (комутативність); A JOIN (B JOIN C) = (A JOIN B) JOIN C (асоціативність).

Дія цих властивостей допомагає в обробці з'єднань більш ніж двох таблиць, коли можна вибрати будь-яку послідовність з'єднань двох таблиць для завершення операції.

Це дає можливість виконувати з'єднання будь-якого числа таблиць як серію з'єднань двох таблиць, що дозволяє уникати виконання специфічних фізичних операцій при з'єднанні довільного числа таблиць.

Два найбільш відомі алгоритми виконання з'єднання наведені нижче.

З'єднання за допомогою вкладеного циклу (Nested Loop Join). У цьому алгоритмі рядок читається з першої таблиці – зовнішньої (outer) таблиці, і потім читається кожен рядок другої таблиці – внутрішньої (inner), як кандидата для з'єднання. Потім читається другий рядок першої таблиці і знову кожен рядок з другої, і так до тих пір, поки всі рядки першої таблиці не будуть прочитані. Якщо у першій таблиці знаходиться M рядків, у другій – N, то читається M x N рядків;

З'єднання за допомогою об'єднання (Merge Join). Цей метод виконання з'єднання передбачає, що обидві таблиці розсортовані таким чином, що рядки читаються в порядку значень поля (полів), за якими вони з'єднуються. Це дозволяє виконувати з'єднання за допомогою читання рядків з кожної таблиці та порівнювання значень полів з'єднання до тих пір, поки відповідність цих значень має місце. У цьому способі з'єднання завершується за один прохід по кожній таблиці.

Також важливим моментом є використання в скриптах індексованих полів. Використання техніки індексування збільшує складність обробки запиту. Наприклад, якщо таблиця має індекси за трьома різними полями, то кожне з них може бути використаним для доступу до таблиці (крім послідовного доступу до таблиці у фізичному порядку розташування рядків).

Наведений нижче опис показує, що створювана ТПС суттєво покриє всі оціночні та оптимізаційні функції.

Структура аналізатора ТПС (табл. 1) складається з трьох обробників:

- лексичний аналізатор – перевіряє написання операторів а також послідовність їх написання;
- синтаксичний аналізатор – перевірка позиції даних та назв об'єктів бази даних, а також синтаксису самих даних;
- оптимізаційний аналізатор – аналізує скрипт та формує рекомендації щодо його покращення.

Таблиця 1

Структура аналізатора ТПС

Лексичні	Синтаксичні	Оптимізація
Помилки в написанні операторів	Відсутність лапок " ' " у даних	Використання * в операторі SELECT
Невірний порядок операторів та даних	Відсутність лапок " ` "у об'єктів	Використання складної умови замість з'єднання
Перевірка регістру скрипта	Відсутність екранування спеціальних символів	Постійне звернення до об'єкта замість складної передачі
	Помилки приведення до типу	Використання математичних операцій в скрипті
		Використання оператора OUTER JOIN
		Пошук по не індексованому полю

Найскладніша (та основна) частина ТПС – це перевірка ланцюга операторів, він декларує положення того чи іншого оператора і перевіряє скрипт відповідно до набору ланцюгів, якщо ланцюг не виявлений.

Приклад ланцюга: SELECT → FROM [→ WHERE] [→ ORDER BY].

На місці стрілок в ланцюгу стоять відповідно дані або назви об'єктів. У ТПС це запис, що посилається на певний оператор зі списку констант та на тип даних (на місці стрілок).

Усі інші перевірки – це прості умови на наявність певного символу.

Перевірка на оптимальність запиту з умовою є найскладнішою задачею для ТПС, але у цьому випадку можна вивести користувачу час виконання запиту та рекомендацію щодо використання з'єднань, що допоможуть зменшити кількість операцій запису/ зчитування.

Крім ланцюгів, необхідно також описати структури, що допоможуть визначити умови та арифметичні операції. У випадку з умовами в скриптах опис є доволі складною задачею. Але найпродуктивнішим рішенням є пошук умовних операторів та покрокове їх видалення з кеш-скрипту при правильному значенні комбінації і так до повного видалення умов або виявлення помилки. Арифметичні операції знаходяться за допомогою пошуку у скрипті відповідних операторів (і з подальшим зауваженням розробнику).

Коли не знайдено операторів та є неправильна послідовність символів, подальший аналіз неможливий і має сенс зниження оцінки k.

Наведемо приклади, що показують швидкодію описаних вище методів та інформацію у вигляді логів, що формує ТПС.

У всіх прикладах виконується оператор COUNT(*), що допомагає показати "чисту" швидкість без затрат на формування результуючої таблиці. ТПС працювала на комп'ютері конфігурації: процесор – Intel Dual-Core E5400 4 GHz, оперативна пам'ять – DDR2 1300 MHz, сервер БД – mysql-5.5.17-win32.

1. Швидкість вибірки 10 та 200 елементів майже не відрізняється, це підтверджує, що запит здійснює перебір усіх значень таблиці table_big.

```
SQL>SQL> SELECT COUNT(*)
FROM table_big WHERE Rownum <
10;

 COUNT(*)
----------
    9
Executed in 0,015 seconds
```

```
SQL>
SQL> SELECT COUNT(*) FROM
table_big WHERE Rownum < 200;

 COUNT(*)
----------
   199
Executed in 0,016 seconds
```

2. Два логи як приклад того, що умови повинні бути правильно роставлені відповідно до вірогідності роботи умов.

```
SQL>
SQL> SELECT COUNT(*) FROM
Table(expand_big) t
 WHERE t.id NOT IN
      (
      SELECT t1.id FROM
      Table(expand_small) t1
       )
 AND Rownum < 10;
 COUNT(*)
----------
    9
Executed in 0,578 seconds
```

```
SQL>
SQL> SELECT COUNT(*) FROM
TABLE(expand_big) t
WHERE t.id NOT IN
        (
        SELECT t1.id FROM
        TABLE(expand_small) t1
         )
 AND Rownum < 200;
 COUNT(*)
----------
   199
Executed in 12,063 seconds
```

Різниця в часі виконання цих двох запитів зумовлена тим, що складна умова виконується при кожному входженні в елемент при перебірці.

3. Приклад використання індексованих полів та полів без індексу

```
SQL>
SQL> SELECT /*+ INDEX(table1
ix_table1)*
COUNT(*) FROM table1;

 COUNT(*)
----------
 1099998
Executed in 0,172 seconds
```

```
SQL>
SQL> SELECT /*+ NO_INDEX(table1
ix_table1) */
COUNT(*) FROM table1;

 COUNT(*)
----------
 1 099998
Executed in 0,063 seconds
```

Використання складних індексів по двом і більше полям з малою унікальністю поля робить запит повільнішим ніж запит з умовами.

4. Наступні два логи показують, що для з'єднання важлива не індексованість поля з'єднання, а кількість з'єднуваних полів.

```
SQL>
SQL> SELECT COUNT(*) FROM
table1 t1 INNER JOIN table2 t2 ON
t2.codea = t1.codea
AND t2.codeb = t1.codeb;
 COUNT(*)
----------
  99999
Executed in 0,078 seconds
```

```
SQL>
SQL> SELECT COUNT(*) FROM
table1 t1 INNER JOIN table2 t2 ON
t2.id = t1.id;

 COUNT(*)
----------
  99999
Executed in 0,079 seconds
```

Невелика різниця в швидкості зумовлена малою кількістю даних (100000 записів) і буде збільшуватися при збільшені кількості записів. Також з'єднання сильно залежить від типу полів, по яким проводиться з'єднання. З'єднання також виконує функції формування результуючої таблиці, тому, якщо запит не буде з оператором COUNT(*), а з певними полями, то з'єднання покаже свій результат ще більш явно.

ТПС надає можливості виконання запитів з різними планами для набору статистики, наприклад, за статистикою у веб-проектах для покращення швидкодії всієї ПС потрібно виключити автоматичні формувачі скриптів.

Ці формувачі збільшують кількість малих запитів до БД, це виникає тому, що бібліотеки запитів робляться універсальними, а отже є досить повільними. Виключення формувачів запитів допомагає сконструювати оптимальний запит, враховуючи особливості предметної області, а також налаштування ПС.

Також за статистикою всі СС повинні бути в верхньому регістрі, невиконання цієї умови не призведе до виникнення критичної помилки, але зменшить швидкість за рахунок додаткової обробки внутрішнім оптимізатором СУБД.

Висновки

Максимальний ефект у формуванні правильних запитів буде за наявності системного підходу до ПС.

Виходячи з цього, потрібно загалом писати скрипти і оцінювати швидкодію системи. Описані в статті помилки та варіанти їх вирішення допоможуть здійснювати аналіз та оцінку скриптів. Постійний контроль за скриптами, що діють в проекті, допоможе майже повністю вирішити проблеми швидкості та системних помилок під час виконання скрипта на великих проектах.

Література

1. Jarke, M. Query Optimization in Database Systems [Текст] / M. Jarke, J. Koch // Computing Surveys. – 1984. – Vol. 16, № 2 – P. 111-152.
2. Білоусова, Л. І. Потенціал комп'ютерного тестування [Текст] / Л. І. Білоусова // Вісник ТІМО. – 2008. – №10. – С. 40-44.
3. Фісун, М. Т. Розробка синтаксичного аналізатора мови програмування PL/I для реінженірінгу блок-схем алгоритмів [Текст] / М.Т. Фісун, О.В. Гнездьонова // Проблеми програмування. – 2006. – №2-3. – С. 617-625.
4. Моисеенко, С. SQL. Задачи и решения. Интерактивный учебник [Електронній ресурс] – Режим доступу: http://www.sql-tutorial.ru/ru/content.html.
5. Интерфейс Ltd. Internet&Software Company – Режим доступу: http://www.interface.ru/home.asp.

Застосування принципів локально-паралельної обробки до клітинних автоматів дозволяє суттєво підвищити розміри моделей. Запропоновано і продемонстровано на тестових прикладах локально-паралельний стековий алгоритм бінарного клітинного автомату

Ключові слова: клітинні автомати, локальна паралельність, стек

Применение принципов локально-параллельной обработки к клеточным автоматам позволяет существенно увеличить размеры моделей. Предложен и продемонстрирован на тестовых примерах локально-параллельный стековый алгоритм бинарного клеточного автомата

Ключевые слова: клеточные автоматы, локальная параллельность, стек

Application of principle of local-parallel processing to cellular automaton allows greatly enlarge the sizes of models. Local-parallel stacking algorithm of the binary cellular automaton is offered and demonstrated on test examples

Keywords: cellular automatons, local parallelism, stack

УДК 519.87

СТЕКОВЫЙ АЛГОРИТМ ДЛЯ ЛОКАЛЬНО-ПАРАЛЛЕЛЬНОГО БИНАРНОГО КЛЕТОЧНОГО АВТОМАТА

Бенаддия Абдельлатиф
Аспирант*

О.Ф.Михаль
Доктор технических наук, доцент, профессор*
*Кафедра Электронно-вычислительных машин
Харьковский национальный университет радиоэлектроники
пр. Ленина, 14, г. Харьков, 61166
Контактный тел. (057) 40-93-54
E-mail: fuzzy16@pisem.net

1. Введение

Моделирование крупномасштабных систем с распределённой обработкой информации актуально в связи с изучением динамики процессов, происходящих в человеческом окружении, на различных уровнях от глобальных климатических явлений до живой природы и социальных структур (экология, климатология, демография, социология). Общность подхода состоит в том, что система рассматривается как состоящая из большого числа дискретных элементов (ячеек), взаимодействующих друг с другом в соответствии с фиксированным наборами правил. Перспективным аппаратом компьютерного моделирования подобных систем являются *клеточные автоматы* (КА). Распространённость моделирования поведения систем на КА обусловлена простотой описания конфигурации и набора рабочих правил. Дополнительный выигрыш по эффективности работы КА может быть обеспечен *локально-параллельными* (ЛП) алгоритмами [1], которые разработаны [2, 3] для целочисленных неотрицательных значений операндов, что применимо к описанию ячеек КА. В частности, если ячейка принимает одно из значений (0, 1), это соответствует подвиду КА – *бинарным клеточным автоматам* (БКА) [4]. Для полноты оперирования данными, необходимо дополнить аппарат описания [2, 3] ЛП алгоритмами, реализующими